cloudera®

Apache Impala Guide

Important Notice
© 2010-2020 Cloudera, Inc. All rights reserved.

Cloudera, the Cloudera logo, and any other product or
service names or slogans contained in this document are trademarks of Cloudera and its suppliers or licensors, and may not be copied, imitated or used, in whole or in part, without the prior written permission of Cloudera or the applicable trademark holder. If this documentation includes code, including but not limited to, code examples, Cloudera makes this available to you under the terms of the Apache License, Version 2.0, including any required notices. A copy of the Apache License Version 2.0, including any notices, is included herein. A copy of the Apache License Version 2.0 can also be found here: https://opensource.org/licenses/Apache-2.0

Hadoop and the Hadoop elephant logo are trademarks of the Apache Software Foundation. All other trademarks, registered trademarks, product names and company names or logos mentioned in this document are the property of their respective owners. Reference to any products, services, processes or other information, by trade name, trademark, manufacturer, supplier or otherwise does not constitute or imply endorsement, sponsorship or recommendation thereof by us.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Cloudera.

Cloudera may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Cloudera, the furnishing of this document does not give you any license to these patents, trademarks copyrights, or other intellectual property. For information about patents covering Cloudera products, see http://tiny.cloudera.com/patents.

The information in this document is subject to change without notice. Cloudera shall not be liable for any damages resulting from technical errors or omissions which may be present in this document, or from use of this document.

Cloudera, Inc.
395 Page Mill Road
Palo Alto, CA 94306
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information

Version: CDH 6.3.x
Date: October 30, 2020

Table of Contents

Introducing Apache Impala....16

Impala Benefits....16

How Impala Works with CDH....16

Primary Impala Features....17

Impala Concepts and Architecture....18

Components of the Impala Server....18

The Impala Daemon....18

The Impala Statestore....18

The Impala Catalog Service....19

Developing Impala Applications....20

Overview of the Impala SQL Dialect....20

Overview of Impala Programming Interfaces....21

How Impala Fits Into the Hadoop Ecosystem....21

How Impala Works with Hive....21

Overview of Impala Metadata and the Metastore....22

How Impala Uses HDFS....22

How Impala Uses HBase....22

Planning for Impala Deployment....23

Impala Requirements....23

Product Compatibility Matrix....23

Supported Operating Systems....23

Hive Metastore and Related Configuration....23

Java Dependencies....23

Networking Configuration Requirements....24

Hardware Requirements....24

User Account Requirements....25

Guidelines for Designing Impala Schemas....25

Setting Up Apache Impala Using the Command Line....27

What is Included in an Impala Installation....27

Installing Impala from the Command Line....27

Modifying Impala Startup Options....29

Configuring Impala Startup Options through the Command Line....29

Checking the Values of Impala Configuration Options 32
Startup Options for impalad Daemon 32
Startup Options for statestored Daemon 32
Startup Options for catalogd Daemon 32
Starting Impala 32
Starting Impala from the Command Line 33
Installing Impala with Cloudera Manager 33
Installing Impala from the Command Line 34

Managing Impala 36
Post-Installation Configuration for Impala 36

Impala Upgrade Considerations 38
Converting Legacy UDFs During Upgrade to CDH 5.12 or Higher 38
Handling Large Rows During Upgrade to CDH 5.13 / Impala 2.10 or Higher 38
Change Impala catalogd Heap when Upgrading from CDH 5.6 or Lower 39
List of Reserved Words Updated in CDH 6.0 / Impala 3.0 40
Decimal V2 Used by Default in CDH 6.0 / Impala 3.0 40
Behavior of Column Aliases Changed in CDH 6.0 / Impala 3.0 40
Default PARQUET_ARRAY_RESOLUTION Changed in CDH 6.0 / Impala 3.0 40
Enable Clustering Hint for Inserts 40
Deprecated Query Options Removed in CDH 6.0 / Impala 3.0 40
refresh_after_connect Impala Shell Option Removed in CDH 6.0 / Impala 3.0 41
Return Type Changed for EXTRACT and DATE_PART Functions in CDH 6.0 / Impala 3.0 41
Impala Roles with SELECT or INSERT Privilege Receive REFRESH Privilege During the Upgrade to CDH 5.16 / CDH 6.1 41
Port Change for SHUTDOWN Command 41
Change in Client Connection Timeout 41
Default Setting Changes 42

Impala Tutorials 43
Tutorials for Getting Started 43
Explore a New Impala Instance 43
Load CSV Data from Local Files 48
Point an Impala Table at Existing Data Files 49
Describe the Impala Table 51
Query the Impala Table 51
Data Loading and Querying Examples 52
Advanced Tutorials 54
Attaching an External Partitioned Table to an HDFS Directory Structure 54

Switching Back and Forth Between Impala and Hive 56
Cross Joins and Cartesian Products with the CROSS JOIN Operator 57
Dealing with Parquet Files with Unknown Schema 58
Download the Data Files into HDFS 59
Create Database and Tables 59
Examine Physical and Logical Schema 60
Analyze Data 61

Impala Administration 70

Setting Timeout Periods for Daemons, Queries, and Sessions 70
Increasing the Statestore Timeout 70
Setting the Idle Query and Idle Session Timeouts for impalad 70
Setting Timeout and Retries for Thrift Connections to the Backend Client 71
Cancelling a Query 72
Using Impala through a Proxy for High Availability 72
Overview of Proxy Usage and Load Balancing for Impala 72
Choosing the Load-Balancing Algorithm 73
Special Proxy Considerations for Clusters Using Kerberos 74
Special Proxy Considerations for TLS/SSL Enabled Clusters 75
Example of Configuring HAProxy Load Balancer for Impala 75
Managing Disk Space for Impala Data 77
Auditing Impala Operations 78
Durability and Performance Considerations for Impala Auditing 79
Format of the Audit Log Files 79
Which Operations Are Audited 80
Reviewing the Audit Logs 80
Viewing Lineage Information for Impala Data 80

Impala Security 82

Security Guidelines for Impala 82
Securing Impala Data and Log Files 83
Installation Considerations for Impala Security 84
Securing the Hive Metastore Database 84
Securing the Impala Web User Interface 84
Configuring TLS/SSL for Impala 85
Using Cloudera Manager 85
Configuring TLS/SSL Communication for the Impala Shell 86
Using TLS/SSL with Business Intelligence Tools 86
Specifying TLS/SSL Minimum Allowed Version and Ciphers 87
Enabling Sentry Authorization for Impala 87
The Sentry Privilege Model 87
Starting the impalad Daemon with Sentry Authorization Enabled 90

Enabling Sentry for Impala in Cloudera Manager.....91
Using Impala with the Sentry Service (CDH 5.1 or higher only).....91
Setting Up Schema Objects for a Secure Impala Deployment.....94
The DEFAULT Database in a Secure Deployment.....94
Impala Authentication.....94
Enabling Kerberos Authentication for Impala.....94
Enabling LDAP Authentication for Impala.....96
Using Multiple Authentication Methods with Impala.....98
Configuring Impala Delegation for Hue and BI Tools.....99

Impala SQL Language Reference.....101
Comments.....101
Data Types.....101
ARRAY Complex Type (CDH 5.5 or higher only).....102
BIGINT Data Type.....105
BOOLEAN Data Type.....106
CHAR Data Type (CDH 5.2 or higher only).....107
DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only).....109
DOUBLE Data Type.....114
FLOAT Data Type.....116
INT Data Type.....117
MAP Complex Type (CDH 5.5 or higher only).....118
REAL Data Type.....121
SMALLINT Data Type.....122
STRING Data Type.....123
STRUCT Complex Type (CDH 5.5 or higher only).....124
TIMESTAMP Data Type.....130
TINYINT Data Type.....136
VARCHAR Data Type (CDH 5.2 or higher only).....137
Complex Types (CDH 5.5 or higher only).....139
Literals.....167
Numeric Literals.....167
String Literals.....168
Boolean Literals.....169
Timestamp Literals.....169
NULL.....170
SQL Operators.....171
Arithmetic Operators.....171
BETWEEN Operator.....174
Comparison Operators.....175
EXISTS Operator.....176
ILIKE Operator.....179
IN Operator.....180
IREGEXP Operator.....182

IS DISTINCT FROM Operator *183*
IS NULL Operator *185*
IS TRUE Operator *186*
LIKE Operator *186*
Logical Operators *187*
REGEXP Operator *189*
RLIKE Operator *191*
Impala Schema Objects and Object Names 191
Overview of Impala Aliases *192*
Overview of Impala Databases *193*
Overview of Impala Functions *194*
Overview of Impala Identifiers *195*
Overview of Impala Tables *196*
Overview of Impala Views *199*
Impala SQL Statements 202
DDL Statements *203*
DML Statements *204*
ALTER DATABASE Statement *204*
ALTER TABLE Statement *205*
ALTER VIEW Statement *218*
COMMENT Statement *219*
COMPUTE STATS Statement *219*
CREATE DATABASE Statement *226*
CREATE FUNCTION Statement *228*
CREATE ROLE Statement (CDH 5.2 or higher only) *234*
CREATE TABLE Statement *234*
CREATE VIEW Statement *248*
DELETE Statement (CDH 5.10 or higher only) *249*
DESCRIBE Statement *251*
DROP DATABASE Statement *262*
DROP FUNCTION Statement *263*
DROP ROLE Statement (CDH 5.2 or higher only) *265*
DROP STATS Statement *265*
DROP TABLE Statement *268*
DROP VIEW Statement *270*
EXPLAIN Statement *271*
GRANT Statement (CDH 5.2 or higher only) *273*
INSERT Statement *277*
INVALIDATE METADATA Statement *286*
LOAD DATA Statement *288*
REFRESH Statement *291*
REFRESH AUTHORIZATION Statement *293*
REFRESH FUNCTIONS Statement *293*
REVOKE Statement (CDH 5.2 or higher only) *293*
SELECT Statement *295*

SET Statement *321*
SHOW Statement *363*
SHUTDOWN Statement *379*
TRUNCATE TABLE Statement (CDH 5.5 or higher only) *381*
UPDATE Statement (CDH 5.10 or higher only) *383*
UPSERT Statement (CDH 5.10 or higher only) *384*
USE Statement *385*
VALUES Statement *386*
Optimizer Hints in Impala *387*
Impala Built-In Functions 391
Impala Mathematical Functions *397*
Impala Bit Functions *414*
Impala Type Conversion Functions *423*
Impala Date and Time Functions *424*
Impala Conditional Functions *457*
Impala String Functions *462*
Impala Miscellaneous Functions *477*
Impala Aggregate Functions *479*
Impala Analytic Functions *506*
User-Defined Functions (UDFs) 525
UDF Concepts *525*
Runtime Environment for UDFs *528*
Installing the UDF Development Package *528*
Writing User-Defined Functions (UDFs) *529*
Writing User-Defined Aggregate Functions (UDAFs) *532*
Building and Deploying UDFs *533*
Performance Considerations for UDFs *535*
Examples of Creating and Using UDFs *535*
Security Considerations for User-Defined Functions *540*
Limitations and Restrictions for Impala UDFs *540*
Converting Legacy UDFs During Upgrade to CDH 5.12 or Higher *541*
SQL Differences Between Impala and Hive 541
HiveQL Features not Available in Impala *541*
Semantic Differences Between Impala and HiveQL Features *542*
Porting SQL from Other Database Systems to Impala 543
Porting DDL and DML Statements *543*
Porting Data Types from Other Database Systems *543*
SQL Statements to Remove or Adapt *546*
SQL Constructs to Double-check *547*
Next Porting Steps after Verifying Syntax and Semantics *548*

Resource Management 549

Admission Control and Query Queuing 549
Concurrent Queries and Admission Control *549*

Memory Limits and Admission Control.....550
How Impala Admission Control Relates to Other Resource Management Tools.....552
How Impala Schedules and Enforces Limits on Concurrent Queries.....552
How Admission Control works with Impala Clients (JDBC, ODBC, HiveServer2).....553
SQL and Schema Considerations for Admission Control.....553
Guidelines for Using Admission Control.....554
Configuring Resource Pools and Admission Control.....554
Creating Static Service Pools.....555
Using Admission Control.....555
Setting Per-query Memory Limits.....558
Configuring Admission Control in Command Line Interface.....559
Configuring Cluster-wide Admission Control.....560
Admission Control Sample Scenario.....562

Tuning Impala for Performance.....565
Impala Performance Guidelines and Best Practices.....565
Performance Considerations for Join Queries.....568
How Joins Are Processed when Statistics Are Unavailable.....569
Overriding Join Reordering with STRAIGHT_JOIN.....569
Examples of Join Order Optimization.....570
Table and Column Statistics.....575
Overview of Table Statistics.....575
Overview of Column Statistics.....576
How Table and Column Statistics Work for Partitioned Tables.....577
Generating Table and Column Statistics.....578
Detecting Missing Statistics.....582
Manually Setting Table and Column Statistics with ALTER TABLE.....584
Examples of Using Table and Column Statistics with Impala.....585
Benchmarking Impala Queries.....588
Controlling Impala Resource Usage.....588
Runtime Filtering for Impala Queries (CDH 5.7 or higher only).....588
Background Information for Runtime Filtering.....589
Runtime Filtering Internals.....590
File Format Considerations for Runtime Filtering.....590
Wait Intervals for Runtime Filters.....590
Query Options for Runtime Filtering.....591
Runtime Filtering and Query Plans.....591
Examples of Queries that Benefit from Runtime Filtering.....592
Tuning and Troubleshooting Queries that Use Runtime Filtering.....593
Limitations and Restrictions for Runtime Filtering.....593
Using HDFS Caching with Impala (CDH 5.3 or higher only).....593
Overview of HDFS Caching for Impala.....594
Setting Up HDFS Caching for Impala.....594

Enabling HDFS Caching for Impala Tables and Partitions....595
Loading and Removing Data with HDFS Caching Enabled....596
Administration for HDFS Caching with Impala....597
Performance Considerations for HDFS Caching with Impala....598
Detecting and Correcting HDFS Block Skew Conditions....599
Data Cache for Remote Reads....600
Testing Impala Performance....601
Understanding Impala Query Performance - EXPLAIN Plans and Query Profiles....602
Using the EXPLAIN Plan for Performance Tuning....602
Using the SUMMARY Report for Performance Tuning....603
Using the Query Profile for Performance Tuning....604

Scalability Considerations for Impala....605
Impact of Many Tables or Partitions on Impala Catalog Performance and Memory Usage....605
Scalability Consideration for Large Clusters....605
Scalability Considerations for the Impala Statestore....606
Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher)....607
SQL Operations that Spill to Disk....607
Limits on Query Size and Complexity....611
Scalability Considerations for Impala I/O....611
Scalability Considerations for Table Layout....611
Kerberos-Related Network Overhead for Large Clusters....611
Avoiding CPU Hotspots for HDFS Cached Data....612
Scalability Considerations for File Handle Caching....612
Scaling Limits and Guidelines....613
Deployment Limits....613
Data Storage Limits....614
Schema Design Limits....614
Security Limits....614
Query Limits - Compile Time....614
Query Limits - Runtime Time....614
How to Configure Impala with Dedicated Coordinators....614
Determining the Optimal Number of Dedicated Coordinators....615
Deploying Dedicated Coordinators and Executors in Cloudera Manager....618
Deploying Dedicated Coordinators and Executors from Command Line....619
Metadata Management....620
On-demand Metadata....620
Automatic Invalidation of Metadata Cache....620
Automatic Invalidation/Refresh of Metadata....621

Partitioning for Impala Tables....625

When to Use Partitioned Tables....625
SQL Statements for Partitioned Tables....625
Static and Dynamic Partitioning Clauses....626
Refreshing a Single Partition....626
Permissions for Partition Subdirectories....627
Partition Pruning for Queries....627
Checking if Partition Pruning Happens for a Query....627
What SQL Constructs Work with Partition Pruning....628
Dynamic Partition Pruning....628
Partition Key Columns....630
Setting Different File Formats for Partitions....630
Managing Partitions....631
Using Partitioning with Kudu Tables....631
Keeping Statistics Up to Date for Partitioned Tables....631

How Impala Works with Hadoop File Formats....634

Choosing the File Format for a Table....635
Using Text Data Files with Impala Tables....636
Query Performance for Impala Text Tables....636
Creating Text Tables....637
Data Files for Text Tables....638
Loading Data into Impala Text Tables....639
Using LZO-Compressed Text Files....640
Using gzip, bzip2, or Snappy-Compressed Text Files....642
Using the Parquet File Format with Impala Tables....643
Creating Parquet Tables in Impala....643
Loading Data into Parquet Tables....644
Query Performance for Impala Parquet Tables....645
Compressions for Parquet Data Files....647
Parquet Tables for Impala Complex Types....649
Exchanging Parquet Data Files with Other Hadoop Components....649
How Parquet Data Files Are Organized....652
Compacting Data Files for Parquet Tables....653
Schema Evolution for Parquet Tables....654
Data Type Considerations for Parquet Tables....655
Using the ORC File Format with Impala Tables....656
Creating ORC Tables and Loading Data....657
Enabling Compression for ORC Tables....657
Query Performance for Impala ORC Tables....658
Data Type Considerations for ORC Tables....659
Using the Avro File Format with Impala Tables....659
Creating Avro Tables....660

Using a Hive-Created Avro Table in Impala *662*
Specifying the Avro Schema through JSON *662*
Loading Data into an Avro Table *662*
Enabling Compression for Avro Tables *663*
How Impala Handles Avro Schema Evolution *663*
Data Type Considerations for Avro Tables *664*
Query Performance for Impala Avro Tables *664*
Using the RCFile File Format with Impala Tables 665
Creating RCFile Tables and Loading Data *665*
Enabling Compression for RCFile Tables *666*
Query Performance for Impala RCFile Tables *667*
Using the SequenceFile File Format with Impala Tables 667
Creating SequenceFile Tables and Loading Data *667*
Enabling Compression for SequenceFile Tables *668*
Query Performance for Impala SequenceFile Tables *669*

Using Impala to Query Kudu Tables 670
Benefits of Using Kudu Tables with Impala 670
Configuring Impala for Use with Kudu 670
Cluster Topology for Kudu Tables *670*
Kudu Replication Factor 671
Impala DDL Enhancements for Kudu Tables (CREATE TABLE and ALTER TABLE) 671
Primary Key Columns for Kudu Tables *671*
Kudu-Specific Column Attributes for CREATE TABLE *671*
Partitioning for Kudu Tables *675*
Handling Date, Time, or Timestamp Data with Kudu *678*
How Impala Handles Kudu Metadata *680*
Working with Kudu Integrated with Hive Metastore 681
Loading Data into Kudu Tables 681
Impala DML Support for Kudu Tables (INSERT, UPDATE, DELETE, UPSERT) 681
Consistency Considerations for Kudu Tables 682
Security Considerations for Kudu Tables 682
Impala Query Performance for Kudu Tables 683

Using Impala to Query HBase Tables 684
Overview of Using HBase with Impala 684
Configuring HBase for Use with Impala 684
Supported Data Types for HBase Columns 685
Performance Considerations for the Impala-HBase Integration 685
Use Cases for Querying HBase through Impala 689
Loading Data into an HBase Table 689

Limitations and Restrictions of the Impala and HBase Integration....689
Examples of Querying HBase Tables from Impala....690

Using Impala with the Amazon S3 Filesystem....692

How Impala SQL Statements Work with S3....692
Loading Data into S3 for Impala Queries....693
Using Impala DML Statements for S3 Data....693
Manually Loading Data into Impala Tables on S3....693
Creating Impala Databases, Tables, and Partitions for Data Stored on S3....694
Internal and External Tables Located on S3....695
Running and Tuning Impala Queries for Data Stored on S3....697
Understanding and Tuning Impala Query Performance for S3 Data....697
Restrictions on Impala Support for S3....698
Best Practices for Using Impala with S3....698
Specifying Impala Credentials to Access Data in S3 with Cloudera Manager....699
Specifying Impala Credentials on Clusters Not Secured by Sentry or Kerberos....699
Specifying Impala Credentials to Access Data in S3....699

Using Impala with the Azure Data Lake Store (ADLS)....701

Prerequisites....701
How Impala SQL Statements Work with ADLS....701
Specifying Impala Credentials to Access Data in ADLS....702
Loading Data into ADLS for Impala Queries....703
Using Impala DML Statements for ADLS Data....703
Manually Loading Data into Impala Tables on ADLS....703
Creating Impala Databases, Tables, and Partitions for Data Stored on ADLS....703
Internal and External Tables Located on ADLS....705
Running and Tuning Impala Queries for Data Stored on ADLS....707
Understanding and Tuning Impala Query Performance for ADLS Data....707
Restrictions on Impala Support for ADLS....708
Best Practices for Using Impala with ADLS....708

Using Impala Logging....709

Locations and Names of Impala Log Files....709
Managing Impala Logs through Cloudera Manager or Manually....710
Rotating Impala Logs....710
Reviewing Impala Logs....710
Understanding Impala Log Contents....711
Setting Logging Levels....711
Redacting Sensitive Information from Impala Log Files....712

Impala Client Access.....713
Using the Impala Shell (impala-shell Command).....713
impala-shell Configuration Options.....714
Connecting to impalad through impala-shell.....718
Running Commands and SQL Statements in impala-shell.....720
impala-shell Command Reference.....721
Configuring Impala to Work with ODBC.....723
Downloading the ODBC Driver.....723
Configuring the ODBC Port.....724
Example of Setting Up an ODBC Application for Impala.....724
Notes about JDBC and ODBC Interaction with Impala SQL Features.....725
Configuring Impala to Work with JDBC.....725
Configuring the JDBC Port.....726
Choosing the JDBC Driver.....726
Enabling Impala JDBC Support on Client Systems.....726
Establishing JDBC Connections.....727
Notes about JDBC and ODBC Interaction with Impala SQL Features.....728
Kudu Considerations for DML Statements.....729

Troubleshooting Impala.....730
Troubleshooting Impala SQL Syntax Issues.....730
Troubleshooting Crashes Caused by Memory Resource Limit.....730
Troubleshooting I/O Capacity Problems.....731
Impala Troubleshooting Quick Reference.....731
Impala Web User Interface for Debugging.....733
Debug Web UI for impalad.....733
Debug Web UI for statestored.....736
Debug Web UI for catalogd.....738
Breakpad Minidumps for Impala (CDH 5.8 or higher only).....739
Enabling or Disabling Minidump Generation.....740
Specifying the Location for Minidump Files.....740
Controlling the Number of Minidump Files.....740
Detecting Crash Events.....740
Using the Minidump Files for Problem Resolution.....740
Demonstration of Breakpad Feature.....741

Ports Used by Impala.....743

Impala Reserved Words.....745
List of Current Reserved Words.....745

Impala Frequently Asked Questions.....761
Transition to Apache Governance.....761
Trying Impala.....761
Impala System Requirements.....762
Supported and Unsupported Functionality In Impala.....763
How do I?.....765
Impala Performance.....765
Impala Use Cases.....768
Questions about Impala And Hive.....768
Impala Availability.....769
Impala Internals.....770
SQL.....772
Partitioned Tables.....773
HBase.....773

Appendix: Apache License, Version 2.0.....775

Introducing Apache Impala

Impala provides fast, interactive SQL queries directly on your Apache Hadoop data stored in HDFS, HBase, or the Amazon Simple Storage Service (S3). In addition to using the same unified storage platform, Impala also uses the same metadata, SQL syntax (Hive SQL), ODBC driver, and user interface (Impala query UI in Hue) as Apache Hive. This provides a familiar and unified platform for real-time or batch-oriented queries.

Impala is an addition to tools available for querying big data. Impala does not replace the batch processing frameworks built on MapReduce such as Hive. Hive and other frameworks built on MapReduce are best suited for long running batch jobs, such as those involving batch processing of Extract, Transform, and Load (ETL) type jobs.

Note: Impala graduated from the Apache Incubator on November 15, 2017. In places where the documentation formerly referred to “Cloudera Impala”, now the official name is “Apache Impala”.

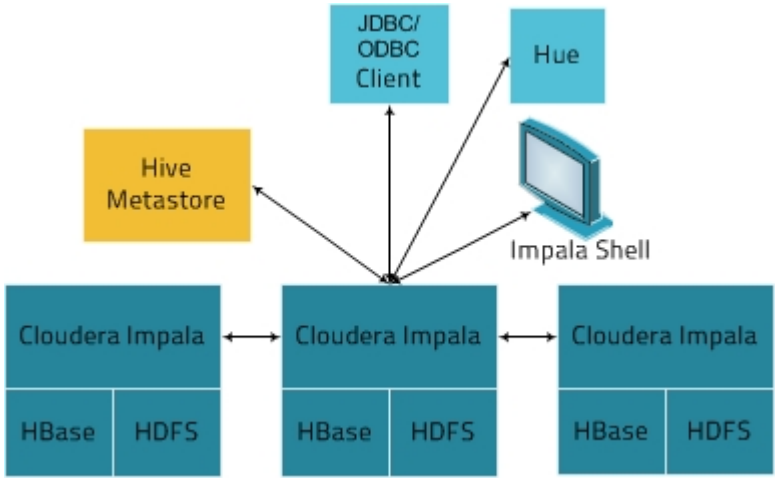
Impala Benefits

Impala provides:

- Familiar SQL interface that data scientists and analysts already know.
- Ability to query high volumes of data (“big data”) in Apache Hadoop.
- Distributed queries in a cluster environment, for convenient scaling and to make use of cost-effective commodity hardware.
- Ability to share data files between different components with no copy or export/import step; for example, to write with Pig, transform with Hive and query with Impala. Impala can read from and write to Hive tables, enabling simple data interchange using Impala for analytics on Hive-produced data.
- Single system for big data processing and analytics, so customers can avoid costly modeling and ETL just for analytics.

How Impala Works with CDH

The following graphic illustrates how Impala is positioned in the broader Cloudera environment:



The Impala solution is composed of the following components:

- Clients - Entities including Hue, ODBC clients, JDBC clients, and the Impala Shell can all interact with Impala. These interfaces are typically used to issue queries or complete administrative tasks such as connecting to Impala.

- Hive Metastore - Stores information about the data available to Impala. For example, the metastore lets Impala know what databases are available and what the structure of those databases is. As you create, drop, and alter schema objects, load data into tables, and so on through Impala SQL statements, the relevant metadata changes are automatically broadcast to all Impala nodes by the dedicated catalog service introduced in Impala 1.2.
- Impala - This process, which runs on DataNodes, coordinates and executes queries. Each instance of Impala can receive, plan, and coordinate queries from Impala clients. Queries are distributed among Impala nodes, and these nodes then act as workers, executing parallel query fragments.
- HBase and HDFS - Storage for data to be queried.

Queries executed using Impala are handled as follows:

1. User applications send SQL queries to Impala through ODBC or JDBC, which provide standardized querying interfaces. The user application may connect to any `impalad` in the cluster. This `impalad` becomes the coordinator for the query.
2. Impala parses the query and analyzes it to determine what tasks need to be performed by `impalad` instances across the cluster. Execution is planned for optimal efficiency.
3. Services such as HDFS and HBase are accessed by local `impalad` instances to provide data.
4. Each `impalad` returns data to the coordinating `impalad`, which sends these results to the client.

Primary Impala Features

Impala provides support for:

- Most common SQL-92 features of Hive Query Language (HiveQL) including SELECT, joins, and aggregate functions.
- HDFS, HBase, and Amazon Simple Storage System (S3) storage, including:
  - HDFS file formats: delimited text files, Parquet, Avro, SequenceFile, and RCFile.
  - Compression codecs: Snappy, GZIP, Deflate, BZIP.
- Common data access interfaces including:
  - JDBC driver.
  - ODBC driver.
  - Hue Beeswax and the Impala Query UI.
- impala-shell command-line interface.
- Kerberos authentication.

Impala Concepts and Architecture

The following sections provide background information to help you become productive using Impala and its features. Where appropriate, the explanations include context to help understand how aspects of Impala relate to other technologies you might already be familiar with, such as relational database management systems and data warehouses, or other Hadoop components such as Hive, HDFS, and HBase.

Components of the Impala Server

The Impala server is a distributed, massively parallel processing (MPP) database engine. It consists of different daemon processes that run on specific hosts within your CDH cluster.

The Impala Daemon

The core Impala component is the Impala daemon, physically represented by the `impalad` process. A few of the key functions that an Impala daemon performs are:

- Reads and writes to data files.
- Accepts queries transmitted from the `impala-shell` command, Hue, JDBC, or ODBC.
- Parallelizes the queries and distributes work across the cluster.
- Transmits intermediate query results back to the central coordinator.

Impala daemons can be deployed in one of the following ways:

- HDFS and Impala are co-located, and each Impala daemon runs on the same host as a DataNode.
- Impala is deployed separately in a compute cluster and reads remotely from HDFS, S3, ADLS, etc.

The Impala daemons are in constant communication with StateStore, to confirm which daemons are healthy and can accept new work.

They also receive broadcast messages from the `catalogd` daemon (introduced in Impala 1.2) whenever any Impala daemon in the cluster creates, alters, or drops any type of object, or when an `INSERT` or `LOAD DATA` statement is processed through Impala. This background communication minimizes the need for `REFRESH` or `INVALIDATE METADATA` statements that were needed to coordinate metadata across Impala daemons prior to Impala 1.2.

In CDH 5.12 / Impala 2.9 and higher, you can control which hosts act as query coordinators and which act as query executors, to improve scalability for highly concurrent workloads on large clusters. See How to Configure Impala with Dedicated Coordinators on page 614 for details.

Related information: Setting the Idle Query and Idle Session Timeouts for impalad on page 70, Modifying Impala Startup Options, Ports Used by Impala on page 743, Using Impala through a Proxy for High Availability on page 72

The Impala Statestore

The Impala component known as the StateStore checks on the health of all Impala daemons in a cluster, and continuously relays its findings to each of those daemons. It is physically represented by a daemon process named `statestored`. You only need such a process on one host in a cluster. If an Impala daemon goes offline due to hardware failure, network error, software issue, or other reason, the StateStore informs all the other Impala daemons so that future queries can avoid making requests to the unreachable Impala daemon.

Because the StateStore's purpose is to help when things go wrong and to broadcast metadata to coordinators, it is not always critical to the normal operation of an Impala cluster. If the StateStore is not running or becomes unreachable, the Impala daemons continue running and distributing work among themselves as usual when working with the data known to Impala. The cluster just becomes less robust if other Impala daemons fail, and metadata becomes less consistent as it changes while the StateStore is offline. When the StateStore comes back online, it re-establishes communication with the Impala daemons and resumes its monitoring and broadcasting functions.

If you issue a DDL statement while the StateStore is down, the queries that access the new object the DDL created will fail.

Most considerations for load balancing and high availability apply to the `impalad` daemon. The `statestored` and `catalogd` daemons do not have special requirements for high availability, because problems with those daemons do not result in data loss. If those daemons become unavailable due to an outage on a particular host, you can stop the Impala service, delete the **Impala StateStore** and **Impala Catalog Server** roles, add the roles on a different host, and restart the Impala service.

Related information:

Scalability Considerations for the Impala Statestore on page 606, Modifying Impala Startup Options, Increasing the Statestore Timeout on page 70, Ports Used by Impala on page 743

The Impala Catalog Service

The Impala component known as the Catalog Service relays the metadata changes from Impala SQL statements to all the Impala daemons in a cluster. It is physically represented by a daemon process named `catalogd`. You only need such a process on one host in a cluster. Because the requests are passed through the StateStore daemon, it makes sense to run the `statestored` and `catalogd` services on the same host.

The catalog service avoids the need to issue `REFRESH` and `INVALIDATE METADATA` statements when the metadata changes are performed by statements issued through Impala. When you create a table, load data, and so on through Hive, you do need to issue `REFRESH` or `INVALIDATE METADATA` on an Impala node before executing a query there.

This feature touches a number of aspects of Impala:

- See Impala Upgrade Considerations on page 38 and Modifying Impala Startup Options for usage information for the `catalogd` daemon.
- The `REFRESH` and `INVALIDATE METADATA` statements are not needed when the `CREATE TABLE`, `INSERT`, or other table-changing or data-changing operation is performed through Impala. These statements are still needed if such operations are done through Hive or by manipulating data files directly in HDFS, but in those cases the statements only need to be issued on one Impala daemon rather than on all daemons. See REFRESH Statement on page 291 and INVALIDATE METADATA Statement on page 286 for the latest usage information for those statements.

Use `--load_catalog_in_background` option to control when the metadata of a table is loaded.

- If set to `false`, the metadata of a table is loaded when it is referenced for the first time. This means that the first run of a particular query can be slower than subsequent runs. Starting in Impala 2.2, the default for `load_catalog_in_background` is `false`.
- If set to `true`, the catalog service attempts to load metadata for a table even if no query needed that metadata. So metadata will possibly be already loaded when the first query that would need it is run. However, for the following reasons, we recommend not to set the option to `true`.
  - Background load can interfere with query-specific metadata loading. This can happen on startup or after invalidating metadata, with a duration depending on the amount of metadata, and can lead to a seemingly random long running queries that are difficult to diagnose.
  - Impala may load metadata for tables that are possibly never used, potentially increasing catalog size and consequently memory usage for both catalog service and Impala Daemon.

Most considerations for load balancing and high availability apply to the `impalad` daemon. The `statestored` and `catalogd` daemons do not have special requirements for high availability, because problems with those daemons do not result in data loss. If those daemons become unavailable due to an outage on a particular host, you can stop the Impala service, delete the **Impala StateStore** and **Impala Catalog Server** roles, add the roles on a different host, and restart the Impala service.



Note:

In Impala 1.2.4 and higher, you can specify a table name with `INVALIDATE METADATA` after the table is created in Hive, allowing you to make individual tables visible to Impala without doing a full reload of the catalog metadata. Impala 1.2.4 also includes other changes to make the metadata broadcast mechanism faster and more responsive, especially during Impala startup.

Related information:

Modifying Impala Startup Options, Ports Used by Impala on page 743

Developing Impala Applications

The core development language with Impala is SQL. You can also use Java or other languages to interact with Impala through the standard JDBC and ODBC interfaces used by many business intelligence tools. For specialized kinds of analysis, you can supplement the SQL built-in functions by writing user-defined functions (UDFs) in C++ or Java.

Overview of the Impala SQL Dialect

The Impala SQL dialect is highly compatible with the SQL syntax used in the Apache Hive component (HiveQL). As such, it is familiar to users who are already familiar with running SQL queries on the Hadoop infrastructure. Currently, Impala SQL supports a subset of HiveQL statements, data types, and built-in functions. Impala also includes additional built-in functions for common industry features, to simplify porting SQL from non-Hadoop systems.

For users coming to Impala from traditional database or data warehousing backgrounds, the following aspects of the SQL dialect might seem familiar:

- The SELECT statement includes familiar clauses such as `WHERE`, `GROUP BY`, `ORDER BY`, and `WITH`. You will find familiar notions such as joins, built-in functions for processing strings, numbers, and dates, aggregate functions, subqueries, and comparison operators such as `IN()` and `BETWEEN`. The `SELECT` statement is the place where SQL standards compliance is most important.
- From the data warehousing world, you will recognize the notion of partitioned tables. One or more columns serve as partition keys, and the data is physically arranged so that queries that refer to the partition key columns in the `WHERE` clause can skip partitions that do not match the filter conditions. For example, if you have 10 years worth of data and use a clause such as `WHERE year = 2015`, `WHERE year > 2010`, or `WHERE year IN (2014, 2015)`, Impala skips all the data for non-matching years, greatly reducing the amount of I/O for the query.
- In Impala 1.2 and higher, UDFs let you perform custom comparisons and transformation logic during `SELECT` and `INSERT...SELECT` statements.

For users coming to Impala from traditional database or data warehousing backgrounds, the following aspects of the SQL dialect might require some learning and practice for you to become proficient in the Hadoop environment:

- Impala SQL is focused on queries and includes relatively little DML. There is no `UPDATE` or `DELETE` statement. Stale data is typically discarded (by `DROP TABLE` or `ALTER TABLE ... DROP PARTITION` statements) or replaced (by `INSERT OVERWRITE` statements).
- All data creation is done by `INSERT` statements, which typically insert data in bulk by querying from other tables. There are two variations, `INSERT INTO` which appends to the existing data, and `INSERT OVERWRITE` which replaces the entire contents of a table or partition (similar to `TRUNCATE TABLE` followed by a new `INSERT`). Although there is an `INSERT ... VALUES` syntax to create a small number of values in a single statement, it is far more efficient to use the `INSERT ... SELECT` to copy and transform large amounts of data from one table to another in a single operation.
- You often construct Impala table definitions and data files in some other environment, and then attach Impala so that it can run real-time queries. The same data files and table metadata are shared with other components of the Hadoop ecosystem. In particular, Impala can access tables created by Hive or data inserted by Hive, and Hive

can access tables and data produced by Impala. Many other Hadoop components can write files in formats such as Parquet and Avro, that can then be queried by Impala.

- Because Hadoop and Impala are focused on data warehouse-style operations on large data sets, Impala SQL includes some idioms that you might find in the import utilities for traditional database systems. For example, you can create a table that reads comma-separated or tab-separated text files, specifying the separator in the CREATE TABLE statement. You can create **external tables** that read existing data files but do not move or transform them.
- Because Impala reads large quantities of data that might not be perfectly tidy and predictable, it does not require length constraints on string data types. For example, you can define a database column as STRING with unlimited length, rather than CHAR(1) or VARCHAR(64). (Although in Impala 2.0 and later, you can also use length-constrained CHAR and VARCHAR types.)

Related information: Impala SQL Language Reference on page 101, especially Impala SQL Statements on page 202 and Impala Built-In Functions on page 391

Overview of Impala Programming Interfaces

You can connect and submit requests to the Impala daemons through:

- The impala-shell interactive command interpreter.
- The Hue web-based user interface.
- JDBC.
- ODBC.

With these options, you can use Impala in heterogeneous environments, with JDBC or ODBC applications running on non-Linux platforms. You can also use Impala on combination with various Business Intelligence tools that use the JDBC and ODBC interfaces.

Each impalad daemon process, running on separate nodes in a cluster, listens to several ports for incoming requests. Requests from impala-shell and Hue are routed to the impalad daemons through the same port. The impalad daemons listen on separate ports for JDBC and ODBC requests.

How Impala Fits Into the Hadoop Ecosystem

Impala makes use of many familiar components within the Hadoop ecosystem. Impala can interchange data with other Hadoop components, as both a consumer and a producer, so it can fit in flexible ways into your ETL and ELT pipelines.

How Impala Works with Hive

A major Impala goal is to make SQL-on-Hadoop operations fast and efficient enough to appeal to new categories of users and open up Hadoop to new types of use cases. Where practical, it makes use of existing Apache Hive infrastructure that many Hadoop users already have in place to perform long-running, batch-oriented SQL queries.

In particular, Impala keeps its table definitions in a traditional MySQL or PostgreSQL database known as the **metastore**, the same database where Hive keeps this type of data. Thus, Impala can access tables defined or loaded by Hive, as long as all columns use Impala-supported data types, file formats, and compression codecs.

The initial focus on query features and performance means that Impala can read more types of data with the SELECT statement than it can write with the INSERT statement. To query data using the Avro, RCFile, or SequenceFile file formats, you load the data using Hive.

The Impala query optimizer can also make use of table statistics and column statistics. Originally, you gathered this information with the ANALYZE TABLE statement in Hive; in Impala 1.2.2 and higher, use the Impala COMPUTE STATS statement instead. COMPUTE STATS requires less setup, is more reliable, and does not require switching back and forth between impala-shell and the Hive shell.

Overview of Impala Metadata and the Metastore

As discussed in How Impala Works with Hive on page 21, Impala maintains information about table definitions in a central database known as the **metastore**. Impala also tracks other metadata for the low-level characteristics of data files:

- The physical locations of blocks within HDFS.

For tables with a large volume of data and/or many partitions, retrieving all the metadata for a table can be time-consuming, taking minutes in some cases. Thus, each Impala node caches all of this metadata to reuse for future queries against the same table.

If the table definition or the data in the table is updated, all other Impala daemons in the cluster must receive the latest metadata, replacing the obsolete cached metadata, before issuing a query against that table. In Impala 1.2 and higher, the metadata update is automatic, coordinated through the `catalogd` daemon, for all DDL and DML statements issued through Impala. See The Impala Catalog Service on page 19 for details.

For DDL and DML issued through Hive, or changes made manually to files in HDFS, you still use the `REFRESH` statement (when new data files are added to existing tables) or the `INVALIDATE METADATA` statement (for entirely new tables, or after dropping a table, performing an HDFS rebalance operation, or deleting data files). Issuing `INVALIDATE METADATA` by itself retrieves metadata for all the tables tracked by the metastore. If you know that only specific tables have been changed outside of Impala, you can issue `REFRESH` *`table_name`* for each affected table to only retrieve the latest metadata for those tables.

How Impala Uses HDFS

Impala uses the distributed filesystem HDFS as its primary data storage medium. Impala relies on the redundancy provided by HDFS to guard against hardware or network outages on individual nodes. Impala table data is physically represented as data files in HDFS, using familiar HDFS file formats and compression codecs. When data files are present in the directory for a new table, Impala reads them all, regardless of file name. New data is added in files with names controlled by Impala.

How Impala Uses HBase

HBase is an alternative to HDFS as a storage medium for Impala data. It is a database storage system built on top of HDFS, without built-in SQL support. Many Hadoop users already have it configured and store large (often sparse) data sets in it. By defining tables in Impala and mapping them to equivalent tables in HBase, you can query the contents of the HBase tables through Impala, and even perform join queries including both Impala and HBase tables. See Using Impala to Query HBase Tables on page 684 for details.

Planning for Impala Deployment

Before you set up Impala in production, do some planning to make sure that your hardware setup has sufficient capacity, that your cluster topology is optimal for Impala queries, and that your schema design and ETL processes follow the best practices for Impala.

Impala Requirements

To perform as expected, Impala depends on the availability of the software, hardware, and configurations described in the following sections.

Product Compatibility Matrix

The ultimate source of truth about compatibility between various versions of CDH, Cloudera Manager, and various CDH components is the .

Supported Operating Systems

The relevant supported operating systems and versions for Impala are the same as for the corresponding CDH platforms. For details, see the *Supported Operating Systems* page for CDH.

Hive Metastore and Related Configuration

Impala can interoperate with data stored in Hive, and uses the same infrastructure as Hive for tracking metadata about schema objects such as tables and columns. The following components are prerequisites for Impala:

- MySQL or PostgreSQL, to act as a metastore database for both Impala and Hive.

  Always configure a **Hive metastore service** rather than connecting directly to the metastore database. The Hive metastore service is required to interoperate between different levels of metastore APIs if this is necessary for your environment, and using it avoids known issues with connecting directly to the metastore database.

  See below for a summary of the metastore installation process.

- Hive (optional). Although only the Hive metastore database is required for Impala to function, you might install Hive on some client machines to create and load data into tables that use certain file formats. See How Impala Works with Hadoop File Formats on page 634 for details. Hive does not need to be installed on the same DataNodes as Impala; it just needs access to the same metastore database.

To install the metastore:

1. Install a MySQL or PostgreSQL database. Start the database if it is not started after installation.
2. Download the MySQL connector or the PostgreSQL connector and place it in the `/usr/share/java/` directory.
3. Use the appropriate command line tool for your database to create the metastore database.
4. Use the appropriate command line tool for your database to grant privileges for the metastore database to the `hive` user.
5. Modify `hive-site.xml` to include information matching your particular database: its URL, username, and password. You will copy the `hive-site.xml` file to the Impala Configuration Directory later in the Impala installation process.

Java Dependencies

Although Impala is primarily written in C++, it does use Java to communicate with various Hadoop components:

- The officially supported JVMs for Impala are the OpenJDK JVM and Oracle JVM. Other JVMs might cause issues, typically resulting in a failure at `impalad` startup. In particular, the JamVM used by default on certain levels of Ubuntu systems can cause `impalad` to fail to start.

- Internally, the `impalad` daemon relies on the `JAVA_HOME` environment variable to locate the system Java libraries. Make sure the `impalad` service is not run from an environment with an incorrect setting for this variable.
- All Java dependencies are packaged in the `impala-dependencies.jar` file, which is located at `/usr/lib/impala/lib/`. These map to everything that is built under `fe/target/dependency`.

Networking Configuration Requirements

As part of ensuring best performance, Impala attempts to complete tasks on local data, as opposed to using network connections to work with remote data. To support this goal, Impala matches the **hostname** provided to each Impala daemon with the **IP address** of each DataNode by resolving the hostname flag to an IP address. For Impala to work with local data, use a single IP interface for the DataNode and the Impala daemon on each machine. Ensure that the Impala daemon's hostname flag resolves to the IP address of the DataNode. For single-homed machines, this is usually automatic, but for multi-homed machines, ensure that the Impala daemon's hostname resolves to the correct interface. Impala tries to detect the correct hostname at start-up, and prints the derived hostname at the start of the log in a message of the form:

```
Using hostname: impala-daemon-1.example.com
```

In the majority of cases, this automatic detection works correctly. If you need to explicitly set the hostname, do so by setting the `--hostname` flag.

Hardware Requirements

The memory allocation should be consistent across Impala executor nodes. A single Impala executor with a lower memory limit than the rest can easily become a bottleneck and lead to suboptimal performance.

This guideline does not apply to coordinator-only nodes.

Hardware Requirements for Optimal Join Performance

During join operations, portions of data from each joined table are loaded into memory. Data sets can be very large, so ensure your hardware has sufficient memory to accommodate the joins you anticipate completing.

While requirements vary according to data set size, the following is generally recommended:

- CPU

  Impala version 2.2 and higher uses the SSSE3 instruction set, which is included in newer processors.

  Note: This required level of processor is the same as in Impala version 1.x. The Impala 2.0 and 2.1 releases had a stricter requirement for the SSE4.1 instruction set, which has now been relaxed.

- Memory

  128 GB or more recommended, ideally 256 GB or more. If the intermediate results during query processing on a particular node exceed the amount of memory available to Impala on that node, the query writes temporary work data to disk, which can lead to long query times. Note that because the work is parallelized, and intermediate results for aggregate queries are typically smaller than the original data, Impala can query and join tables that are much larger than the memory available on an individual node.

- JVM Heap Size for Catalog Server

  4 GB or more recommended, ideally 8 GB or more, to accommodate the maximum numbers of tables, partitions, and data files you are planning to use with Impala.

- Storage

  DataNodes with 12 or more disks each. I/O speeds are often the limiting factor for disk performance with Impala. Ensure that you have sufficient disk space to store the data Impala will be querying.

User Account Requirements

Impala creates and uses a user and group named `impala`. Do not delete this account or group and do not modify the account's or group's permissions and rights. Ensure no existing systems obstruct the functioning of these accounts and groups. For example, if you have scripts that delete user accounts not in a white-list, add these accounts to the list of permitted accounts.

For correct file deletion during `DROP TABLE` operations, Impala must be able to move files to the HDFS trashcan. You might need to create an HDFS directory `/user/impala`, writeable by the `impala` user, so that the trashcan can be created. Otherwise, data files might remain behind after a `DROP TABLE` statement.

Impala should not run as root. Best Impala performance is achieved using direct reads, but root is not permitted to use direct reads. Therefore, running Impala as root negatively affects performance.

By default, any user can connect to Impala and access all the associated databases and tables. You can enable authorization and authentication based on the Linux OS user who connects to the Impala server, and the associated groups for that user. Impala Security on page 82 for details. These security features do not change the underlying file permission requirements; the `impala` user still needs to be able to access the data files.

Guidelines for Designing Impala Schemas

The guidelines in this topic help you to construct an optimized and scalable schema, one that integrates well with your existing data management processes. Use these guidelines as a checklist when doing any proof-of-concept work, porting exercise, or before deploying to production.

If you are adapting an existing database or Hive schema for use with Impala, read the guidelines in this section and then see Porting SQL from Other Database Systems to Impala on page 543 for specific porting and compatibility tips.

Prefer binary file formats over text-based formats.

To save space and improve memory usage and query performance, use binary file formats for any large or intensively queried tables. Parquet file format is the most efficient for data warehouse-style analytic queries. Avro is the other binary file format that Impala supports, that you might already have as part of a Hadoop ETL pipeline.

Although Impala can create and query tables with the RCFile and SequenceFile file formats, such tables are relatively bulky due to the text-based nature of those formats, and are not optimized for data warehouse-style queries due to their row-oriented layout. Impala does not support `INSERT` operations for tables with these file formats.

Guidelines:

- For an efficient and scalable format for large, performance-critical tables, use the Parquet file format.
- To deliver intermediate data during the ETL process, in a format that can also be used by other Hadoop components, Avro is a reasonable choice.
- For convenient import of raw data, use a text table instead of RCFile or SequenceFile, and convert to Parquet in a later stage of the ETL process.

Use Snappy compression where practical.

Snappy compression involves low CPU overhead to decompress, while still providing substantial space savings. In cases where you have a choice of compression codecs, such as with the Parquet and Avro file formats, use Snappy compression unless you find a compelling reason to use a different codec.

Prefer numeric types over strings.

If you have numeric values that you could treat as either strings or numbers (such as YEAR, `MONTH`, and DAY for partition key columns), define them as the smallest applicable integer types. For example, YEAR can be `SMALLINT`, `MONTH` and DAY can be `TINYINT`. Although you might not see any difference in the way partitioned tables or text files are laid out on disk, using numeric types will save space in binary formats such as Parquet, and in memory when doing queries, particularly resource-intensive queries such as joins.

Partition, but do not over-partition.

Partitioning is an important aspect of performance tuning for Impala. Follow the procedures in Partitioning for Impala Tables on page 625 to set up partitioning for your biggest, most intensively queried tables.

If you are moving to Impala from a traditional database system, or just getting started in the Big Data field, you might not have enough data volume to take advantage of Impala parallel queries with your existing partitioning scheme. For example, if you have only a few tens of megabytes of data per day, partitioning by `YEAR`, `MONTH`, and `DAY` columns might be too granular. Most of your cluster might be sitting idle during queries that target a single day, or each node might have very little work to do. Consider reducing the number of partition key columns so that each partition directory contains several gigabytes worth of data.

For example, consider a Parquet table where each data file is 1 HDFS block, with a maximum block size of 1 GB. (In Impala 2.0 and later, the default Parquet block size is reduced to 256 MB. For this exercise, let's assume you have bumped the size back up to 1 GB by setting the query option `PARQUET_FILE_SIZE=1g`.) if you have a 10-node cluster, you need 10 data files (up to 10 GB) to give each node some work to do for a query. But each core on each machine can process a separate data block in parallel. With 16-core machines on a 10-node cluster, a query could process up to 160 GB fully in parallel. If there are only a few data files per partition, not only are most cluster nodes sitting idle during queries, so are most cores on those machines.

You can reduce the Parquet block size to as low as 128 MB or 64 MB to increase the number of files per partition and improve parallelism. But also consider reducing the level of partitioning so that analytic queries have enough data to work with.

Always compute stats after loading data.

Impala makes extensive use of statistics about data in the overall table and in each column, to help plan resource-intensive operations such as join queries and inserting into partitioned Parquet tables. Because this information is only available after data is loaded, run the `COMPUTE STATS` statement on a table after loading or replacing data in a table or partition.

Having accurate statistics can make the difference between a successful operation, or one that fails due to an out-of-memory error or a timeout. When you encounter performance or capacity issues, always use the `SHOW STATS` statement to check if the statistics are present and up-to-date for all tables in the query.

When doing a join query, Impala consults the statistics for each joined table to determine their relative sizes and to estimate the number of rows produced in each join stage. When doing an `INSERT` into a Parquet table, Impala consults the statistics for the source table to determine how to distribute the work of constructing the data files for each partition.

See COMPUTE STATS Statement on page 219 for the syntax of the `COMPUTE STATS` statement, and Table and Column Statistics on page 575 for all the performance considerations for table and column statistics.

Verify sensible execution plans with EXPLAIN and SUMMARY.

Before executing a resource-intensive query, use the `EXPLAIN` statement to get an overview of how Impala intends to parallelize the query and distribute the work. If you see that the query plan is inefficient, you can take tuning steps such as changing file formats, using partitioned tables, running the `COMPUTE STATS` statement, or adding query hints. For information about all of these techniques, see Tuning Impala for Performance on page 565.

After you run a query, you can see performance-related information about how it actually ran by issuing the `SUMMARY` command in `impala-shell`. Prior to Impala 1.4, you would use the `PROFILE` command, but its highly technical output was only useful for the most experienced users. `SUMMARY`, new in Impala 1.4, summarizes the most useful information for all stages of execution, for all nodes rather than splitting out figures for each node.

Setting Up Apache Impala Using the Command Line

Impala is an open-source add-on to the Cloudera Enterprise Core that returns rapid responses to queries.



Note:

Under CDH 5 and CDH 6, Impala is included as part of the CDH installation and no separate steps are needed. Therefore, the instruction steps in this section apply to CDH 4 only.

What is Included in an Impala Installation

Impala is made up of a set of components that can be installed on multiple nodes throughout your cluster. The key installation step for performance is to install the `impalad` daemon (which does most of the query processing work) on *all* DataNodes in the cluster.

The Impala package installs these binaries:

- `impalad` - The Impala daemon. Plans and executes queries against HDFS, HBase, and Amazon S3 data. Run one impalad process on each node in the cluster that has a DataNode.
- `statestored` - Name service that tracks location and status of all `impalad` instances in the cluster. Run one instance of this daemon on a node in your cluster. Most production deployments run this daemon on the namenode.
- `catalogd` - Metadata coordination service that broadcasts changes from Impala DDL and DML statements to all affected Impala nodes, so that new tables, newly loaded data, and so on are immediately visible to queries submitted through any Impala node. (Prior to Impala 1.2, you had to run the `REFRESH` or `INVALIDATE METADATA` statement on each node to synchronize changed metadata. Now those statements are only required if you perform the DDL or DML through an external mechanism such as Hive or by uploading data to the Amazon S3 filesystem.) Run one instance of this daemon on a node in your cluster, preferably on the same host as the `statestored` daemon.
- `impala-shell` - Command-line interface for issuing queries to the Impala daemon. You install this on one or more hosts anywhere on your network, not necessarily DataNodes or even within the same cluster as Impala. It can connect remotely to any instance of the Impala daemon.

Before doing the installation, ensure that you have all necessary prerequisites. See Impala Requirements on page 23 for details.

Installing Impala from the Command Line

Before installing Impala manually, make sure all applicable nodes have the appropriate hardware configuration, levels of operating system and CDH, and any other software prerequisites. See Impala Requirements on page 23 for details.

You can install Impala across many hosts or on one host:

- Installing Impala across multiple machines creates a distributed configuration. For best performance, install Impala on **all** DataNodes.
- Installing Impala on a single machine produces a pseudo-distributed cluster.

To install Impala on a host:

1. Install CDH, including Hive, as described in Installing and Deploying Unmanaged CDH Using the Command Line.
2. Configure the Hive metastore to use an external database as a metastore. Impala uses this same database for its own table metadata. You can choose either a MySQL or PostgreSQL database as the metastore. The process for configuring each type of database is described in the CDH Installation Guide).

Cloudera recommends setting up a Hive metastore service rather than connecting directly to the metastore database; this configuration is required when running Impala under CDH 4.1. Make sure the `/etc/impala/conf/hive-site.xml` file contains the following setting, substituting the appropriate hostname for *metastore_server_host*:

```
<property>
<name>hive.metastore.uris</name>
<value>thrift://metastore_server_host:9083</value>
</property>
<property>
<name>hive.metastore.client.socket.timeout</name>
<value>3600</value>
<description>MetaStore Client socket timeout in seconds</description>
</property>
```

3. (Optional) If you installed the full Hive component on any host, you can verify that the metastore is configured properly by starting the Hive console and querying for the list of available tables. Once you confirm that the console starts, exit the console to continue the installation:

```
$ hive
Hive history file=/tmp/root/hive_job_log_root_201207272011_678722950.txt
hive> show tables;
table1
table2
hive> quit;
$
```

4. Confirm that your package management command is aware of the Impala repository settings, as described in Impala Requirements on page 23. (For CDH 4, this is a different repository than for CDH.) You might need to download a repo or list file into a system directory underneath `/etc`.
5. Use **one** of the following sets of commands to install the Impala package:

   For RHEL, Oracle Linux, or CentOS systems:

```
$ sudo yum install impala             # Binaries for daemons
$ sudo yum install impala-server      # Service start/stop script
$ sudo yum install impala-state-store # Service start/stop script
$ sudo yum install impala-catalog     # Service start/stop script
```

   For SUSE systems:

```
$ sudo zypper install impala             # Binaries for daemons
$ sudo zypper install impala-server      # Service start/stop script
$ sudo zypper install impala-state-store # Service start/stop script
$ sudo zypper install impala-catalog     # Service start/stop script
```

   For Debian or Ubuntu systems:

```
$ sudo apt-get install impala             # Binaries for daemons
$ sudo apt-get install impala-server      # Service start/stop script
$ sudo apt-get install impala-state-store # Service start/stop script
$ sudo apt-get install impala-catalog     # Service start/stop script
```



Note: Cloudera recommends that you not install Impala on any HDFS NameNode. Installing Impala on NameNodes provides no additional data locality, and executing queries with such a configuration might cause memory contention and negatively impact the HDFS NameNode.

6. Copy the client `hive-site.xml`, `core-site.xml`, `hdfs-site.xml`, and `hbase-site.xml` configuration files to the Impala configuration directory, which defaults to `/etc/impala/conf`. Create this directory if it does not already exist.

7. Use **one** of the following commands to install `impala-shell` on the machines from which you want to issue queries. You can install `impala-shell` on any supported machine that can connect to DataNodes that are running `impalad`.

   For RHEL/CentOS systems:

```
$ sudo yum install impala-shell
```

   For SUSE systems:

```
$ sudo zypper install impala-shell
```

   For Debian/Ubuntu systems:

```
$ sudo apt-get install impala-shell
```

8. Complete any required or recommended configuration, as described in Post-Installation Configuration for Impala on page 36. Some of these configuration changes are mandatory.

Once installation and configuration are complete, see Starting Impala on page 32 for how to activate the software on the appropriate nodes in your cluster.

If this is your first time setting up and using Impala in this cluster, run through some of the exercises in Impala Tutorials on page 43 to verify that you can do basic operations such as creating tables and querying them.

Modifying Impala Startup Options

The configuration options for the Impala-related daemons let you choose which hosts and ports to use for the services that run on a single host, specify directories for logging, control resource usage and security, and specify other aspects of the Impala software.

Configuring Impala Startup Options through the Command Line

When you run Impala in a non-Cloudera Manager environment, the Impala server, statestore, and catalog services start up using values provided in a defaults file, `/etc/default/impala`.

This file includes information about many resources used by Impala. Most of the defaults included in this file should be effective in most cases. For example, typically you would not change the definition of the CLASSPATH variable, but you would always set the address used by the statestore server. Some of the content you might modify includes:

```
IMPALA_STATE_STORE_HOST=127.0.0.1
IMPALA_STATE_STORE_PORT=24000
IMPALA_BACKEND_PORT=22000
IMPALA_LOG_DIR=/var/log/impala
IMPALA_CATALOG_SERVICE_HOST=...
IMPALA_STATE_STORE_HOST=...

export IMPALA_STATE_STORE_ARGS=${IMPALA_STATE_STORE_ARGS:- \
    -log_dir=${IMPALA_LOG_DIR} -state_store_port=${IMPALA_STATE_STORE_PORT}}
IMPALA_SERVER_ARGS=" \
-log_dir=${IMPALA_LOG_DIR} \
-catalog_service_host=${IMPALA_CATALOG_SERVICE_HOST} \
-state_store_port=${IMPALA_STATE_STORE_PORT} \
-state_store_host=${IMPALA_STATE_STORE_HOST} \
-be_port=${IMPALA_BACKEND_PORT}"
export ENABLE_CORE_DUMPS=${ENABLE_COREDUMPS:-false}
```

To use alternate values, edit the defaults file, then restart all the Impala-related services so that the changes take effect. Restart the Impala server using the following commands:

```
$ sudo service impala-server restart
Stopping Impala Server:                                    [  OK  ]
Starting Impala Server:                                    [  OK  ]
```

Restart the Impala statestore using the following commands:

```
$ sudo service impala-state-store restart
Stopping Impala State Store Server:                        [  OK  ]
Starting Impala State Store Server:                        [  OK  ]
```

Restart the Impala catalog service using the following commands:

```
$ sudo service impala-catalog restart
Stopping Impala Catalog Server:                            [  OK  ]
Starting Impala Catalog Server:                            [  OK  ]
```

Some common settings to change include:

- Statestore address. Where practical, put the statestore on a separate host not running the `impalad` daemon. In that recommended configuration, the `impalad` daemon cannot refer to the statestore server using the loopback address. If the statestore is hosted on a machine with an IP address of 192.168.0.27, change:

```
IMPALA_STATE_STORE_HOST=127.0.0.1
```

  to:

```
IMPALA_STATE_STORE_HOST=192.168.0.27
```

- Catalog server address (including both the hostname and the port number). Update the value of the `IMPALA_CATALOG_SERVICE_HOST` variable. Cloudera recommends the catalog server be on the same host as the statestore. In that recommended configuration, the `impalad` daemon cannot refer to the catalog server using the loopback address. If the catalog service is hosted on a machine with an IP address of 192.168.0.27, add the following line:

```
IMPALA_CATALOG_SERVICE_HOST=192.168.0.27:26000
```

  The `/etc/default/impala` defaults file currently does not define an `IMPALA_CATALOG_ARGS` environment variable, but if you add one it will be recognized by the service startup/shutdown script. Add a definition for this variable to `/etc/default/impala` and add the option `-catalog_service_host=`*`hostname`*. If the port is different than the default 26000, also add the option `-catalog_service_port=`*`port`*.

- Memory limits. You can limit the amount of memory available to Impala. For example, to allow Impala to use no more than 70% of system memory, change:

```
export IMPALA_SERVER_ARGS=${IMPALA_SERVER_ARGS:- \
    -log_dir=${IMPALA_LOG_DIR} \
    -state_store_port=${IMPALA_STATE_STORE_PORT} \
    -state_store_host=${IMPALA_STATE_STORE_HOST} \
    -be_port=${IMPALA_BACKEND_PORT}}
```

  to:

```
export IMPALA_SERVER_ARGS=${IMPALA_SERVER_ARGS:- \
    -log_dir=${IMPALA_LOG_DIR} -state_store_port=${IMPALA_STATE_STORE_PORT} \
    -state_store_host=${IMPALA_STATE_STORE_HOST} \
    -be_port=${IMPALA_BACKEND_PORT} -mem_limit=70%}
```

You can specify the memory limit using absolute notation such as `500m` or `2G`, or as a percentage of physical memory such as `60%`.

> **Note:** Queries that exceed the specified memory limit are aborted. Percentage limits are based on the physical memory of the machine and do not consider cgroups.

- Core dump enablement. To enable core dumps on systems not managed by Cloudera Manager, change:

```
export ENABLE_CORE_DUMPS=${ENABLE_COREDUMPS:-false}
```

to:

```
export ENABLE_CORE_DUMPS=${ENABLE_COREDUMPS:-true}
```

On systems managed by Cloudera Manager, enable the **Enable Core Dump** setting for the Impala service.

> **Note:**
>
> - The location of core dump files may vary according to your operating system configuration.
> - Other security settings may prevent Impala from writing core dumps even when this option is enabled.
> - The default location for core dumps is on a temporary filesystem, which can lead to out-of-space issues if the core dumps are large, frequent, or not removed promptly. To specify an alternative location for the core dumps, filter the Impala configuration settings to find the `core_dump_dir` option. This option lets you specify a different directory for core dumps for each of the Impala-related daemons.

- Authorization using the open source Sentry plugin. Specify the `-server_name` and `-authorization_policy_file` options as part of the `IMPALA_SERVER_ARGS` and `IMPALA_STATE_STORE_ARGS` settings to enable the core Impala support for authentication. See Enabling Sentry Authorization for Impala on page 87 for details.
- Auditing for successful or blocked Impala queries, another aspect of security. Specify the `-audit_event_log_dir=`*`directory_path`* option and optionally the `-max_audit_event_log_file_size=`*`number_of_queries`* and `-abort_on_failed_audit_event` options as part of the `IMPALA_SERVER_ARGS` settings, for each Impala node, to enable and customize auditing. See Auditing Impala Operations on page 78 for details.
- Password protection for the Impala web UI, which listens on port 25000 by default. This feature involves adding some or all of the `--webserver_password_file`, `--webserver_authentication_domain`, and `--webserver_certificate_file` options to the `IMPALA_SERVER_ARGS` and `IMPALA_STATE_STORE_ARGS` settings. See Security Guidelines for Impala on page 82 for details.
- Another setting you might add to `IMPALA_SERVER_ARGS` is a comma-separated list of query options and values:

```
-default_query_options='option=value,option=value,...'
```

These options control the behavior of queries performed by this `impalad` instance. The option values you specify here override the default values for Impala query options, as shown by the `SET` statement in `impala-shell`.

- During troubleshooting, Cloudera Support might direct you to change other values, particularly for `IMPALA_SERVER_ARGS`, to work around issues or gather debugging information.



Note:

These startup options for the `impalad` daemon are different from the command-line options for the `impala-shell` command. For the `impala-shell` options, see impala-shell Configuration Options on page 714.

Checking the Values of Impala Configuration Options

You can check the current runtime value of all these settings through the Impala web interface, available by default at `http://impala_hostname:25000/varz` for the `impalad` daemon, `http://impala_hostname:25010/varz` for the `statestored` daemon, or `http://impala_hostname:25020/varz` for the `catalogd` daemon. In the Cloudera Manager interface, you can see the link to the appropriate ***service_name* Web UI** page when you look at the status page for a specific daemon on a specific host.

Startup Options for impalad Daemon

The `impalad` daemon implements the main Impala service, which performs query processing and reads and writes the data files. Some of the noteworthy options are:

- The `fe_service_threads` option specifies the maximum number of concurrent client connections allowed. The default value is 64 with which 64 queries can run simultaneously.

  If you have more clients trying to connect to Impala than the value of this setting, the later arriving clients have to wait until previous clients disconnect. You can increase this value to allow more client connections. However, a large value means more threads to be maintained even if most of the connections are idle, and it could negatively impact query latency. Client applications should use the connection pool to avoid the need for large number of sessions.

Startup Options for statestored Daemon

The `statestored` daemon implements the Impala statestore service, which monitors the availability of Impala services across the cluster, and handles situations such as nodes becoming unavailable or becoming available again.

Startup Options for catalogd Daemon

The `catalogd` daemon implements the Impala catalog service, which broadcasts metadata changes to all the Impala nodes when Impala creates a table, inserts data, or performs other kinds of DDL and DML operations.

Use `--load_catalog_in_background` option to control when the metadata of a table is loaded.

- If set to `false`, the metadata of a table is loaded when it is referenced for the first time. This means that the first run of a particular query can be slower than subsequent runs. Starting in Impala 2.2, the default for `load_catalog_in_background` is `false`.
- If set to `true`, the catalog service attempts to load metadata for a table even if no query needed that metadata. So metadata will possibly be already loaded when the first query that would need it is run. However, for the following reasons, we recommend not to set the option to `true`.
  - Background load can interfere with query-specific metadata loading. This can happen on startup or after invalidating metadata, with a duration depending on the amount of metadata, and can lead to a seemingly random long running queries that are difficult to diagnose.
  - Impala may load metadata for tables that are possibly never used, potentially increasing catalog size and consequently memory usage for both catalog service and Impala Daemon.

Starting Impala

To activate Impala if it is installed but not yet started:

1. Set any necessary configuration options for the Impala services. See Modifying Impala Startup Options for details.
2. Start one instance of the Impala statestore. The statestore helps Impala to distribute work efficiently, and to continue running in the event of availability problems for other Impala nodes. If the statestore becomes unavailable, Impala continues to function.
3. Start one instance of the Impala catalog service.
4. Start the main Impala service on one or more DataNodes, ideally on all DataNodes to maximize local processing and avoid network traffic due to remote reads.

Once Impala is running, you can conduct interactive experiments using the instructions in Impala Tutorials on page 43 and try Using the Impala Shell (impala-shell Command) on page 713.

Starting Impala from the Command Line

To start the Impala state store and Impala from the command line or a script, you can either use the `service` command or you can start the daemons directly through the `impalad`, `statestored`, and `catalogd` executables.

Start the Impala statestore and then start `impalad` instances. You can modify the values the service initialization scripts use when starting the statestore and Impala by editing `/etc/default/impala`.

Start the statestore service using a command similar to the following:

```
$ sudo service impala-state-store start
```

Start the catalog service using a command similar to the following:

```
$ sudo service impala-catalog start
```

Start the Impala service on each DataNode using a command similar to the following:

```
$ sudo service impala-server start
```



Note:

In CDH 5.7 / Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new `CREATE FUNCTION` syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old `CREATE FUNCTION` syntax do not persist across restarts because they are held in the memory of the `catalogd` daemon. Until you re-create such Java UDFs using the new `CREATE FUNCTION` syntax, you must reload those Java-based UDFs by running the original `CREATE FUNCTION` statements again each time you restart the `catalogd` daemon. Prior to CDH 5.7 / Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

If any of the services fail to start, review:

- Reviewing Impala Logs on page 710
- Troubleshooting Impala on page 730

Installing Impala with Cloudera Manager

Before installing Impala through the Cloudera Manager interface, make sure all applicable nodes have the appropriate hardware configuration and levels of operating system and CDH. See Impala Requirements on page 23 for details.

For information on installing Impala in a Cloudera Manager-managed environment, see http://www.cloudera.com/documentation/enterprise/latest/topics/cm_ig_install_impala.html

Managing your Impala installation through Cloudera Manager has a number of advantages. For example, when you make configuration changes to CDH components using Cloudera Manager, it automatically applies changes to the

copies of configuration files, such as `hive-site.xml`, that Impala keeps under `/etc/impala/conf`. It also sets up the Hive Metastore service that is required for Impala.

In some cases, depending on the level of Impala, CDH, and Cloudera Manager, you might need to add particular component configuration details in some of the free-form option fields on the Impala configuration pages within Cloudera Manager. In Cloudera Manager 4, these fields are labelled **Safety Valve**; in Cloudera Manager 5, they are called **Advanced Configuration Snippet**.

Installing Impala from the Command Line

Before installing Impala manually, make sure all applicable nodes have the appropriate hardware configuration, levels of operating system and CDH, and any other software prerequisites. See Impala Requirements on page 23 for details.

You can install Impala across many hosts or on one host:

- Installing Impala across multiple machines creates a distributed configuration. For best performance, install Impala on **all** DataNodes.
- Installing Impala on a single machine produces a pseudo-distributed cluster.

To install Impala on a host:

1. Install CDH, including Hive, as described in Installing and Deploying Unmanaged CDH Using the Command Line.
2. Configure the Hive metastore to use an external database as a metastore. Impala uses this same database for its own table metadata. You can choose either a MySQL or PostgreSQL database as the metastore. The process for configuring each type of database is described in the CDH Installation Guide).

   Cloudera recommends setting up a Hive metastore service rather than connecting directly to the metastore database; this configuration is required when running Impala under CDH 4.1. Make sure the `/etc/impala/conf/hive-site.xml` file contains the following setting, substituting the appropriate hostname for *metastore_server_host*:

```
<property>
<name>hive.metastore.uris</name>
<value>thrift://metastore_server_host:9083</value>
</property>
<property>
<name>hive.metastore.client.socket.timeout</name>
<value>3600</value>
<description>MetaStore Client socket timeout in seconds</description>
</property>
```

3. (Optional) If you installed the full Hive component on any host, you can verify that the metastore is configured properly by starting the Hive console and querying for the list of available tables. Once you confirm that the console starts, exit the console to continue the installation:

```
$ hive
Hive history file=/tmp/root/hive_job_log_root_201207272011_678722950.txt
hive> show tables;
table1
table2
hive> quit;
$
```

4. Confirm that your package management command is aware of the Impala repository settings, as described in Impala Requirements on page 23. (For CDH 4, this is a different repository than for CDH.) You might need to download a repo or list file into a system directory underneath `/etc`.
5. Use **one** of the following sets of commands to install the Impala package:

   For RHEL, Oracle Linux, or CentOS systems:

```
$ sudo yum install impala             # Binaries for daemons
$ sudo yum install impala-server      # Service start/stop script
```

```
$ sudo yum install impala-state-store # Service start/stop script
$ sudo yum install impala-catalog     # Service start/stop script
```

For SUSE systems:

```
$ sudo zypper install impala              # Binaries for daemons
$ sudo zypper install impala-server       # Service start/stop script
$ sudo zypper install impala-state-store # Service start/stop script
$ sudo zypper install impala-catalog      # Service start/stop script
```

For Debian or Ubuntu systems:

```
$ sudo apt-get install impala              # Binaries for daemons
$ sudo apt-get install impala-server       # Service start/stop script
$ sudo apt-get install impala-state-store # Service start/stop script
$ sudo apt-get install impala-catalog      # Service start/stop script
```



Note: Cloudera recommends that you not install Impala on any HDFS NameNode. Installing Impala on NameNodes provides no additional data locality, and executing queries with such a configuration might cause memory contention and negatively impact the HDFS NameNode.

6. Copy the client `hive-site.xml`, `core-site.xml`, `hdfs-site.xml`, and `hbase-site.xml` configuration files to the Impala configuration directory, which defaults to `/etc/impala/conf`. Create this directory if it does not already exist.
7. Use **one** of the following commands to install `impala-shell` on the machines from which you want to issue queries. You can install `impala-shell` on any supported machine that can connect to DataNodes that are running `impalad`.

   For RHEL/CentOS systems:

```
$ sudo yum install impala-shell
```

   For SUSE systems:

```
$ sudo zypper install impala-shell
```

   For Debian/Ubuntu systems:

```
$ sudo apt-get install impala-shell
```

8. Complete any required or recommended configuration, as described in Post-Installation Configuration for Impala on page 36. Some of these configuration changes are mandatory.

Once installation and configuration are complete, see Starting Impala on page 32 for how to activate the software on the appropriate nodes in your cluster.

If this is your first time setting up and using Impala in this cluster, run through some of the exercises in Impala Tutorials on page 43 to verify that you can do basic operations such as creating tables and querying them.

Managing Impala

The first part of this section describes the following Impala configuration topics:

- The Impala Service
- Post-Installation Configuration for Impala on page 36
- Impala Security on page 82
- Modifying Impala Startup Options

The rest of this section describes how to configure Impala to accept connections from applications that use popular programming APIs:

- Configuring Impala to Work with ODBC on page 723
- Configuring Impala to Work with JDBC on page 725

This type of configuration is especially useful when using Impala in combination with Business Intelligence tools, which use these standard interfaces to query different kinds of database and Big Data systems.

Post-Installation Configuration for Impala

This section describes the mandatory and recommended configuration settings for Impala. If Impala is installed using Cloudera Manager, some of these configurations are completed automatically; you must still configure short-circuit reads manually. If you installed Impala without Cloudera Manager, or if you want to customize your environment, consider making the changes described in this topic.

In some cases, depending on the level of Impala, CDH, and Cloudera Manager, you might need to add particular component configuration details in one of the free-form fields on the Impala configuration pages within Cloudera Manager. In Cloudera Manager 4, these fields are labelled **Safety Valve**; in Cloudera Manager 5, they are called **Advanced Configuration Snippet**.

- You must enable short-circuit reads, whether or not Impala was installed through Cloudera Manager. This setting goes in the Impala configuration settings, not the Hadoop-wide settings.
- If you installed Impala in an environment that is not managed by Cloudera Manager, you must enable block location tracking, and you can optionally enable native checksumming for optimal performance.
- If you deployed Impala using Cloudera Manager see Testing Impala Performance on page 601 to confirm proper configuration.

Mandatory: Short-Circuit Reads

Enabling short-circuit reads allows Impala to read local data directly from the file system. This removes the need to communicate through the DataNodes, improving performance. This setting also minimizes the number of additional copies of data. Short-circuit reads requires `libhadoop.so` (the Hadoop Native Library) to be accessible to both the server and the client. You must install it from an `.rpm`, `.deb`, or parcel to use short-circuit local reads.



Note: If you use Cloudera Manager, you can enable short-circuit reads through a checkbox in the user interface and that setting takes effect for Impala as well.

To configure DataNodes for short-circuit reads:

1. Copy the client `core-site.xml` and `hdfs-site.xml` configuration files from the Hadoop configuration directory to the Impala configuration directory. The default Impala configuration location is `/etc/impala/conf`.
2. On all Impala nodes, configure the following properties in Impala's copy of `hdfs-site.xml` as shown:

```
<property>
    <name>dfs.client.read.shortcircuit</name>
```

```
    <value>true</value>
</property>

<property>
    <name>dfs.domain.socket.path</name>
    <value>/var/run/hdfs-sockets/dn</value>
</property>

<property>
    <name>dfs.client.file-block-storage-locations.timeout.millis</name>
    <value>10000</value>
</property>
```

3. If `/var/run/hadoop-hdfs/` is group-writable, make sure its group is `root`.

> **Note:** If you are also going to enable block location tracking, you can skip copying configuration files and restarting DataNodes and go straight to Optional: Block Location Tracking. Configuring short-circuit reads and block location tracking require the same process of copying files and restarting services, so you can complete that process once when you have completed all configuration changes. Whether you copy files and restart services now or during configuring block location tracking, short-circuit reads are not enabled until you complete those final steps.

4. After applying these changes, restart all DataNodes.

Mandatory: Block Location Tracking

Enabling block location metadata allows Impala to know which disk data blocks are located on, allowing better utilization of the underlying disks. Impala will not start unless this setting is enabled.

To enable block location tracking:

1. For each DataNode, adding the following to the `hdfs-site.xml` file:

```
<property>
  <name>dfs.datanode.hdfs-blocks-metadata.enabled</name>
  <value>true</value>
</property>
```

2. Copy the client `core-site.xml` and `hdfs-site.xml` configuration files from the Hadoop configuration directory to the Impala configuration directory. The default Impala configuration location is `/etc/impala/conf`.
3. After applying these changes, restart all DataNodes.

Optional: Native Checksumming

Enabling native checksumming causes Impala to use an optimized native library for computing checksums, if that library is available.

To enable native checksumming:

If you installed CDH from packages, the native checksumming library is installed and setup correctly. In such a case, no additional steps are required. Conversely, if you installed by other means, native checksumming may not be available due to missing shared objects. Finding the message "`Unable to load native-hadoop library for your platform... using builtin-java classes where applicable`" in the Impala logs indicates native checksumming may be unavailable. To enable native checksumming, you must build and install `libhadoop.so` (the Hadoop Native Library).

Impala Upgrade Considerations

Converting Legacy UDFs During Upgrade to CDH 5.12 or Higher

In CDH 5.7 / Impala 2.5 and higher, the CREATE FUNCTION Statement on page 228 is available for creating Java-based UDFs. UDFs created with the new syntax persist across Impala restarts, and are more compatible with Hive UDFs. Because the replication features in CDH 5.12 and higher only work with the new-style syntax, convert any older Java UDFs to use the new syntax at the same time you upgrade to CDH 5.12 or higher.

Follow these steps to convert old-style Java UDFs to the new persistent kind:

- Use `SHOW FUNCTIONS` to identify all UDFs and UDAs.
- For each function, use `SHOW CREATE FUNCTION` and save the statement in a script file.
- For Java UDFs, change the output of `SHOW CREATE FUNCTION` to use the new `CREATE FUNCTION` syntax (without argument types), which makes the UDF persistent.
- For each function, drop it and re-create it, using the new `CREATE FUNCTION` syntax for all Java UDFs.

Handling Large Rows During Upgrade to CDH 5.13 / Impala 2.10 or Higher

In CDH 5.13 / Impala 2.10 and higher, the handling of memory management for large column values is different than in previous releases. Some queries that succeeded previously might now fail immediately with an error message. The `--read_size` option no longer needs to be increased from its default of 8 MB for queries against tables with huge column values. Instead, the query option `MAX_ROW_SIZE` lets you fine-tune this value at the level of individual queries or sessions. The default for `MAX_ROW_SIZE` is 512 KB. If your queries process rows with column values totalling more than 512 KB, you might need to take action to avoid problems after upgrading.

Follow these steps to verify if your deployment needs any special setup to deal with the new way of dealing with large rows:

1. Check if your `impalad` daemons are already running with a larger-than-normal value for the `--read_size` configuration setting.
2. Examine all tables to find if any have `STRING` values that are hundreds of kilobytes or more in length. This information is available under the `Max Size` column in the output from the `SHOW TABLE STATS` statement, after the `COMPUTE STATS` statement has been run on the table. In the following example, the `S1` column with a maximum length of `700006` could cause an issue by itself, or if a combination of values from the `S1`, `S2`, and `S3` columns exceeded the 512 KB `MAX_ROW_SIZE` value.

```
show column stats big_strings;
+--------+--------+------------------+--------+----------+-------------------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size          |
+--------+--------+------------------+--------+----------+-------------------+
| x      | BIGINT | 30000            | -1     | 8        | 8                 |
| s1     | STRING | 30000            | -1     | 700006   | 392625            |
| s2     | STRING | 30000            | -1     | 10532    | 9232.6669921875   |
| s3     | STRING | 30000            | -1     | 103      | 87.66670227050781 |
+--------+--------+------------------+--------+----------+-------------------+
```

3. For each candidate table, run a query to materialize the largest string values from the largest columns all at once. Check if the query fails with a message suggesting to set the `MAX_ROW_SIZE` query option.

```
select count(distinct s1, s2, s3) from little_strings;
+---------------------------+
| count(distinct s1, s2, s3) |
+---------------------------+
| 30000                     |
```

```
+---------------------------+

select count(distinct s1, s2, s3) from big_strings;
WARNINGS: Row of size 692.13 KB could not be materialized in plan node with id 1.
  Increase the max_row_size query option (currently 512.00 KB) to process larger rows.
```

If any of your tables are affected, make sure the MAX_ROW_SIZE is set large enough to allow all queries against the affected tables to deal with the large column values:

- In SQL scripts run by impala-shell with the -q or -f options, or in interactive impala-shell sessions, issue a statement SET MAX_ROW_SIZE=*large_enough_size* before the relevant queries:

```
$ impala-shell -i localhost -q \
  'set max_row_size=1mb; select count(distinct s1, s2, s3) from big_strings'
```

- If large column values are common to many of your tables and it is not practical to set MAX_ROW_SIZE only for a limited number of queries or scripts, use the --default_query_options configuration setting for all your impalad daemons, and include the larger MAX_ROW_SIZE setting as part of the argument to that setting. For example:

```
impalad --default_query_options='max_row_size=1gb;appx_count_distinct=true'
```

- If your deployment uses a non-default value for the --read_size configuration setting, remove that setting and let Impala use the default. A high value for --read_size could cause higher memory consumption in CDH 5.13 / Impala 2.10 and higher than in previous versions. The --read_size setting still controls the HDFS I/O read size (which is rarely if ever necessary to change), but no longer affects the spill-to-disk buffer size.

Change Impala catalogd Heap when Upgrading from CDH 5.6 or Lower

The default heap size for Impala catalogd has changed in CDH 5.7 / Impala 2.5 and higher:

- Before 5.7, by default catalogd was using the JVM's default heap size, which is the smaller of 1/4th of the physical memory or 32 GB.
- Starting with CDH 5.7.0, the default catalogd heap size is 4 GB.

For example, on a host with 128 GB physical memory this will result in catalogd heap decreasing from 32 GB to 4 GB.

For schemas with large numbers of tables, partitions, and data files, the catalogd daemon might encounter an out-of-memory error. To prevent the error, increase the memory limit for the catalogd daemon:

1. Check current memory usage for the catalogd daemon by running the following commands on the host where that daemon runs on your cluster:

```
jcmd catalogd_pid VM.flags
jmap -heap catalogd_pid
```

2. Decide on a large enough value for the catalogd heap.
   - On systems managed by Cloudera Manager, include this value in the configuration field **Java Heap Size of Catalog Server in Bytes** (Cloudera Manager 5.7 and higher), or **Impala Catalog Server Environment Advanced Configuration Snippet (Safety Valve)** (prior to Cloudera Manager 5.7). Then restart the Impala service.
   - On systems not managed by Cloudera Manager, put the JAVA_TOOL_OPTIONS environment variable setting into the startup script for the catalogd daemon, then restart the catalogd daemon.

For example, the following environment variable setting specifies the maximum heap size of 8 GB.

```
JAVA_TOOL_OPTIONS="-Xmx8g"
```

3. Use the same `jcmd` and `jmap` commands as earlier to verify that the new settings are in effect.

List of Reserved Words Updated in CDH 6.0 / Impala 3.0

The list of Impala Reserved Words on page 745 in Impala was updated in CDH 6.0 / Impala 3.0. If you need to use a reserved word as an identifier, e.g. a table name, enclose the word in back-ticks.

If you need to use the reserved words from previous versions of CDH, set the impalad and catalogd startup option, `--reserved_words_version`, to `"2.11.0"`.

Decimal V2 Used by Default in CDH 6.0 / Impala 3.0

In Impala, two different behaviors of `DECIMAL` types are supported. In CDH 6.0 / Impala 3.0, `DECIMAL` V2 is used by default. See DECIMAL Data Type for detail information.

If you need to continue using the first version of the `DECIMAL` type for the backward compatibility of your queries, set the `DECIMAL_V2` query option to `FALSE`:

```
SET DECIMAL_V2=FALSE;
```

Behavior of Column Aliases Changed in CDH 6.0 / Impala 3.0

To conform to the SQL standard, Impala no longer performs alias substitution in the subexpressions of `GROUP BY`, `HAVING`, and `ORDER BY`. See Overview of Impala Aliases on page 192 for examples of supported and unsupported aliases syntax.

Default PARQUET_ARRAY_RESOLUTION Changed in CDH 6.0 / Impala 3.0

The default value for the `PARQUET_ARRAY_RESOLUTION` was changed to `THREE_LEVEL` in CDH 6.0 / Impala 3.0, to match the Parquet standard 3-level encoding.

See PARQUET_RESOLUTION Query option for the information about the query option.

Enable Clustering Hint for Inserts

In CDH 6.0 / Impala 3.0, the clustered hint is enabled by default. The hint adds a local sort by the partitioning columns to a query plan.

The `clustered` hint is only effective for HDFS and Kudu tables.

As in previous versions, the `noclustered` hint prevents clustering. If a table has ordering columns defined, the `noclustered` hint is ignored with a warning.

Deprecated Query Options Removed in CDH 6.0 / Impala 3.0

The following query options have been deprecated for several releases and removed:

- DEFAULT_ORDER_BY_LIMIT
- ABORT_ON_DEFAULT_LIMIT_EXCEEDED
- V_CPU_CORES
- RESERVATION_REQUEST_TIMEOUT
- RM_INITIAL_MEM
- SCAN_NODE_CODEGEN_THRESHOLD
- MAX_IO_BUFFERS
- RM_INITIAL_MEM
- DISABLE_CACHED_READS

refresh_after_connect Impala Shell Option Removed in CDH 6.0 / Impala 3.0

The deprecated `refresh_after_connect` option was removed from Impala Shell in CDH 6.0 / Impala 3.0

Return Type Changed for EXTRACT and DATE_PART Functions in CDH 6.0 / Impala 3.0

The following changes were made to the EXTRACT and DATE_PART functions:

- The output type of the EXTRACT and DATE_PART functions was changed to BIGINT.
- Extracting the millisecond part from a TIMESTAMP returns the seconds component and the milliseconds component. For example, `EXTRACT (CAST('2006-05-12 18:27:28.123456789' AS TIMESTAMP), 'MILLISECOND')` will return 28123.

Impala Roles with SELECT or INSERT Privilege Receive REFRESH Privilege During the Upgrade to CDH 5.16 / CDH 6.1

Due to the Sentry and Impala fine grained privileges feature in CDH 5.16 / CDH 6.1, if a role has the SELECT or INSERT privilege on an object in Impala before upgrading to CDH 5.16 / CDH 6.1, that role will automatically get the REFRESH privilege during the upgrade.

Port Change for SHUTDOWN Command

If you used the SHUTDOWN command in CDH 6.1, and specified a port explicitly, change the port number parameter, in CDH 6.2, to use the KRPC port.

Change in Client Connection Timeout

The default behavior of client connection timeout changed.

In CDH 6.2 and lower, client waited indefinitely to open the new session if the maximum number of threads specified by `--fe_service_threads` has been allocated.

In CDH 6.3 and higher, a new startup flag, `--accepted_client_cnxn_timeout`, was added to control how the server should treat new connection requests if we have run out of the configured number of server threads.

If `--accepted_client_cnxn_timeout > 0`, new connection requests are rejected after the specified timeout.

If `--accepted_client_cnxn_timeout=0`, clients waits indefinitely to connect to Impala. You can use this setting to restore the pre-CDH 6.3 behavior.

The default timeout is 5 minutes.

Default Setting Changes

Release Changed	Setting	Default Value
CDH 5.15 & CDH 6.1 / Impala 2.12	`--compact_catalog_topic` impalad flag	`true`
CDH 6.1 / Impala 2.12	`--max_cached_file_handles` impalad flag	`20000`
CDH 6.0 / Impala 3.0	`PARQUET_ARRAY_RESOLUTION` query option	`THREE_LEVEL`
CDH 6.0 / Impala 3.0	`DECIMAL_V2` query option	`TRUE`

Impala Tutorials

This section includes tutorial scenarios that demonstrate how to begin using Impala once the software is installed. It focuses on techniques for loading data, because once you have some data in tables and can query that data, you can quickly progress to more advanced Impala features.



Note:

Where practical, the tutorials take you from "ground zero" to having the desired Impala tables and data. In some cases, you might need to download additional files from outside sources, set up additional software components, modify commands or scripts to fit your own configuration, or substitute your own sample data.

Before trying these tutorial lessons, install Impala using one of these procedures:

- If you already have some CDH environment set up and just need to add Impala to it, add the **Impala** service using the instructions in Adding a Service. Make sure to also install the Hive metastore service if you do not already have Hive configured.
- To set up Impala and all its prerequisites at once, in a minimal configuration that you can use for small-scale experiments, set up the Cloudera QuickStart VM, which includes CDH and Impala on CentOS. Use this single-node VM to try out basic SQL functionality, not anything related to performance and scalability. For more information, see the Cloudera QuickStart VM.

Tutorials for Getting Started

These tutorials demonstrate the basics of using Impala. They are intended for first-time users, and for trying out Impala on any new cluster to make sure the major components are working correctly.

Explore a New Impala Instance

This tutorial demonstrates techniques for finding your way around the tables and databases of an unfamiliar (possibly empty) Impala instance.

When you connect to an Impala instance for the first time, you use the `SHOW DATABASES` and `SHOW TABLES` statements to view the most common types of objects. Also, call the `version()` function to confirm which version of Impala you are running; the version number is important when consulting documentation and dealing with support issues.

A completely empty Impala instance contains no tables, but still has two databases:

- `default`, where new tables are created when you do not specify any other database.
- `_impala_builtins`, a system database used to hold all the built-in functions.

The following example shows how to see the available databases, and the tables in each. If the list of databases or tables is long, you can use wildcard notation to locate specific databases or tables based on their names.

```
$ impala-shell -i localhost --quiet
Starting Impala Shell without Kerberos authentication
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v...
[localhost:21000] > select version();
+------------------------------------------
| version()
+------------------------------------------
| impalad version ...
| Built on ...
```

```
+-----------------------------------------
[localhost:21000] > show databases;
+--------------------------+
| name                     |
+--------------------------+
| _impala_builtins         |
| ctas                     |
| d1                       |
| d2                       |
| d3                       |
| default                  |
| explain_plans            |
| external_table           |
| file_formats             |
| tpc                      |
+--------------------------+
[localhost:21000] > select current_database();
+--------------------+
| current_database() |
+--------------------+
| default            |
+--------------------+
[localhost:21000] > show tables;
+-------+
| name  |
+-------+
| ex_t  |
| t1    |
+-------+
[localhost:21000] > show tables in d3;

[localhost:21000] > show tables in tpc;
+------------------------+
| name                   |
+------------------------+
| city                   |
| customer               |
| customer_address       |
| customer_demographics  |
| household_demographics |
| item                   |
| promotion              |
| store                  |
| store2                 |
| store_sales            |
| ticket_view            |
| time_dim               |
| tpc_tables             |
+------------------------+
[localhost:21000] > show tables in tpc like 'customer*';
+-----------------------+
| name                  |
+-----------------------+
| customer              |
| customer_address      |
| customer_demographics |
+-----------------------+
```

Once you know what tables and databases are available, you descend into a database with the USE statement. To understand the structure of each table, you use the DESCRIBE command. Once inside a database, you can issue statements such as INSERT and SELECT that operate on particular tables.

The following example explores a database named TPC whose name we learned in the previous example. It shows how to filter the table names within a database based on a search string, examine the columns of a table, and run queries to examine the characteristics of the table data. For example, for an unfamiliar table you might want to know the number of rows, the number of different values for a column, and other properties such as whether the column

contains any NULL values. When sampling the actual data values from a table, use a LIMIT clause to avoid excessive output if the table contains more rows or distinct values than you expect.

```
[localhost:21000] > use tpc;
[localhost:21000] > show tables like '*view*';
+-------------+
| name        |
+-------------+
| ticket_view |
+-------------+
[localhost:21000] > describe city;
+-------------+--------+---------+
| name        | type   | comment |
+-------------+--------+---------+
| id          | int    |         |
| name        | string |         |
| countrycode | string |         |
| district    | string |         |
| population  | int    |         |
+-------------+--------+---------+
[localhost:21000] > select count(*) from city;
+----------+
| count(*) |
+----------+
| 0        |
+----------+
[localhost:21000] > desc customer;
+------------------------+--------+---------+
| name                   | type   | comment |
+------------------------+--------+---------+
| c_customer_sk          | int    |         |
| c_customer_id          | string |         |
| c_current_cdemo_sk     | int    |         |
| c_current_hdemo_sk     | int    |         |
| c_current_addr_sk      | int    |         |
| c_first_shipto_date_sk | int    |         |
| c_first_sales_date_sk  | int    |         |
| c_salutation           | string |         |
| c_first_name           | string |         |
| c_last_name            | string |         |
| c_preferred_cust_flag  | string |         |
| c_birth_day            | int    |         |
| c_birth_month          | int    |         |
| c_birth_year           | int    |         |
| c_birth_country        | string |         |
| c_login                | string |         |
| c_email_address        | string |         |
| c_last_review_date     | string |         |
+------------------------+--------+---------+
[localhost:21000] > select count(*) from customer;
+----------+
| count(*) |
+----------+
| 100000   |
+----------+
[localhost:21000] > select count(distinct c_birth_month) from customer;
+------------------------------+
| count(distinct c_birth_month) |
+------------------------------+
| 12                           |
+------------------------------+
[localhost:21000] > select count(*) from customer where c_email_address is null;
+----------+
| count(*) |
+----------+
| 0        |
+----------+
[localhost:21000] > select distinct c_salutation from customer limit 10;
+--------------+
| c_salutation |
+--------------+
| Mr.          |
| Ms.          |
```

```
| Dr.          |
|              |
| Miss         |
| Sir          |
| Mrs.         |
+--------------+
```

When you graduate from read-only exploration, you use statements such as CREATE DATABASE and CREATE TABLE to set up your own database objects.

The following example demonstrates creating a new database holding a new table. Although the last example ended inside the TPC database, the new EXPERIMENTS database is not nested inside TPC; all databases are arranged in a single top-level list.

```
[localhost:21000] > create database experiments;
[localhost:21000] > show databases;
+--------------------------+
| name                     |
+--------------------------+
| _impala_builtins         |
| ctas                     |
| d1                       |
| d2                       |
| d3                       |
| default                  |
| experiments              |
| explain_plans            |
| external_table           |
| file_formats             |
| tpc                      |
+--------------------------+
[localhost:21000] > show databases like 'exp*';
+---------------+
| name          |
+---------------+
| experiments   |
| explain_plans |
+---------------+
```

The following example creates a new table, T1. To illustrate a common mistake, it creates this table inside the wrong database, the TPC database where the previous example ended. The ALTER TABLE statement lets you move the table to the intended database, EXPERIMENTS, as part of a rename operation. The USE statement is always needed to switch to a new database, and the current_database() function confirms which database the session is in, to avoid these kinds of mistakes.

```
[localhost:21000] > create table t1 (x int);

[localhost:21000] > show tables;
+------------------------+
| name                   |
+------------------------+
| city                   |
| customer               |
| customer_address       |
| customer_demographics  |
| household_demographics |
| item                   |
| promotion              |
| store                  |
| store2                 |
| store_sales            |
| t1                     |
| ticket_view            |
| time_dim               |
| tpc_tables             |
+------------------------+
[localhost:21000] > select current_database();
+--------------------+
| current_database() |
```

```
+--------------------+
| tpc                |
+--------------------+
[localhost:21000] > alter table t1 rename to experiments.t1;
[localhost:21000] > use experiments;
[localhost:21000] > show tables;
+------+
| name |
+------+
| t1   |
+------+
[localhost:21000] > select current_database();
+--------------------+
| current_database() |
+--------------------+
| experiments        |
+--------------------+
```

For your initial experiments with tables, you can use ones with just a few columns and a few rows, and text-format data files.



Note: As you graduate to more realistic scenarios, you will use more elaborate tables with many columns, features such as partitioning, and file formats such as Parquet. When dealing with realistic data volumes, you will bring in data using `LOAD DATA` or `INSERT ... SELECT` statements to operate on millions or billions of rows at once.

The following example sets up a couple of simple tables with a few rows, and performs queries involving sorting, aggregate functions and joins.

```
[localhost:21000] > insert into t1 values (1), (3), (2), (4);
[localhost:21000] > select x from t1 order by x desc;
+---+
| x |
+---+
| 4 |
| 3 |
| 2 |
| 1 |
+---+
[localhost:21000] > select min(x), max(x), sum(x), avg(x) from t1;
+--------+--------+--------+--------+
| min(x) | max(x) | sum(x) | avg(x) |
+--------+--------+--------+--------+
| 1      | 4      | 10     | 2.5    |
+--------+--------+--------+--------+

[localhost:21000] > create table t2 (id int, word string);
[localhost:21000] > insert into t2 values (1, "one"), (3, "three"), (5, 'five');
[localhost:21000] > select word from t1 join t2 on (t1.x = t2.id);
+-------+
| word  |
+-------+
| one   |
| three |
+-------+
```

After completing this tutorial, you should now know:

- How to tell which version of Impala is running on your system.
- How to find the names of databases in an Impala instance, either displaying the full list or searching for specific names.
- How to find the names of tables in an Impala database, either displaying the full list or searching for specific names.
- How to switch between databases and check which database you are currently in.
- How to learn the column names and types of a table.
- How to create databases and tables, insert small amounts of test data, and run simple queries.

Load CSV Data from Local Files

This scenario illustrates how to create some very small tables, suitable for first-time users to experiment with Impala SQL features. TAB1 and TAB2 are loaded with data from files in HDFS. A subset of data is copied from TAB1 into TAB3.

Populate HDFS with the data you want to query. To begin this process, create one or more new subdirectories underneath your user directory in HDFS. The data for each table resides in a separate subdirectory. Substitute your own username for `username` where appropriate. This example uses the `-p` option with the `mkdir` operation to create any necessary parent directories if they do not already exist.

```
$ whoami
username
$ hdfs dfs -ls /user
Found 3 items
drwxr-xr-x   - username username            0 2013-04-22 18:54 /user/username
drwxrwx---   - mapred   mapred              0 2013-03-15 20:11 /user/history
drwxr-xr-x   - hue      supergroup          0 2013-03-15 20:10 /user/hive

$ hdfs dfs -mkdir -p /user/username/sample_data/tab1 /user/username/sample_data/tab2
```

Here is some sample data, for two tables named TAB1 and TAB2.

Copy the following content to `.csv` files in your local filesystem:

`tab1.csv`:

```
1,true,123.123,2012-10-24 08:55:00
2,false,1243.5,2012-10-25 13:40:00
3,false,24453.325,2008-08-22 09:33:21.123
4,false,243423.325,2007-05-12 22:32:21.33454
5,true,243.325,1953-04-22 09:11:33
```

`tab2.csv`:

```
1,true,12789.123
2,false,1243.5
3,false,24453.325
4,false,2423.3254
5,true,243.325
60,false,243565423.325
70,true,243.325
80,false,243423.325
90,true,243.325
```

Put each `.csv` file into a separate HDFS directory using commands like the following, which use paths available in the Impala Demo VM:

```
$ hdfs dfs -put tab1.csv /user/username/sample_data/tab1
$ hdfs dfs -ls /user/username/sample_data/tab1
Found 1 items
-rw-r--r--   1 username username        192 2013-04-02 20:08
/user/username/sample_data/tab1/tab1.csv


$ hdfs dfs -put tab2.csv /user/username/sample_data/tab2
$ hdfs dfs -ls /user/username/sample_data/tab2
Found 1 items
-rw-r--r--   1 username username        158 2013-04-02 20:09
/user/username/sample_data/tab2/tab2.csv
```

The name of each data file is not significant. In fact, when Impala examines the contents of the data directory for the first time, it considers all files in the directory to make up the data of the table, regardless of how many files there are or what the files are named.

To understand what paths are available within your own HDFS filesystem and what the permissions are for the various directories and files, issue `hdfs dfs -ls /` and work your way down the tree doing `-ls` operations for the various directories.

Use the `impala-shell` command to create tables, either interactively or through a SQL script.

The following example shows creating three tables. For each table, the example shows creating columns with various attributes such as Boolean or integer types. The example also includes commands that provide information about how the data is formatted, such as rows terminating with commas, which makes sense in the case of importing data from a `.csv` file. Where we already have `.csv` files containing data in the HDFS directory tree, we specify the location of the directory containing the appropriate `.csv` file. Impala considers all the data from all the files in that directory to represent the data for the table.

```
DROP TABLE IF EXISTS tab1;
-- The EXTERNAL clause means the data is located outside the central location
-- for Impala data files and is preserved when the associated Impala table is dropped.
-- We expect the data to already exist in the directory specified by the LOCATION clause.
CREATE EXTERNAL TABLE tab1
(
   id INT,
   col_1 BOOLEAN,
   col_2 DOUBLE,
   col_3 TIMESTAMP
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/user/username/sample_data/tab1';

DROP TABLE IF EXISTS tab2;
-- TAB2 is an external table, similar to TAB1.
CREATE EXTERNAL TABLE tab2
(
   id INT,
   col_1 BOOLEAN,
   col_2 DOUBLE
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/user/username/sample_data/tab2';

DROP TABLE IF EXISTS tab3;
-- Leaving out the EXTERNAL clause means the data will be managed
-- in the central Impala data directory tree. Rather than reading
-- existing data files when the table is created, we load the
-- data after creating the table.
CREATE TABLE tab3
(
   id INT,
   col_1 BOOLEAN,
   col_2 DOUBLE,
   month INT,
   day INT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```



Note: Getting through these CREATE TABLE statements successfully is an important validation step to confirm everything is configured correctly with the Hive metastore and HDFS permissions. If you receive any errors during the CREATE TABLE statements:

- Make sure the `hive.metastore.warehouse.dir` property points to a directory that Impala can write to. The ownership should be `hive:hive`, and the `impala` user should also be a member of the `hive` group.
- If the value of `hive.metastore.warehouse.dir` is different in the Cloudera Manager dialogs and in the Hive shell, you might need to designate the hosts running `impalad` with the "gateway" role for Hive, and deploy the client configuration files to those hosts.

Point an Impala Table at Existing Data Files

A convenient way to set up data for Impala to access is to use an external table, where the data already exists in a set of HDFS files and you just point the Impala table at the directory containing those files. For example, you might run in

`impala-shell` a `*.sql` file with contents similar to the following, to create an Impala table that accesses an existing data file used by Hive.

The following examples set up 2 tables, referencing the paths and sample data from the sample TPC-DS kit for Impala. For historical reasons, the data physically resides in an HDFS directory tree under `/user/hive`, although this particular data is entirely managed by Impala rather than Hive. When we create an external table, we specify the directory containing one or more data files, and Impala queries the combined content of all the files inside that directory. Here is how we examine the directories and files within the HDFS filesystem:

```
$ cd ~/username/datasets
$ ./tpcds-setup.sh
... Downloads and unzips the kit, builds the data and loads it into HDFS ...
$ hdfs dfs -ls /user/hive/tpcds/customer
Found 1 items
-rw-r--r--   1 username supergroup   13209372 2013-03-22 18:09
/user/hive/tpcds/customer/customer.dat
$ hdfs dfs -cat /user/hive/tpcds/customer/customer.dat | more
1|AAAAAAAABAAAAAAA|980124|7135|32946|2452238|2452208|Mr.|Javier|Lewis|Y|9|12|1936|CHILE||Javie
r.Lewis@VFAxlnZEvOx.org|2452508|
2|AAAAAAAACAAAAAAA|819667|1461|31655|2452318|2452288|Dr.|Amy|Moses|Y|9|4|1966|TOGO||Amy.Moses@
Ovk9KjHH.com|2452318|
3|AAAAAAAADAAAAAAA|1473522|6247|48572|2449130|2449100|Miss|Latisha|Hamilton|N|18|9|1979|NIUE||
Latisha.Hamilton@V.com|2452313|
4|AAAAAAAAEAAAAAAA|1703214|3986|39558|2450030|2450000|Dr.|Michael|White|N|7|6|1983|MEXICO||Mic
hael.White@i.org|2452361|
5|AAAAAAAAFAAAAAAA|953372|4470|36368|2449438|2449408|Sir|Robert|Moran|N|8|5|1956|FIJI||Robert.
Moran@Hh.edu|2452469|
...
```

Here is a SQL script to set up Impala tables pointing to some of these data files in HDFS. (The script in the VM sets up tables like this through Hive; ignore those tables for purposes of this demonstration.) Save the following as `customer_setup.sql`:

```
--
-- store_sales fact table and surrounding dimension tables only
--
create database tpcds;
use tpcds;

drop table if exists customer;
create external table customer
(
    c_customer_sk             int,
    c_customer_id             string,
    c_current_cdemo_sk        int,
    c_current_hdemo_sk        int,
    c_current_addr_sk         int,
    c_first_shipto_date_sk    int,
    c_first_sales_date_sk     int,
    c_salutation              string,
    c_first_name              string,
    c_last_name               string,
    c_preferred_cust_flag     string,
    c_birth_day               int,
    c_birth_month             int,
    c_birth_year              int,
    c_birth_country           string,
    c_login                   string,
    c_email_address           string,
    c_last_review_date        string
)
row format delimited fields terminated by '|'
location '/user/hive/tpcds/customer';

drop table if exists customer_address;
create external table customer_address
(
    ca_address_sk             int,
    ca_address_id             string,
    ca_street_number          string,
```

```
    ca_street_name            string,
    ca_street_type            string,
    ca_suite_number           string,
    ca_city                   string,
    ca_county                 string,
    ca_state                  string,
    ca_zip                    string,
    ca_country                string,
    ca_gmt_offset             float,
    ca_location_type          string
)
row format delimited fields terminated by '|'
location '/user/hive/tpcds/customer_address';
```

We would run this script with a command such as:

```
impala-shell -i localhost -f customer_setup.sql
```

Describe the Impala Table

Now that you have updated the database metadata that Impala caches, you can confirm that the expected tables are accessible by Impala and examine the attributes of one of the tables. We created these tables in the database named `default`. If the tables were in a database other than the default, we would issue a command `use` *db_name* to switch to that database before examining or querying its tables. We could also qualify the name of a table by prepending the database name, for example `default.customer` and `default.customer_name`.

```
[impala-host:21000] > show databases
Query finished, fetching results ...
default
Returned 1 row(s) in 0.00s
[impala-host:21000] > show tables
Query finished, fetching results ...
customer
customer_address
Returned 2 row(s) in 0.00s
[impala-host:21000] > describe customer_address
+------------------+--------+---------+
| name             | type   | comment |
+------------------+--------+---------+
| ca_address_sk    | int    |         |
| ca_address_id    | string |         |
| ca_street_number | string |         |
| ca_street_name   | string |         |
| ca_street_type   | string |         |
| ca_suite_number  | string |         |
| ca_city          | string |         |
| ca_county        | string |         |
| ca_state         | string |         |
| ca_zip           | string |         |
| ca_country       | string |         |
| ca_gmt_offset    | float  |         |
| ca_location_type | string |         |
+------------------+--------+---------+
Returned 13 row(s) in 0.01
```

Query the Impala Table

You can query data contained in the tables. Impala coordinates the query execution across a single node or multiple nodes depending on your configuration, without the overhead of running MapReduce jobs to perform the intermediate processing.

There are a variety of ways to execute queries on Impala:

- Using the `impala-shell` command in interactive mode:

```
$ impala-shell -i impala-host
Connected to localhost:21000
```

```
[impala-host:21000] > select count(*) from customer_address;
50000
Returned 1 row(s) in 0.37s
```

- Passing a set of commands contained in a file:

```
$ impala-shell -i impala-host -f myquery.sql
Connected to localhost:21000
50000
Returned 1 row(s) in 0.19s
```

- Passing a single command to the `impala-shell` command. The query is executed, the results are returned, and the shell exits. Make sure to quote the command, preferably with single quotation marks to avoid shell expansion of characters such as `*`.

```
$ impala-shell -i impala-host -q 'select count(*) from customer_address'
Connected to localhost:21000
50000
Returned 1 row(s) in 0.29s
```

Data Loading and Querying Examples

This section describes how to create some sample tables and load data into them. These tables can then be queried using the Impala shell.

Loading Data

Loading data involves:

- Establishing a data set. The example below uses `.csv` files.
- Creating tables to which to load data.
- Loading the data into the tables you created.

Sample Queries

To run these sample queries, create a SQL query file `query.sql`, copy and paste each query into the query file, and then run the query file using the shell. For example, to run `query.sql` on `impala-host`, you might use the command:

```
impala-shell.sh -i impala-host -f query.sql
```

The examples and results below assume you have loaded the sample data into the tables as described above.

Example: Examining Contents of Tables

Let's start by verifying that the tables do contain the data we expect. Because Impala often deals with tables containing millions or billions of rows, when examining tables of unknown size, include the `LIMIT` clause to avoid huge amounts of unnecessary output, as in the final query. (If your interactive query starts displaying an unexpected volume of data, press `Ctrl-C` in `impala-shell` to cancel the query.)

```
SELECT * FROM tab1;
SELECT * FROM tab2;
SELECT * FROM tab2 LIMIT 5;
```

Results:

```
+----+-------+------------+-------------------------------+
| id | col_1 | col_2      | col_3                         |
+----+-------+------------+-------------------------------+
| 1  | true  | 123.123    | 2012-10-24 08:55:00           |
| 2  | false | 1243.5     | 2012-10-25 13:40:00           |
| 3  | false | 24453.325  | 2008-08-22 09:33:21.123000000 |
| 4  | false | 243423.325 | 2007-05-12 22:32:21.334540000 |
```

```
| 5  | true  | 243.325    | 1953-04-22 09:11:33           |
+----+-------+------------+-------------------------------+

+----+-------+---------------+
| id | col_1 | col_2         |
+----+-------+---------------+
| 1  | true  | 12789.123     |
| 2  | false | 1243.5        |
| 3  | false | 24453.325     |
| 4  | false | 2423.3254     |
| 5  | true  | 243.325       |
| 60 | false | 243565423.325 |
| 70 | true  | 243.325       |
| 80 | false | 243423.325    |
| 90 | true  | 243.325       |
+----+-------+---------------+

+----+-------+-----------+
| id | col_1 | col_2     |
+----+-------+-----------+
| 1  | true  | 12789.123 |
| 2  | false | 1243.5    |
| 3  | false | 24453.325 |
| 4  | false | 2423.3254 |
| 5  | true  | 243.325   |
+----+-------+-----------+
```

Example: Aggregate and Join

```
SELECT tab1.col_1, MAX(tab2.col_2), MIN(tab2.col_2)
FROM tab2 JOIN tab1 USING (id)
GROUP BY col_1 ORDER BY 1 LIMIT 5;
```

Results:

```
+-------+-----------------+-----------------+
| col_1 | max(tab2.col_2) | min(tab2.col_2) |
+-------+-----------------+-----------------+
| false | 24453.325       | 1243.5          |
| true  | 12789.123       | 243.325         |
+-------+-----------------+-----------------+
```

Example: Subquery, Aggregate and Joins

```
SELECT tab2.*
FROM tab2,
(SELECT tab1.col_1, MAX(tab2.col_2) AS max_col2
 FROM tab2, tab1
 WHERE tab1.id = tab2.id
 GROUP BY col_1) subquery1
WHERE subquery1.max_col2 = tab2.col_2;
```

Results:

```
+----+-------+-----------+
| id | col_1 | col_2     |
+----+-------+-----------+
| 1  | true  | 12789.123 |
| 3  | false | 24453.325 |
+----+-------+-----------+
```

Example: INSERT Query

```
INSERT OVERWRITE TABLE tab3
SELECT id, col_1, col_2, MONTH(col_3), DAYOFMONTH(col_3)
FROM tab1 WHERE YEAR(col_3) = 2012;
```

Query TAB3 to check the result:

```
SELECT * FROM tab3;
```

Results:

```
+----+-------+---------+-------+-----+
| id | col_1 | col_2   | month | day |
+----+-------+---------+-------+-----+
| 1  | true  | 123.123 | 10    | 24  |
| 2  | false | 1243.5  | 10    | 25  |
+----+-------+---------+-------+-----+
```

Advanced Tutorials

These tutorials walk you through advanced scenarios or specialized features.

Attaching an External Partitioned Table to an HDFS Directory Structure

This tutorial shows how you might set up a directory tree in HDFS, put data files into the lowest-level subdirectories, and then use an Impala external table to query the data files from their original locations.

The tutorial uses a table with web log data, with separate subdirectories for the year, month, day, and host. For simplicity, we use a tiny amount of CSV data, loading the same data into each partition.

First, we make an Impala partitioned table for CSV data, and look at the underlying HDFS directory structure to understand the directory structure to re-create elsewhere in HDFS. The columns `field1`, `field2`, and `field3` correspond to the contents of the CSV data files. The `year`, `month`, `day`, and `host` columns are all represented as subdirectories within the table structure, and are not part of the CSV files. We use `STRING` for each of these columns so that we can produce consistent subdirectory names, with leading zeros for a consistent length.

```
create database external_partitions;
use external_partitions;
create table logs (field1 string, field2 string, field3 string)
  partitioned by (year string, month string , day string, host string)
  row format delimited fields terminated by ',';
insert into logs partition (year="2013", month="07", day="28", host="host1") values
("foo","foo","foo");
insert into logs partition (year="2013", month="07", day="28", host="host2") values
("foo","foo","foo");
insert into logs partition (year="2013", month="07", day="29", host="host1") values
("foo","foo","foo");
insert into logs partition (year="2013", month="07", day="29", host="host2") values
("foo","foo","foo");
insert into logs partition (year="2013", month="08", day="01", host="host1") values
("foo","foo","foo");
```

Back in the Linux shell, we examine the HDFS directory structure. (Your Impala data directory might be in a different location; for historical reasons, it is sometimes under the HDFS path `/user/hive/warehouse`.) We use the `hdfs dfs -ls` command to examine the nested subdirectories corresponding to each partitioning column, with separate subdirectories at each level (with = in their names) representing the different values for each partitioning column. When we get to the lowest level of subdirectory, we use the `hdfs dfs -cat` command to examine the data file and see CSV-formatted data produced by the `INSERT` statement in Impala.

```
$ hdfs dfs -ls /user/impala/warehouse/external_partitions.db
Found 1 items
drwxrwxrwt   - impala hive          0 2013-08-07 12:24
/user/impala/warehouse/external_partitions.db/logs
$ hdfs dfs -ls /user/impala/warehouse/external_partitions.db/logs
Found 1 items
drwxr-xr-x   - impala hive          0 2013-08-07 12:24
/user/impala/warehouse/external_partitions.db/logs/year=2013
$ hdfs dfs -ls /user/impala/warehouse/external_partitions.db/logs/year=2013
```

```
Found 2 items
drwxr-xr-x   - impala hive          0 2013-08-07 12:23
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07
drwxr-xr-x   - impala hive          0 2013-08-07 12:24
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=08
$ hdfs dfs -ls /user/impala/warehouse/external_partitions.db/logs/year=2013/month=07
Found 2 items
drwxr-xr-x   - impala hive          0 2013-08-07 12:22
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28
drwxr-xr-x   - impala hive          0 2013-08-07 12:23
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=29
$ hdfs dfs -ls
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28
Found 2 items
drwxr-xr-x   - impala hive          0 2013-08-07 12:21
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=host1
drwxr-xr-x   - impala hive          0 2013-08-07 12:22
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=host2
$ hdfs dfs -ls
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=host1
Found 1 items
-rw-r--r--   3 impala hive         12 2013-08-07 12:21
/user/impala/warehouse/external_partiti
ons.db/logs/year=2013/month=07/day=28/host=host1/3981726974111751120--8907184999369517436_822630111_data.0
$ hdfs dfs -cat
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/\
host=host1/3981726974111751120--8 907184999369517436_822630111_data.0
foo,foo,foo
```

Still in the Linux shell, we use `hdfs dfs -mkdir` to create several data directories outside the HDFS directory tree that Impala controls (`/user/impala/warehouse` in this example, maybe different in your case). Depending on your configuration, you might need to log in as a user with permission to write into this HDFS directory tree; for example, the commands shown here were run while logged in as the `hdfs` user.

```
hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host1
hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host2
hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=28/host=host1
hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=07/day=29/host=host1
hdfs dfs -mkdir -p /user/impala/data/logs/year=2013/month=08/day=01/host=host1
```

We make a tiny CSV file, with values different than in the `INSERT` statements used earlier, and put a copy within each subdirectory that we will use as an Impala partition.

```
$ cat >dummy_log_data
bar,baz,bletch
hdfs dfs -mkdir -p
/user/impala/data/external_partitions/year=2013/month=08/day=01/host=host1
hdfs dfs -mkdir -p
/user/impala/data/external_partitions/year=2013/month=07/day=28/host=host1
hdfs dfs -mkdir -p
/user/impala/data/external_partitions/year=2013/month=07/day=28/host=host2
hdfs dfs -mkdir -p
/user/impala/data/external_partitions/year=2013/month=07/day=29/host=host1
hdfs dfs -put dummy_log_data /user/impala/data/logs/year=2013/month=07/day=28/host=host1
$ hdfs dfs -put dummy_log_data /user/impala/data/logs/year=2013/month=07/day=28/host=host2
hdfs dfs -put dummy_log_data /user/impala/data/logs/year=2013/month=07/day=29/host=host1
$ hdfs dfs -put dummy_log_data /user/impala/data/logs/year=2013/month=08/day=01/host=host1
```

Back in the `impala-shell` interpreter, we move the original Impala-managed table aside, and create a new *external* table with a `LOCATION` clause pointing to the directory under which we have set up all the partition subdirectories and data files.

```
use external_partitions;
alter table logs rename to logs_original;
create external table logs (field1 string, field2 string, field3 string)
  partitioned by (year string, month string, day string, host string)
```

```
  row format delimited fields terminated by ','
  location '/user/impala/data/logs';
```

Because partition subdirectories and data files come and go during the data lifecycle, you must identify each of the partitions through an ALTER TABLE statement before Impala recognizes the data files they contain.

```
alter table logs add partition (year="2013",month="07",day="28",host="host1")
alter table log_type add partition (year="2013",month="07",day="28",host="host2");
alter table log_type add partition (year="2013",month="07",day="29",host="host1");
alter table log_type add partition (year="2013",month="08",day="01",host="host1");
```

We issue a REFRESH statement for the table, always a safe practice when data files have been manually added, removed, or changed. Then the data is ready to be queried. The SELECT * statement illustrates that the data from our trivial CSV file was recognized in each of the partitions where we copied it. Although in this case there are only a few rows, we include a LIMIT clause on this test query just in case there is more data than we expect.

```
refresh log_type;
select * from log_type limit 100;
+--------+--------+--------+------+-------+-----+-------+
| field1 | field2 | field3 | year | month | day | host  |
+--------+--------+--------+------+-------+-----+-------+
| bar    | baz    | bletch | 2013 | 07    | 28  | host1 |
| bar    | baz    | bletch | 2013 | 08    | 01  | host1 |
| bar    | baz    | bletch | 2013 | 07    | 29  | host1 |
| bar    | baz    | bletch | 2013 | 07    | 28  | host2 |
+--------+--------+--------+------+-------+-----+-------+
```

Switching Back and Forth Between Impala and Hive

Sometimes, you might find it convenient to switch to the Hive shell to perform some data loading or transformation operation, particularly on file formats such as RCFile, SequenceFile, and Avro that Impala currently can query but not write to.

Whenever you create, drop, or alter a table or other kind of object through Hive, the next time you switch back to the impala-shell interpreter, issue a one-time INVALIDATE METADATA statement so that Impala recognizes the new or changed object.

Whenever you load, insert, or change data in an existing table through Hive (or even through manual HDFS operations such as the hdfs command), the next time you switch back to the impala-shell interpreter, issue a one-time REFRESH *table_name* statement so that Impala recognizes the new or changed data.

For examples showing how this process works for the REFRESH statement, look at the examples of creating RCFile and SequenceFile tables in Impala, loading data through Hive, and then querying the data through Impala. See Using the RCFile File Format with Impala Tables on page 665 and Using the SequenceFile File Format with Impala Tables on page 667 for those examples.

For examples showing how this process works for the INVALIDATE METADATA statement, look at the example of creating and loading an Avro table in Hive, and then querying the data through Impala. See Using the Avro File Format with Impala Tables on page 659 for that example.



Note:

Originally, Impala did not support UDFs, but this feature is available in Impala starting in Impala 1.2. Some INSERT ... SELECT transformations that you originally did through Hive can now be done through Impala. See User-Defined Functions (UDFs) on page 525 for details.

Prior to Impala 1.2, the REFRESH and INVALIDATE METADATA statements needed to be issued on each Impala node to which you connected and issued queries. In Impala 1.2 and higher, when you issue either of those statements on any Impala node, the results are broadcast to all the Impala nodes in the cluster, making it truly a one-step operation after each round of DDL or ETL operations in Hive.

Cross Joins and Cartesian Products with the CROSS JOIN Operator

Originally, Impala restricted join queries so that they had to include at least one equality comparison between the columns of the tables on each side of the join operator. With the huge tables typically processed by Impala, any miscoded query that produced a full Cartesian product as a result set could consume a huge amount of cluster resources.

In Impala 1.2.2 and higher, this restriction is lifted when you use the `CROSS JOIN` operator in the query. You still cannot remove all `WHERE` clauses from a query like `SELECT * FROM t1 JOIN t2` to produce all combinations of rows from both tables. But you can use the `CROSS JOIN` operator to explicitly request such a Cartesian product. Typically, this operation is applicable for smaller tables, where the result set still fits within the memory of a single Impala node.

The following example sets up data for use in a series of comic books where characters battle each other. At first, we use an equijoin query, which only allows characters from the same time period and the same planet to meet.

```
[localhost:21000] > create table heroes (name string, era string, planet string);
[localhost:21000] > create table villains (name string, era string, planet string);
[localhost:21000] > insert into heroes values
                  > ('Tesla','20th century','Earth'),
                  > ('Pythagoras','Antiquity','Earth'),
                  > ('Zopzar','Far Future','Mars');
Inserted 3 rows in 2.28s
[localhost:21000] > insert into villains values
                  > ('Caligula','Antiquity','Earth'),
                  > ('John Dillinger','20th century','Earth'),
                  > ('Xibulor','Far Future','Venus');
Inserted 3 rows in 1.93s
[localhost:21000] > select concat(heroes.name,' vs. ',villains.name) as battle
                  > from heroes join villains
                 > where heroes.era = villains.era and heroes.planet = villains.planet;
+--------------------------+
| battle                   |
+--------------------------+
| Tesla vs. John Dillinger |
| Pythagoras vs. Caligula  |
+--------------------------+
Returned 2 row(s) in 0.47s
```

Readers demanded more action, so we added elements of time travel and space travel so that any hero could face any villain. Prior to Impala 1.2.2, this type of query was impossible because all joins had to reference matching values between the two tables:

```
[localhost:21000] > -- Cartesian product not possible in Impala 1.1.
                  > select concat(heroes.name,' vs. ',villains.name) as battle from
heroes join villains;
ERROR: NotImplementedException: Join between 'heroes' and 'villains' requires at least
 one conjunctive equality predicate between the two tables
```

With Impala 1.2.2, we rewrite the query slightly to use `CROSS JOIN` rather than `JOIN`, and now the result set includes all combinations:

```
[localhost:21000] > -- Cartesian product available in Impala 1.2.2 with the CROSS JOIN
 syntax.
                  > select concat(heroes.name,' vs. ',villains.name) as battle from
heroes cross join villains;
+-----------------------------+
| battle                      |
+-----------------------------+
| Tesla vs. Caligula          |
| Tesla vs. John Dillinger    |
| Tesla vs. Xibulor           |
| Pythagoras vs. Caligula     |
| Pythagoras vs. John Dillinger |
| Pythagoras vs. Xibulor      |
| Zopzar vs. Caligula         |
| Zopzar vs. John Dillinger   |
| Zopzar vs. Xibulor          |
```

```
+-------------------------------+
Returned 9 row(s) in 0.33s
```

The full combination of rows from both tables is known as the Cartesian product. This type of result set is often used for creating grid data structures. You can also filter the result set by including WHERE clauses that do not explicitly compare columns between the two tables. The following example shows how you might produce a list of combinations of year and quarter for use in a chart, and then a shorter list with only selected quarters.

```
[localhost:21000] > create table x_axis (x int);
[localhost:21000] > create table y_axis (y int);
[localhost:21000] > insert into x_axis values (1),(2),(3),(4);
Inserted 4 rows in 2.14s
[localhost:21000] > insert into y_axis values (2010),(2011),(2012),(2013),(2014);
Inserted 5 rows in 1.32s
[localhost:21000] > select y as year, x as quarter from x_axis cross join y_axis;
+------+---------+
| year | quarter |
+------+---------+
| 2010 | 1       |
| 2011 | 1       |
| 2012 | 1       |
| 2013 | 1       |
| 2014 | 1       |
| 2010 | 2       |
| 2011 | 2       |
| 2012 | 2       |
| 2013 | 2       |
| 2014 | 2       |
| 2010 | 3       |
| 2011 | 3       |
| 2012 | 3       |
| 2013 | 3       |
| 2014 | 3       |
| 2010 | 4       |
| 2011 | 4       |
| 2012 | 4       |
| 2013 | 4       |
| 2014 | 4       |
+------+---------+
Returned 20 row(s) in 0.38s
[localhost:21000] > select y as year, x as quarter from x_axis cross join y_axis where
 x in (1,3);
+------+---------+
| year | quarter |
+------+---------+
| 2010 | 1       |
| 2011 | 1       |
| 2012 | 1       |
| 2013 | 1       |
| 2014 | 1       |
| 2010 | 3       |
| 2011 | 3       |
| 2012 | 3       |
| 2013 | 3       |
| 2014 | 3       |
+------+---------+
Returned 10 row(s) in 0.39s
```

Dealing with Parquet Files with Unknown Schema

As data pipelines start to include more aspects such as NoSQL or loosely specified schemas, you might encounter situations where you have data files (particularly in Parquet format) where you do not know the precise table definition. This tutorial shows how you can build an Impala table around data that comes from non-Impala or even non-SQL sources, where you do not have control of the table layout and might not be familiar with the characteristics of the data.

The data used in this tutorial represents airline on-time arrival statistics, from October 1987 through April 2008. See the details on the 2009 ASA Data Expo web site. You can also see the explanations of the columns; for purposes of this exercise, wait until after following the tutorial before examining the schema, to better simulate a real-life situation where you cannot rely on assumptions and assertions about the ranges and representations of data values.

Download the Data Files into HDFS

First, we download and unpack the data files. There are 8 files totalling 1.4 GB.

```
$ wget -O airlines_parquet.tar.gz https://home.apache.org/~arodoni/airlines_parquet.tar.gz
$ wget https://home.apache.org/~arodoni/airlines_parquet.tar.gz.sha512
$ shasum -a 512 -c airlines_parquet.tar.gz.sha512
airlines_parquet.tar.gz: OK

$ tar xvzf airlines_parquet.tar.gz

$ cd airlines_parquet/

$ du -kch *.parq
253M   4345e5eef217aa1b-c8f16177f35fd983_1150363067_data.0.parq
14M    4345e5eef217aa1b-c8f16177f35fd983_1150363067_data.1.parq
253M   4345e5eef217aa1b-c8f16177f35fd984_501176748_data.0.parq
64M    4345e5eef217aa1b-c8f16177f35fd984_501176748_data.1.parq
184M   4345e5eef217aa1b-c8f16177f35fd985_1199995767_data.0.parq
241M   4345e5eef217aa1b-c8f16177f35fd986_2086627597_data.0.parq
212M   4345e5eef217aa1b-c8f16177f35fd987_1048668565_data.0.parq
152M   4345e5eef217aa1b-c8f16177f35fd988_1432111844_data.0.parq
1.4G   total
```

Next, we put the Parquet data files in HDFS, all together in a single directory, with permissions on the directory and the files so that the `impala` user will be able to read them.

After unpacking, we saw the largest Parquet file was 253 MB. When copying Parquet files into HDFS for Impala to use, for maximum query performance, make sure that each file resides in a single HDFS data block. Therefore, we pick a size larger than any single file and specify that as the block size, using the argument `-Ddfs.block.size=253m` on the `hdfs dfs -put` command.

```
$ sudo -u hdfs hdfs dfs -mkdir -p /user/impala/staging/airlines
$ sudo -u hdfs hdfs dfs -Ddfs.block.size=253m -put *.parq /user/impala/staging/airlines
$ sudo -u hdfs hdfs dfs -ls /user/impala/staging
Found 1 items

$ sudo -u hdfs hdfs dfs -ls /user/impala/staging/airlines
Found 8 items
```

Create Database and Tables

With the files in an accessible location in HDFS, you create a database table that uses the data in those files:

- The `CREATE EXTERNAL` syntax and the `LOCATION` attribute point Impala at the appropriate HDFS directory.
- The `LIKE PARQUET '`*`path_to_any_parquet_file`*`'` clause means we skip the list of column names and types; Impala automatically gets the column names and data types straight from the data files. (Currently, this technique only works for Parquet files.)
- Ignore the warning about lack of `READ_WRITE` access to the files in HDFS; the `impala` user can read the files, which will be sufficient for us to experiment with queries and perform some copy and transform operations into other tables.

```
$ impala-shell
> CREATE DATABASE airlines_data;
  USE airlines_data;
  CREATE EXTERNAL TABLE airlines_external
  LIKE PARQUET
'hdfs:staging/airlines/4345e5eef217aa1b-c8f16177f35fd983_1150363067_data.0.parq'
  STORED AS PARQUET LOCATION 'hdfs:staging/airlines';
```

```
WARNINGS: Impala does not have READ_WRITE access to path
'hdfs://myhost.com:8020/user/impala/staging'
```

Examine Physical and Logical Schema

With the table created, we examine its physical and logical characteristics to confirm that the data is really there and in a format and shape that we can work with.

- The `SHOW TABLE STATS` statement gives a very high-level summary of the table, showing how many files and how much total data it contains. Also, it confirms that the table is expecting all the associated data files to be in Parquet format. (The ability to work with all kinds of HDFS data files in different formats means that it is possible to have a mismatch between the format of the data files, and the format that the table expects the data files to be in.)
- The `SHOW FILES` statement confirms that the data in the table has the expected number, names, and sizes of the original Parquet files.
- The `DESCRIBE` statement (or its abbreviation `DESC`) confirms the names and types of the columns that Impala automatically created after reading that metadata from the Parquet file.
- The `DESCRIBE FORMATTED` statement prints out some extra detail along with the column definitions. The pieces we care about for this exercise are:
  - The containing database for the table.
  - The location of the associated data files in HDFS.
  - The table is an external table so Impala will not delete the HDFS files when we finish the experiments and drop the table.
  - The table is set up to work exclusively with files in the Parquet format.

```
> SHOW TABLE STATS airlines_external;
+-------+--------+--------+--------------+-------------------+---------+-------------------+
| #Rows | #Files | Size   | Bytes Cached | Cache Replication | Format  | Incremental
stats |
+-------+--------+--------+--------------+-------------------+---------+-------------------+
| -1    | 8      | 1.34GB | NOT CACHED   | NOT CACHED        | PARQUET | false
     |
+-------+--------+--------+--------------+-------------------+---------+-------------------+

> SHOW FILES IN airlines_external;
+-----------------------------------------------------------------------------------------+---------+-----------+
| path
   | size     | partition |
+-----------------------------------------------------------------------------------------+---------+-----------+
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd983_1150363067_data.0.parq
 | 252.99MB |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd983_1150363067_data.1.parq
 | 13.43MB  |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd984_501176748_data.0.parq
  | 252.84MB |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd984_501176748_data.1.parq
  | 63.92MB  |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd985_1199995767_data.0.parq
 | 183.64MB |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd986_2086627597_data.0.parq
 | 240.04MB |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd987_1048668565_data.0.parq
 | 211.35MB |           |
| /user/impala/staging/airlines/4345e5eef217aa1b-c8f16177f35fd988_1432111844_data.0.parq
 | 151.46MB |           |
+-----------------------------------------------------------------------------------------+---------+-----------+

> DESCRIBE airlines_external;
+---------------------+--------+-----------------------------------------------------+
| name                | type   | comment                                             |
+---------------------+--------+-----------------------------------------------------+
| year                | int    | Inferred from Parquet file.                         |
| month               | int    | Inferred from Parquet file.                         |
| day                 | int    | Inferred from Parquet file.                         |
```

```
| dayofweek           | int    | Inferred from Parquet file.                        |
| dep_time            | int    | Inferred from Parquet file.                         |
| crs_dep_time        | int    | Inferred from Parquet file.                        |
| arr_time            | int    | Inferred from Parquet file.                        |
| crs_arr_time        | int    | Inferred from Parquet file.                        |
| carrier             | string | Inferred from Parquet file.                        |
| flight_num          | int    | Inferred from Parquet file.                        |
| tail_num            | int    | Inferred from Parquet file.                        |
| actual_elapsed_time | int    | Inferred from Parquet file.                        |
| crs_elapsed_time    | int    | Inferred from Parquet file.                        |
| airtime             | int    | Inferred from Parquet file.                        |
| arrdelay            | int    | Inferred from Parquet file.                        |
| depdelay            | int    | Inferred from Parquet file.                        |
| origin              | string | Inferred from Parquet file.                        |
| dest                | string | Inferred from Parquet file.                        |
| distance            | int    | Inferred from Parquet file.                        |
| taxi_in             | int    | Inferred from Parquet file.                        |
| taxi_out            | int    | Inferred from Parquet file.                        |
| cancelled           | int    | Inferred from Parquet file.                        |
| cancellation_code   | string | Inferred from Parquet file.                        |
| diverted            | int    | Inferred from Parquet file.                        |
| carrier_delay       | int    | Inferred from Parquet file.                        |
| weather_delay       | int    | Inferred from Parquet file.                        |
| nas_delay           | int    | Inferred from Parquet file.                        |
| security_delay      | int    | Inferred from Parquet file.                        |
| late_aircraft_delay | int    | Inferred from Parquet file.                        |
+---------------------+--------+----------------------------------------------------+

> DESCRIBE FORMATTED airlines_external;
+------------------------------+------------------------------
| name                         | type
+------------------------------+------------------------------
...
| # Detailed Table Information | NULL
| Database:                    | airlines_data
| Owner:                       | impala
...
| Location:                    | /user/impala/staging/airlines
| Table Type:                  | EXTERNAL_TABLE
...
| # Storage Information        | NULL
| SerDe Library:               |
org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe
| InputFormat:                 |
org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputForma
| OutputFormat:                |
org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat
...
```

Analyze Data

Now that we are confident that the connections are solid between the Impala table and the underlying Parquet files, we run some initial queries to understand the characteristics of the data: the overall number of rows, and the ranges and how many different values are in certain columns.

```
> SELECT COUNT(*) FROM airlines_external;
+-----------+
| count(*)  |
+-----------+
| 123534969 |
+-----------+
```

The `NDV( )` function returns a number of distinct values, which, for performance reasons, is an estimate when there are lots of different values in the column, but is precise when the cardinality is less than 16 K. Use `NDV( )` function for this kind of exploration rather than `COUNT(DISTINCT` *`colname`*`)`, because Impala can evaluate multiple `NDV( )` functions in a single query, but only a single instance of `COUNT DISTINCT`.

```
> SElECT NDV(carrier), NDV(flight_num), NDV(tail_num),
  NDV(origin), NDV(dest) FROM airlines_external;
```

```
+--------------+-----------------+---------------+-------------+-----------+
| ndv(carrier) | ndv(flight_num) | ndv(tail_num) | ndv(origin) | ndv(dest) |
+--------------+-----------------+---------------+-------------+-----------+
| 29           | 8463            | 3             | 342         | 349       |
+--------------+-----------------+---------------+-------------+-----------+

> SELECT tail_num, COUNT(*) AS howmany FROM airlines_external
  GROUP BY tail_num;
+----------+-----------+
| tail_num | howmany   |
+----------+-----------+
| NULL     | 123122001 |
| 715      | 1         |
| 0        | 406405    |
| 112      | 6562      |
+----------+-----------+

> SELECT DISTINCT dest FROM airlines_external
  WHERE dest NOT IN (SELECT origin FROM airlines_external);
+------+
| dest |
+------+
| CBM  |
| SKA  |
| LAR  |
| RCA  |
| LBF  |
+------+

> SELECT DISTINCT dest FROM airlines_external
  WHERE dest NOT IN (SELECT DISTINCT origin FROM airlines_external);
+------+
| dest |
+------+
| CBM  |
| SKA  |
| LAR  |
| RCA  |
| LBF  |
+------+

> SELECT DISTINCT origin FROM airlines_external
  WHERE origin NOT IN (SELECT DISTINCT dest FROM airlines_external);
Fetched 0 row(s) in 2.63
```

With the above queries, we see that there are modest numbers of different airlines, flight numbers, and origin and destination airports. Two things jump out from this query: the number of `tail_num` values is much smaller than we might have expected, and there are more destination airports than origin airports. Let's dig further. What we find is that most `tail_num` values are `NULL`. It looks like this was an experimental column that wasn't filled in accurately. We make a mental note that if we use this data as a starting point, we'll ignore this column. We also find that certain airports are represented in the `ORIGIN` column but not the `DEST` column; now we know that we cannot rely on the assumption that those sets of airport codes are identical.



Note: The first `SELECT DISTINCT DEST` query takes almost 40 seconds. We expect all queries on such a small data set, less than 2 GB, to take a few seconds at most. The reason is because the expression `NOT IN (SELECT origin FROM airlines_external)` produces an intermediate result set of 123 million rows, then runs 123 million comparisons on each data node against the tiny set of destination airports. The way the `NOT IN` operator works internally means that this intermediate result set with 123 million rows might be transmitted across the network to each data node in the cluster. Applying another `DISTINCT` inside the `NOT IN` subquery means that the intermediate result set is only 340 items, resulting in much less network traffic and fewer comparison operations. The more efficient query with the added `DISTINCT` is approximately 7 times as fast.

Next, we try doing a simple calculation, with results broken down by year. This reveals that some years have no data in the `airtime` column. That means we might be able to use that column in queries involving certain date ranges, but we cannot count on it to always be reliable. The question of whether a column contains any `NULL` values, and if so

what is their number, proportion, and distribution, comes up again and again when doing initial exploration of a data set.

```
> SELECT year, SUM(airtime) FROM airlines_external
  GROUP BY year ORDER BY year DESC;
+------+--------------+
| year | sum(airtime) |
+------+--------------+
| 2008 | 713050445    |
| 2007 | 748015545    |
| 2006 | 720372850    |
| 2005 | 708204026    |
| 2004 | 714276973    |
| 2003 | 665706940    |
| 2002 | 549761849    |
| 2001 | 590867745    |
| 2000 | 583537683    |
| 1999 | 561219227    |
| 1998 | 538050663    |
| 1997 | 536991229    |
| 1996 | 519440044    |
| 1995 | 513364265    |
| 1994 | NULL         |
| 1993 | NULL         |
| 1992 | NULL         |
| 1991 | NULL         |
| 1990 | NULL         |
| 1989 | NULL         |
| 1988 | NULL         |
| 1987 | NULL         |
+------+--------------+
```

With the notion of `NULL` values in mind, let's come back to the `tail_num` column that we discovered had a lot of `NULL`s. Let's quantify the `NULL` and non-`NULL` values in that column for better understanding. First, we just count the overall number of rows versus the non-`NULL` values in that column. That initial result gives the appearance of relatively few non-`NULL` values, but we can break it down more clearly in a single query. Once we have the `COUNT(*)` and the `COUNT(`*`colname`*`)` numbers, we can encode that initial query in a `WITH` clause, then run a follow-on query that performs multiple arithmetic operations on those values. Seeing that only one-third of one percent of all rows have non-`NULL` values for the `tail_num` column clearly illustrates that column is not of much use.

```
> SELECT COUNT(*) AS 'rows', COUNT(tail_num) AS 'non-null tail numbers'
  FROM airlines_external;
+-----------+-----------------------+
| rows      | non-null tail numbers |
+-----------+-----------------------+
| 123534969 | 412968                |
+-----------+-----------------------+

> WITH t1 AS
  (SELECT COUNT(*) AS 'rows', COUNT(tail_num) AS 'nonnull'
  FROM airlines_external)
SELECT `rows`, `nonnull`, `rows` - `nonnull` AS 'nulls',
  (`nonnull` / `rows`) * 100 AS 'percentage non-null'
FROM t1;
+-----------+---------+-----------+---------------------+
| rows      | nonnull | nulls     | percentage non-null |
+-----------+---------+-----------+---------------------+
| 123534969 | 412968  | 123122001 | 0.3342923897119365  |
+-----------+---------+-----------+---------------------+
```

By examining other columns using these techniques, we can form a mental picture of the way data is distributed throughout the table, and which columns are most significant for query purposes. For this tutorial, we focus mostly on the fields likely to hold discrete values, rather than columns such as `actual_elapsed_time` whose names suggest they hold measurements. We would dig deeper into those columns once we had a clear picture of which questions were worthwhile to ask, and what kinds of trends we might look for. For the final piece of initial exploration, let's look

at the year column. A simple GROUP BY query shows that it has a well-defined range, a manageable number of distinct values, and relatively even distribution of rows across the different years.

```
> SELECT MIN(year), MAX(year), NDV(year) FROM airlines_external;
+-----------+-----------+-----------+
| min(year) | max(year) | ndv(year) |
+-----------+-----------+-----------+
| 1987      | 2008      | 22        |
+-----------+-----------+-----------+

> SELECT year, COUNT(*) howmany FROM airlines_external
  GROUP BY year ORDER BY year DESC;
+------+---------+
| year | howmany |
+------+---------+
| 2008 | 7009728 |
| 2007 | 7453215 |
| 2006 | 7141922 |
| 2005 | 7140596 |
| 2004 | 7129270 |
| 2003 | 6488540 |
| 2002 | 5271359 |
| 2001 | 5967780 |
| 2000 | 5683047 |
| 1999 | 5527884 |
| 1998 | 5384721 |
| 1997 | 5411843 |
| 1996 | 5351983 |
| 1995 | 5327435 |
| 1994 | 5180048 |
| 1993 | 5070501 |
| 1992 | 5092157 |
| 1991 | 5076925 |
| 1990 | 5270893 |
| 1989 | 5041200 |
| 1988 | 5202096 |
| 1987 | 1311826 |
+------+---------+
```

We could go quite far with the data in this initial raw format, just as we downloaded it from the web. If the data set proved to be useful and worth persisting in Impala for extensive queries, we might want to copy it to an internal table, letting Impala manage the data files and perhaps reorganizing a little for higher efficiency. In this next stage of the tutorial, we copy the original data into a partitioned table, still in Parquet format. Partitioning based on the year column lets us run queries with clauses such as WHERE year = 2001 or WHERE year BETWEEN 1989 AND 1999, which can dramatically cut down on I/O by ignoring all the data from years outside the desired range. Rather than reading all the data and then deciding which rows are in the matching years, Impala can zero in on only the data files from specific year partitions. To do this, Impala physically reorganizes the data files, putting the rows from each year into data files in a separate HDFS directory for each year value. Along the way, we'll also get rid of the tail_num column that proved to be almost entirely NULL.

The first step is to create a new table with a layout very similar to the original airlines_external table. We'll do that by reverse-engineering a CREATE TABLE statement for the first table, then tweaking it slightly to include a PARTITION BY clause for year, and excluding the tail_num column. The SHOW CREATE TABLE statement gives us the starting point.

Although we could edit that output into a new SQL statement, all the ASCII box characters make such editing inconvenient. To get a more stripped-down CREATE TABLE to start with, we restart the impala-shell command with the -B option, which turns off the box-drawing behavior.

```
$ impala-shell -i localhost -B -d airlines_data;

> SHOW CREATE TABLE airlines_external;
"CREATE EXTERNAL TABLE airlines_data.airlines_external (
  year INT COMMENT 'inferred from: optional int32 year',
  month INT COMMENT 'inferred from: optional int32 month',
  day INT COMMENT 'inferred from: optional int32 day',
  dayofweek INT COMMENT 'inferred from: optional int32 dayofweek',
```

```
  dep_time INT COMMENT 'inferred from: optional int32 dep_time',
  crs_dep_time INT COMMENT 'inferred from: optional int32 crs_dep_time',
  arr_time INT COMMENT 'inferred from: optional int32 arr_time',
  crs_arr_time INT COMMENT 'inferred from: optional int32 crs_arr_time',
  carrier STRING COMMENT 'inferred from: optional binary carrier',
  flight_num INT COMMENT 'inferred from: optional int32 flight_num',
  tail_num INT COMMENT 'inferred from: optional int32 tail_num',
  actual_elapsed_time INT COMMENT 'inferred from: optional int32 actual_elapsed_time',
  crs_elapsed_time INT COMMENT 'inferred from: optional int32 crs_elapsed_time',
  airtime INT COMMENT 'inferred from: optional int32 airtime',
  arrdelay INT COMMENT 'inferred from: optional int32 arrdelay',
  depdelay INT COMMENT 'inferred from: optional int32 depdelay',
  origin STRING COMMENT 'inferred from: optional binary origin',
  dest STRING COMMENT 'inferred from: optional binary dest',
  distance INT COMMENT 'inferred from: optional int32 distance',
  taxi_in INT COMMENT 'inferred from: optional int32 taxi_in',
  taxi_out INT COMMENT 'inferred from: optional int32 taxi_out',
  cancelled INT COMMENT 'inferred from: optional int32 cancelled',
  cancellation_code STRING COMMENT 'inferred from: optional binary cancellation_code',
  diverted INT COMMENT 'inferred from: optional int32 diverted',
  carrier_delay INT COMMENT 'inferred from: optional int32 carrier_delay',
  weather_delay INT COMMENT 'inferred from: optional int32 weather_delay',
  nas_delay INT COMMENT 'inferred from: optional int32 nas_delay',
  security_delay INT COMMENT 'inferred from: optional int32 security_delay',
  late_aircraft_delay INT COMMENT 'inferred from: optional int32 late_aircraft_delay'
)
STORED AS PARQUET
LOCATION 'hdfs://a1730.example.com:8020/user/impala/staging/airlines'
TBLPROPERTIES ('numFiles'='0', 'COLUMN_STATS_ACCURATE'='false',
  'transient_lastDdlTime'='1439425228', 'numRows'='-1', 'totalSize'='0',
  'rawDataSize'='-1')"
```

After copying and pasting the `CREATE TABLE` statement into a text editor for fine-tuning, we quit and restart `impala-shell` without the `-B` option, to switch back to regular output.

Next we run the `CREATE TABLE` statement that we adapted from the `SHOW CREATE TABLE` output. We kept the `STORED AS PARQUET` clause because we want to rearrange the data somewhat but still keep it in the high-performance Parquet format. The `LOCATION` and `TBLPROPERTIES` clauses are not relevant for this new table, so we edit those out. Because we are going to partition the new table based on the `year` column, we move that column name (and its type) into a new `PARTITIONED BY` clause.

```
> CREATE TABLE airlines_data.airlines
 (month INT,
  day INT,
  dayofweek INT,
  dep_time INT,
  crs_dep_time INT,
  arr_time INT,
  crs_arr_time INT,
  carrier STRING,
  flight_num INT,
  actual_elapsed_time INT,
  crs_elapsed_time INT,
  airtime INT,
  arrdelay INT,
  depdelay INT,
  origin STRING,
  dest STRING,
  distance INT,
  taxi_in INT,
  taxi_out INT,
  cancelled INT,
  cancellation_code STRING,
  diverted INT,
  carrier_delay INT,
  weather_delay INT,
  nas_delay INT,
  security_delay INT,
  late_aircraft_delay INT)
PARTITIONED BY (year INT)
```

```
STORED AS PARQUET
;
```

Next, we copy all the rows from the original table into this new one with an INSERT statement. (We edited the CREATE TABLE statement to make an INSERT statement with the column names in the same order.) The only change is to add a PARTITION(year) clause, and move the year column to the very end of the SELECT list of the INSERT statement. Specifying PARTITION(year), rather than a fixed value such as PARTITION(year=2000), means that Impala figures out the partition value for each row based on the value of the very last column in the SELECT list. This is the first SQL statement that legitimately takes any substantial time, because the rows from different years are shuffled around the cluster; the rows that go into each partition are collected on one node, before being written to one or more new data files.

```
> INSERT INTO airlines_data.airlines
  PARTITION (year)
  SELECT
    month,
    day,
    dayofweek,
    dep_time,
    crs_dep_time,
    arr_time,
    crs_arr_time,
    carrier,
    flight_num,
    actual_elapsed_time,
    crs_elapsed_time,
    airtime,
    arrdelay,
    depdelay,
    origin,
    dest,
    distance,
    taxi_in,
    taxi_out,
    cancelled,
    cancellation_code,
    diverted,
    carrier_delay,
    weather_delay,
    nas_delay,
    security_delay,
    late_aircraft_delay,
    year
  FROM airlines_data.airlines_external;
```

Once partitioning or join queries come into play, it's important to have statistics that Impala can use to optimize queries on the corresponding tables. The COMPUTE INCREMENTAL STATS statement is the way to collect statistics for partitioned tables. Then the SHOW TABLE STATS statement confirms that the statistics are in place for each partition, and also illustrates how many files and how much raw data is in each partition.

```
> COMPUTE INCREMENTAL STATS airlines;
+-------------------------------------------+
| summary                                   |
+-------------------------------------------+
| Updated 22 partition(s) and 27 column(s). |
+-------------------------------------------+

> SHOW TABLE STATS airlines;
+-------+---------+-------+--------+-------------+-------------------+---------+
| year  | #Rows   | #Files | Size   | Bytes Cached | Cache Replication | Format  |
 Incremental stats | Location
                                |
+-------+---------+-------+--------+-------------+-------------------+---------+
| 1987  | 1311826  | 1      | 11.75MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1987 |
| 1988  | 5202096  | 1      | 44.04MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
```

```
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1988 |
| 1989  | 5041200   | 1      | 46.07MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1989 |
| 1990  | 5270893   | 1      | 46.25MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1990 |
| 1991  | 5076925   | 1      | 46.77MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1991 |
| 1992  | 5092157   | 1      | 48.21MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1992 |
| 1993  | 5070501   | 1      | 47.46MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1993 |
| 1994  | 5180048   | 1      | 47.47MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1994 |
| 1995  | 5327435   | 1      | 62.40MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1995 |
| 1996  | 5351983   | 1      | 62.93MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1996 |
| 1997  | 5411843   | 1      | 65.05MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1997 |
| 1998  | 5384721   | 1      | 62.21MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1998 |
| 1999  | 5527884   | 1      | 65.10MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=1999 |
| 2000  | 5683047   | 1      | 67.68MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2000 |
| 2001  | 5967780   | 1      | 74.03MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2001 |
| 2002  | 5271359   | 1      | 74.00MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2002 |
| 2003  | 6488540   | 1      | 99.35MB  | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2003 |
| 2004  | 7129270   | 1      | 123.29MB | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2004 |
| 2005  | 7140596   | 1      | 120.72MB | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2005 |
| 2006  | 7141922   | 1      | 121.88MB | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2006 |
| 2007  | 7453215   | 1      | 130.87MB | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2007 |
| 2008  | 7009728   | 1      | 123.14MB | NOT CACHED   | NOT CACHED        | PARQUET |
 true              |
hdfs://myhost.com:8020/user/hive/warehouse/airline_data.db/airlines/year=2008 |
| Total | 123534969 | 22     | 1.55GB   | 0B           |                   |         |
                  |
                                          |
+---+-------+---+------+---------+-------------+---+-------+-----------------------------------------------------+
```

At this point, we sanity check the partitioning we did. All the partitions have exactly one file, which is on the low side. A query that includes a clause `WHERE year=2004` will only read a single data block; that data block will be read and processed by a single data node; therefore, for a query targeting a single year, all the other nodes in the cluster will sit idle while all the work happens on a single machine. It's even possible that by chance (depending on HDFS replication factor and the way data blocks are distributed across the cluster), that multiple year partitions selected by a filter such

as `WHERE year BETWEEN 1999 AND 2001` could all be read and processed by the same data node. The more data files each partition has, the more parallelism you can get and the less probability of "hotspots" occurring on particular nodes, therefore a bigger performance boost by having a big cluster.

However, the more data files, the less data goes in each one. The overhead of dividing the work in a parallel query might not be worth it if each node is only reading a few megabytes. 50 or 100 megabytes is a decent size for a Parquet data block; 9 or 37 megabytes is on the small side. Which is to say, the data distribution we ended up with based on this partitioning scheme is on the borderline between sensible (reasonably large files) and suboptimal (few files in each partition). The way to see how well it works in practice is to run the same queries against the original flat table and the new partitioned table, and compare times.

Spoiler: in this case, with my particular 4-node cluster with its specific distribution of data blocks and my particular exploratory queries, queries against the partitioned table do consistently run faster than the same queries against the unpartitioned table. But I could not be sure that would be the case without some real measurements. Here are some queries I ran to draw that conclusion, first against `airlines_external` (no partitioning), then against `AIRLINES` (partitioned by year). The `AIRLINES` queries are consistently faster. Changing the volume of data, changing the size of the cluster, running queries that did or didn't refer to the partition key columns, or other factors could change the results to favor one table layout or the other.



Note: If you find the volume of each partition is only in the low tens of megabytes, consider lowering the granularity of partitioning. For example, instead of partitioning by year, month, and day, partition by year and month or even just by year. The ideal layout to distribute work efficiently in a parallel query is many tens or even hundreds of megabytes per Parquet file, and the number of Parquet files in each partition somewhat higher than the number of data nodes.

```
> SELECT SUM(airtime) FROM airlines_external;
+-------------+
| 8662859484  |
+-------------+

> SELECT SUM(airtime) FROM airlines;
+-------------+
| 8662859484  |
+-------------+

> SELECT SUM(airtime) FROM airlines_external WHERE year = 2005;
+-------------+
| 708204026   |
+-------------+

> SELECT SUM(airtime) FROM airlines WHERE year = 2005;
+-------------+
| 708204026   |
+-------------+
```

Now we can finally analyze this data set that from the raw data files and we didn't know what columns they contained. Let's see whether the `airtime` of a flight tends to be different depending on the day of the week. We can see that the average is a little higher on day number 6; perhaps Saturday is a busy flying day and planes have to circle for longer at the destination airport before landing.

```
> SELECT dayofweek, AVG(airtime) FROM airlines
  GROUP BY dayofweek ORDER BY dayofweek;
+-----------+-------------------+
| dayofweek | avg(airtime)      |
+-----------+-------------------+
| 1         | 102.1560425016671 |
| 2         | 102.1582931538807 |
| 3         | 102.2170009256653 |
| 4         | 102.37477661846   |
| 5         | 102.2697358763511 |
| 6         | 105.3627448363705 |
| 7         | 103.4144351202054 |
+-----------+-------------------+
```

To see if the apparent trend holds up over time, let's do the same breakdown by day of week, but also split up by year. Now we can see that day number 6 consistently has a higher average air time in each year. We can also see that the average air time increased over time across the board. And the presence of NULL for this column in years 1987 to 1994 shows that queries involving this column need to be restricted to a date range of 1995 and higher.

```
> SELECT year, dayofweek, AVG(airtime) FROM airlines
  GROUP BY year, dayofweek ORDER BY year DESC, dayofweek;
+------+-----------+-------------------+
| year | dayofweek | avg(airtime)      |
+------+-----------+-------------------+
| 2008 | 1         | 103.1821651651355 |
| 2008 | 2         | 103.2149301386094 |
| 2008 | 3         | 103.0585076622796 |
| 2008 | 4         | 103.4671383539038 |
| 2008 | 5         | 103.5575385182659 |
| 2008 | 6         | 107.4006306562128 |
| 2008 | 7         | 104.8648851041755 |
| 2007 | 1         | 102.2196114337825 |
| 2007 | 2         | 101.9317791906348 |
| 2007 | 3         | 102.0964767689043 |
| 2007 | 4         | 102.6215927201686 |
| 2007 | 5         | 102.4289399000661 |
| 2007 | 6         | 105.1477448215756 |
| 2007 | 7         | 103.6305945644095 |
...
| 1996 | 1         | 99.33860750862108 |
| 1996 | 2         | 99.54225446396656 |
| 1996 | 3         | 99.41129336113134 |
| 1996 | 4         | 99.5110373340348  |
| 1996 | 5         | 99.22120745027595 |
| 1996 | 6         | 101.1717447111921 |
| 1996 | 7         | 99.95410136133704 |
| 1995 | 1         | 96.93779698300494 |
| 1995 | 2         | 96.93458674589712 |
| 1995 | 3         | 97.00972311337051 |
| 1995 | 4         | 96.90843832024412 |
| 1995 | 5         | 96.78382115425562 |
| 1995 | 6         | 98.70872826057003 |
| 1995 | 7         | 97.85570478374616 |
| 1994 | 1         | NULL              |
| 1994 | 2         | NULL              |
| 1994 | 3         | NULL              |
...
| 1987 | 5         | NULL              |
| 1987 | 6         | NULL              |
| 1987 | 7         | NULL              |
+------+-----------+-------------------+
```

Impala Administration

As an administrator, you monitor Impala's use of resources and take action when necessary to keep Impala running smoothly and avoid conflicts with other Hadoop components running on the same cluster. When you detect that an issue has happened or could happen in the future, you reconfigure Impala or other components such as HDFS or even the hardware of the cluster itself to resolve or avoid problems.

Related tasks:

As an administrator, you can expect to perform installation, upgrade, and configuration tasks for Impala on all machines in a cluster. See Impala Upgrade Considerations on page 38, and Managing Impala on page 36 for details.

For security tasks typically performed by administrators, see Impala Security on page 82.

Administrators also decide how to allocate cluster resources so that all Hadoop components can run smoothly together. For Impala, this task primarily involves:

- Deciding how many Impala queries can run concurrently and with how much memory, through the admission control feature. See Admission Control and Query Queuing on page 549 for details.
- Dividing cluster resources such as memory between Impala and other components, using YARN for overall resource management, and Llama to mediate resource requests from Impala to YARN. See Resource Management on page 549 for details.

Setting Timeout Periods for Daemons, Queries, and Sessions

Depending on how busy your CDH cluster is, you might increase or decrease various timeout values. Increase timeouts if Impala is cancelling operations prematurely, when the system is responding slower than usual but the operations are still successful if given extra time. Decrease timeouts if operations are idle or hanging for long periods, and the idle or hung operations are consuming resources and reducing concurrency.

Increasing the Statestore Timeout

If you have an extensive Impala schema, for example with hundreds of databases, tens of thousands of tables, and so on, you might encounter timeout errors during startup as the Impala catalog service broadcasts metadata to all the Impala nodes using the statestore service. To avoid such timeout errors on startup, increase the statestore timeout value from its default of 10 seconds. Specify the timeout value using the `-statestore_subscriber_timeout_seconds` option for the statestore service, using the configuration instructions in Modifying Impala Startup Options. The symptom of this problem is messages in the `impalad` log such as:

```
Connection with state-store lost
Trying to re-register with state-store
```

See Scalability Considerations for the Impala Statestore on page 606 for more details about statestore operation and settings on clusters with a large number of Impala-related objects such as tables and partitions.

Setting the Idle Query and Idle Session Timeouts for impalad

To keep long-running queries or idle sessions from tying up cluster resources, you can set timeout intervals for both individual queries, and entire sessions.



Note:

The timeout clock for queries and sessions only starts ticking when the query or session is idle.

For queries, this means the query has results ready but is waiting for a client to fetch the data. A query can run for an arbitrary time without triggering a timeout, because the query is computing results rather than sitting idle waiting for the results to be fetched. The timeout period is intended to prevent unclosed queries from consuming resources and taking up slots in the admission count of running queries, potentially preventing other queries from starting.

For sessions, this means that no query has been submitted for some period of time.

Use the following startup options for the `impalad` daemon to specify timeout values:

- `--idle_query_timeout`

  Specifies the time in seconds after which an idle query is cancelled. This could be a query whose results were all fetched but was never closed, or one whose results were partially fetched and then the client program stopped requesting further results. This condition is most likely to occur in a client program using the JDBC or ODBC interfaces, rather than in the interactive `impala-shell` interpreter. Once a query is cancelled, the client program cannot retrieve any further results from the query.

  You can reduce the idle query timeout by using the `QUERY_TIMEOUT_S` query option. Any non-zero value specified for the `--idle_query_timeout` startup option serves as an upper limit for the `QUERY_TIMEOUT_S` query option. See QUERY_TIMEOUT_S Query Option (CDH 5.2 or higher only) on page 355 about the query option.

  A zero value for `--idle_query_timeout` disables query timeouts.

  Cancelled queries remain in the open state but use only the minimal resources.

- `--idle_session_timeout`

  Specifies the time in seconds after which an idle session expires. A session is idle when no activity is occurring for any of the queries in that session, and the session has not started any new queries. Once a session is expired, you cannot issue any new query requests to it. The session remains open, but the only operation you can perform is to close it.

  The default value of 0 specifies sessions never expire.

  You can override the `--idle_session_timeout` value with the IDLE_SESSION_TIMEOUT Query Option (CDH 5.15 / Impala 2.12 or higher only) on page 335 at the session level.

For instructions on changing `impalad` startup options, see Modifying Impala Startup Options.



Note:

Impala checks periodically for idle sessions and queries to cancel. The actual idle time before cancellation might be up to 50% greater than the specified configuration setting. For example, if the timeout setting was 60, the session or query might be cancelled after being idle between 60 and 90 seconds.

Setting Timeout and Retries for Thrift Connections to the Backend Client

Impala connections to the backend client are subject to failure in cases when the network is momentarily overloaded. To avoid failed queries due to transient network problems, you can configure the number of Thrift connection retries using the following option:

- The `--backend_client_connection_num_retries` option specifies the number of times Impala will try connecting to the backend client after the first connection attempt fails. By default, `impalad` will attempt three re-connections before it returns a failure.

You can configure timeouts for sending and receiving data from the backend client. Therefore, if for some reason a query hangs, instead of waiting indefinitely for a response, Impala will terminate the connection after a configurable timeout.

- The `--backend_client_rpc_timeout_ms` option can be used to specify the number of milliseconds Impala should wait for a response from the backend client before it terminates the connection and signals a failure. The default value for this property is 300000 milliseconds, or 5 minutes.

Cancelling a Query

Occasionally, an Impala query might run for an unexpectedly long time, tying up resources in the cluster. This section describes the options to terminate such runaway queries.

Setting a Time Limit on Query Execution

An Impala administrator can set a default value of the `EXEC_TIME_LIMIT_S` query option for a resource pool. If a user accidentally runs a large query that executes for longer than the limit, it will be automatically terminated after the time limit expires to free up resources.

You can override the default value per query or per session if you do not want to apply the default `EXEC_TIME_LIMIT_S` value to a specific query or a session. See EXEC_TIME_LIMIT_S Query Option (CDH 5.15 / Impala 2.12 or higher only) on page 331 for the details of the query option.

Interactively Cancelling a Query

You can manually cancel a query in the Impala Web UI for the `impalad` host (on port 25000 by default):

1. Click **/queries**.
2. Click **Cancel** for a query in the **queries in flight** list.

Various client applications let you interactively cancel queries submitted or monitored through those applications. For example:

- Press ^C in `impala-shell`.
- Click **Cancel** from the **Watch** page in Hue.

Using Impala through a Proxy for High Availability

For most clusters that have multiple users and production availability requirements, you might want to set up a load-balancing proxy server to relay requests to and from Impala.

Set up a software package of your choice to perform these functions.



Note:

Most considerations for load balancing and high availability apply to the `impalad` daemon. The `statestored` and `catalogd` daemons do not have special requirements for high availability, because problems with those daemons do not result in data loss. If those daemons become unavailable due to an outage on a particular host, you can stop the Impala service, delete the **Impala StateStore** and **Impala Catalog Server** roles, add the roles on a different host, and restart the Impala service.

Overview of Proxy Usage and Load Balancing for Impala

Using a load-balancing proxy server for Impala offers the following advantages:

- Applications connect to a single well-known host and port, rather than keeping track of the hosts where the `impalad` daemons are running.
- If any host running an `impalad` becomes unavailable, application connection requests still succeed because you always connect to the proxy server rather than a specific host running the `impalad` daemon.

- The coordinator node for each Impala query potentially requires more memory and CPU cycles than the other nodes that process the query. The proxy server can schedule queries so that each connection uses a different coordinator node. This load-balancing technique lets the `impalad` nodes share this additional work, rather than directing it on a single machine.

The following are the general setup steps that apply to any load-balancing proxy software:

1. Select and download a load-balancing proxy software or other load-balancing hardware appliance. It should only need to be installed and configured on a single host, typically on an edge node.
2. Configure the load balancer (typically by editing a configuration file). In particular:
   - To relay Impala requests back and forth, set up a port that the load balancer will listen on.
   - Select a load balancing algorithm. See Choosing the Load-Balancing Algorithm on page 73 for load balancing algorithm options.
   - For Kerberized clusters, follow the instructions in Special Proxy Considerations for Clusters Using Kerberos on page 74.
3. If you are using Hue or JDBC-based applications, you typically set up load balancing for both ports 21000 and 21050, because these client applications connect through port 21050 while the `impala-shell` command connects through port 21000. See Ports Used by Impala on page 743 for when to use port 21000, 21050, or another value depending on what type of connections you are load balancing.
4. Run the load-balancing proxy server, pointing it at the configuration file that you set up.
5. On systems managed by Cloudera Manager:
   a. Navigate to **Impala** > **Configuration** > **Impala Daemon Default Group**.
   b. In the **Impala Daemons Load Balancer** field, specify the address of the load balancer in the *`host:port`* format.

      This setting lets Cloudera Manager route all appropriate Impala-related operations through the load-balancing proxy server.
6. For any scripts, jobs, or configuration settings for applications that formerly connected to a specific `impalad` to run Impala SQL statements, change the connection information (such as the `-i` option in `impala-shell`) to point to the load balancer instead.



Note: The following sections use the HAProxy software as a representative example of a load balancer that you can use with Impala. For information specifically about using Impala with the F5 BIG-IP load balancer, see Impala HA with F5 BIG-IP.

Choosing the Load-Balancing Algorithm

Load-balancing software offers a number of algorithms to distribute requests. Each algorithm has its own characteristics that make it suitable in some situations but not others.

Leastconn

Connects sessions to the coordinator with the fewest connections, to balance the load evenly. Typically used for workloads consisting of many independent, short-running queries. In configurations with only a few client machines, this setting can avoid having all requests go to only a small set of coordinators.

Recommended for Impala with F5.

Source IP Persistence

Sessions from the same IP address always go to the same coordinator. A good choice for Impala workloads containing a mix of queries and DDL statements, such as `CREATE TABLE` and `ALTER TABLE`. Because the metadata changes from a DDL statement take time to propagate across the cluster, prefer to use the Source IP Persistence algorithm in this case. If you are unable to choose Source IP Persistence, run the DDL and subsequent queries that depend on the results of the DDL through the same session, for example by running `impala-shell -f script_file` to submit several statements through a single session.

Required for setting up high availability with Hue. See Configure Hive and Impala for High Availability for configuring high availability with Hue.

Round-robin

Distributes connections to all coordinator nodes. Typically not recommended for Impala.

You might need to perform benchmarks and load testing to determine which setting is optimal for your use case. Always set up using two load-balancing algorithms: Source IP Persistence for Hue and Leastconn for others.

Special Proxy Considerations for Clusters Using Kerberos

In a cluster using Kerberos, applications check host credentials to verify that the host they are connecting to is the same one that is actually processing the request.

In Impala 2.11 and lower versions, once you enable a proxy server in a Kerberized cluster, users will not be able to connect to individual impala daemons directly from impala-shell.

In Impala 2.12 and higher, when you enable a proxy server in a Kerberized cluster, users have an option to connect to Impala daemons directly from `impala-shell` using the `-b` / `--kerberos_host_fqdn` `impala-shell` flag. This option can be used for testing or troubleshooting purposes, but not recommended for live production environments as it defeats the purpose of a load balancer/proxy.

Example:

```
impala-shell -i impalad-1.mydomain.com -k -b loadbalancer-1.mydomain.com
```

Alternatively, with the fully qualified configurations:

```
impala-shell --impalad=impalad-1.mydomain.com:21000 --kerberos
--kerberos_host_fqdn=loadbalancer-1.mydomain.com
```

See impala-shell Configuration Options on page 714 for information about the option.

To validate the load-balancing proxy server, perform these extra Kerberos setup steps:

1. This section assumes you are starting with a Kerberos-enabled cluster. See Enabling Kerberos Authentication for Impala on page 94 for instructions for setting up Impala with Kerberos. See Enabling Kerberos Authentication for CDH for general steps to set up Kerberos.
2. Choose the host you will use for the proxy server. Based on the Kerberos setup procedure, it should already have an entry `impala/proxy_host@realm` in its `keytab`.
3. In Cloudera Manager, navigate to **Impala** > **Configuration** > **Impala Daemon Default Group**.
4. In the **Impala Daemons Load Balancer** field, specify the address of the load balancer in the *host*:*port* format.

   When this field is specified and Kerberos is enabled, Cloudera Manager adds a principal for `impala/proxy_host@realm` to the `keytab` for all Impala daemons.

5. Restart the Impala service.

Client Connection to Proxy Server in Kerberized Clusters

When a client connect to Impala, the service principal specified by the client must match the `-principal` setting, `impala/proxy_host@realm`, of the Impala proxy server as specified in its `keytab`. And the client should connect to the proxy server port.

In `hue.ini`, set the following to configure Hue to automatically connect to the proxy server:

```
[impala]
server_host=proxy_host
impala_principal=impala/proxy_host
```

The following are the JDBC connection string formats when connecting through the load balancer with the load balancer's host name in the principal:

```
jdbc:hive2://proxy_host:load_balancer_port/;principal=impala/_HOST@realm
jdbc:hive2://proxy_host:load_balancer_port/;principal=impala/proxy_host@realm
```

When starting `impala-shell`, specify the service principal via the `-b` or `--kerberos_host_fqdn` flag.

Special Proxy Considerations for TLS/SSL Enabled Clusters

When TLS/SSL is enabled for Impala, the client application, whether impala-shell, Hue, or something else, expects the certificate common name (CN) to match the hostname that it is connected to. With no load balancing proxy server, the hostname and certificate CN are both that of the `impalad` instance. However, with a proxy server, the certificate presented by the `impalad` instance does not match the load balancing proxy server hostname. If you try to load-balance a TLS/SSL-enabled Impala installation without additional configuration, you see a certificate mismatch error when a client attempts to connect to the load balancing proxy host.

You can configure a proxy server in several ways to load balance TLS/SSL enabled Impala:

TLS/SSL Bridging

In this configuration, the proxy server presents a TLS/SSL certificate to the client, decrypts the client request, then re-encrypts the request before sending it to the backend `impalad`. The client and server certificates can be managed separately. The request or resulting payload is encrypted in transit at all times.

TLS/SSL Passthrough

In this configuration, traffic passes through to the backend `impalad` instance with no interaction from the load balancing proxy server. Traffic is still encrypted end-to-end.

The same server certificate, utilizing either wildcard or Subject Alternate Name (SAN), must be installed on each `impalad` instance.

TLS/SSL Offload

In this configuration, all traffic is decrypted on the load balancing proxy server, and traffic between the backend `impalad` instances is unencrypted. This configuration presumes that cluster hosts reside on a trusted network and only external client-facing communication need to be encrypted in-transit.

If you plan to use Auto-TLS, your load balancer must perform TLS/SSL bridging or TLS/SSL offload.

Refer to your load balancer documentation for the steps to set up Impala and the load balancer using one of the options above.

For information specifically about using Impala with the F5 BIG-IP load balancer with TLS/SSL enabled, see Impala HA with F5 BIG-IP.

Example of Configuring HAProxy Load Balancer for Impala

If you are not already using a load-balancing proxy, you can experiment with HAProxy a free, open source load balancer.

This example shows how you might install and configure that load balancer on a Red Hat Enterprise Linux system.

- Install the load balancer:

```
yum install haproxy
```

- Set up the configuration file: `/etc/haproxy/haproxy.cfg`. See the following section for a sample configuration file.
- Run the load balancer (on a single host, preferably one not running `impalad`):

```
/usr/sbin/haproxy –f /etc/haproxy/haproxy.cfg
```

- In `impala-shell`, JDBC applications, or ODBC applications, connect to the listener port of the proxy host, rather than port 21000 or 21050 on a host actually running `impalad`. The sample configuration file sets haproxy to listen on port 25003, therefore you would send all requests to *`haproxy_host`*`:25003`.

This is the sample `haproxy.cfg` used in this example:

```
global
    # To have these messages end up in /var/log/haproxy.log you will
    # need to:
    #
    # 1) configure syslog to accept network log events.  This is done
    #    by adding the '-r' option to the SYSLOGD_OPTIONS in
    #    /etc/sysconfig/syslog
    #
    # 2) configure local2 events to go to the /var/log/haproxy.log
    #   file. A line like the following can be added to
    #   /etc/sysconfig/syslog
    #
    #    local2.*                       /var/log/haproxy.log
    #
    log         127.0.0.1 local0
    log         127.0.0.1 local1 notice
    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    user        haproxy
    group       haproxy
    daemon

    # turn on stats unix socket
    #stats socket /var/lib/haproxy/stats

#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#
# You might need to adjust timing values to prevent timeouts.
#
# The timeout values should be dependant on how you use the cluster
# and how long your queries run.
#---------------------------------------------------------------------
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 3
    maxconn                 3000
    timeout connect 5000
    timeout client 3600s
    timeout server 3600s

#
# This sets up the admin page for HA Proxy at port 25002.
#
listen stats :25002
    balance
    mode http
    stats enable
    stats auth username:password

# This is the setup for Impala. Impala client connect to load_balancer_host:25003.
# HAProxy will balance connections among the list of servers listed below.
# The list of Impalad is listening at port 21000 for beeswax (impala-shell) or original
 ODBC driver.
# For JDBC or ODBC version 2.x driver, use port 21050 instead of 21000.
listen impala :25003
    mode tcp
    option tcplog
```

```
    balance leastconn

    server symbolic_name_1 impala-host-1.example.com:21000 check
    server symbolic_name_2 impala-host-2.example.com:21000 check
    server symbolic_name_3 impala-host-3.example.com:21000 check
    server symbolic_name_4 impala-host-4.example.com:21000 check

# Setup for Hue or other JDBC-enabled applications.
# In particular, Hue requires sticky sessions.
# The application connects to load_balancer_host:21051, and HAProxy balances
# connections to the associated hosts, where Impala listens for JDBC
# requests on port 21050.
listen impalajdbc :21051
    mode tcp
    option tcplog
    balance source
    server symbolic_name_5 impala-host-1.example.com:21050 check
    server symbolic_name_6 impala-host-2.example.com:21050 check
    server symbolic_name_7 impala-host-3.example.com:21050 check
    server symbolic_name_8 impala-host-4.example.com:21050 check
```

Important: Hue requires the `check` option at the end of each line in the above file to ensure HAProxy can detect any unreachable `impalad` server, and failover can be successful. Without the TCP check, you can hit an error when the `impalad` daemon to which Hue tries to connect is down.

Note: If your JDBC or ODBC application connects to Impala through a load balancer such as `haproxy`, be cautious about reusing the connections. If the load balancer has set up connection timeout values, either check the connection frequently so that it never sits idle longer than the load balancer timeout value, or check the connection validity before using it and create a new one if the connection has been closed.

Managing Disk Space for Impala Data

Although Impala typically works with many large files in an HDFS storage system with plenty of capacity, there are times when you might perform some file cleanup to reclaim space, or advise developers on techniques to minimize space consumption and file duplication.

- Use compact binary file formats where practical. Numeric and time-based data in particular can be stored in more compact form in binary data files. Depending on the file format, various compression and encoding features can reduce file size even further. You can specify the `STORED AS` clause as part of the `CREATE TABLE` statement, or `ALTER TABLE` with the `SET FILEFORMAT` clause for an existing table or partition within a partitioned table. See How Impala Works with Hadoop File Formats on page 634 for details about file formats, especially Using the Parquet File Format with Impala Tables on page 643. See CREATE TABLE Statement on page 234 and ALTER TABLE Statement on page 205 for syntax details.
- You manage underlying data files differently depending on whether the corresponding Impala table is defined as an internal or external table:
  - Use the `DESCRIBE FORMATTED` statement to check if a particular table is internal (managed by Impala) or external, and to see the physical location of the data files in HDFS. See DESCRIBE Statement on page 251 for details.
  - For Impala-managed ("internal") tables, use `DROP TABLE` statements to remove data files. See DROP TABLE Statement on page 268 for details.
  - For tables not managed by Impala ("external" tables), use appropriate HDFS-related commands such as `hadoop fs`, `hdfs dfs`, or `distcp`, to create, move, copy, or delete files within HDFS directories that are accessible by the `impala` user. Issue a `REFRESH` *`table_name`* statement after adding or removing any files from the data directory of an external table. See REFRESH Statement on page 291 for details.

  - Use external tables to reference HDFS data files in their original location. With this technique, you avoid copying the files, and you can map more than one Impala table to the same set of data files. When you drop the Impala table, the data files are left undisturbed. See External Tables on page 197 for details.
  - Use the `LOAD DATA` statement to move HDFS files into the data directory for an Impala table from inside Impala, without the need to specify the HDFS path of the destination directory. This technique works for both internal and external tables. See LOAD DATA Statement on page 288 for details.

- Make sure that the HDFS trashcan is configured correctly. When you remove files from HDFS, the space might not be reclaimed for use by other files until sometime later, when the trashcan is emptied. See DROP TABLE Statement on page 268 and the FAQ entry Why is space not freed up when I issue DROP TABLE? on page 772 for details. See User Account Requirements on page 25 for permissions needed for the HDFS trashcan to operate correctly.
- Drop all tables in a database before dropping the database itself. See DROP DATABASE Statement on page 262 for details.
- Clean up temporary files after failed `INSERT` statements. If an `INSERT` statement encounters an error, and you see a directory named `.impala_insert_staging` or `_impala_insert_staging` left behind in the data directory for the table, it might contain temporary data files taking up space in HDFS. You might be able to salvage these data files, for example if they are complete but could not be moved into place due to a permission error. Or, you might delete those files through commands such as `hadoop fs` or `hdfs dfs`, to reclaim space before re-trying the `INSERT`. Issue `DESCRIBE FORMATTED` *`table_name`* to see the HDFS path where you can check for temporary files.
- By default, intermediate files used during large sort, join, aggregation, or analytic function operations are stored in the directory `/tmp/impala-scratch` . These files are removed when the operation finishes. (Multiple concurrent queries can perform operations that use the “spill to disk” technique, without any name conflicts for these temporary files.) You can specify a different location by starting the `impalad` daemon with the `--scratch_dirs="`*`path_to_directory`*`"` configuration option or the equivalent configuration option in the Cloudera Manager user interface. You can specify a single directory, or a comma-separated list of directories. The scratch directories must be on the local filesystem, not in HDFS. You might specify different directory paths for different hosts, depending on the capacity and speed of the available storage devices. In CDH 5.5 / Impala 2.3 or higher, Impala successfully starts (with a warning written to the log) if it cannot create or read and write files in one of the scratch directories. If there is less than 1 GB free on the filesystem where that directory resides, Impala still runs, but writes a warning message to its log. If Impala encounters an error reading or writing files in a scratch directory during a query, Impala logs the error and the query fails.
- If you use the Amazon Simple Storage Service (S3) as a place to offload data to reduce the volume of local storage, Impala 2.2.0 and higher can query the data directly from S3. See Using Impala with the Amazon S3 Filesystem on page 692 for details.

Auditing Impala Operations

To monitor how Impala data is being used within your organization, ensure that your Impala authorization and authentication policies are effective, and detect attempts at intrusion or unauthorized access to Impala data, you can use the auditing feature in Impala 1.2.1 and higher:

- Enable auditing by including the option `--audit_event_log_dir=`*`directory_path`* in your `impalad` startup options for a cluster not managed by Cloudera Manager, or configuring Impala Daemon logging in Cloudera Manager. The log directory must be a local directory on the server, not an HDFS directory.
- Decide how many queries will be represented in each log files. By default, Impala starts a new log file every 5000 queries. To specify a different number, include the option `--max_audit_event_log_file_size=`*`number_of_queries`* in the `impalad` startup options.
- Configure Cloudera Navigator to collect and consolidate the audit logs from all the hosts in the cluster.
- In CDH 5.12 / Impala 2.9 and higher, you can control how many audit event log files are kept on each host. Specify the option `--max_audit_event_log_files=`*`number_of_log_files`* in the `impalad` startup options. Once the limit is reached, older files are rotated out using the same mechanism as for other Impala log files. The default value for this setting is 0, representing an unlimited number of audit event log files.

- Use Cloudera Navigator or Cloudera Manager to filter, visualize, and produce reports based on the audit data. (The Impala auditing feature works with Cloudera Manager 4.7 to 5.1 and Cloudera Navigator 2.1 and higher.) Check the audit data to ensure that all activity is authorized and detect attempts at unauthorized access.

Durability and Performance Considerations for Impala Auditing

The auditing feature only imposes performance overhead while auditing is enabled.

Because any Impala host can process a query, enable auditing on all hosts where the `impalad` daemon runs. Each host stores its own log files, in a directory in the local filesystem. The log data is periodically flushed to disk (through an `fsync()` system call) to avoid loss of audit data in case of a crash.

The runtime overhead of auditing applies to whichever host serves as the coordinator for the query, that is, the host you connect to when you issue the query. This might be the same host for all queries, or different applications or users might connect to and issue queries through different hosts.

To avoid excessive I/O overhead on busy coordinator hosts, Impala syncs the audit log data (using the `fsync()` system call) periodically rather than after every query. Currently, the `fsync()` calls are issued at a fixed interval, every 5 seconds.

By default, Impala avoids losing any audit log data in the case of an error during a logging operation (such as a disk full error), by immediately shutting down `impalad` on the host where the auditing problem occurred. You can override this setting by specifying the option `--abort_on_failed_audit_event=false` in the `impalad` startup options.

Format of the Audit Log Files

The audit log files represent the query information in JSON format, one query per line. Typically, rather than looking at the log files themselves, you use the Cloudera Navigator product to consolidate the log data from all Impala hosts and filter and visualize the results in useful ways. (If you do examine the raw log data, you might run the files through a JSON pretty-printer first.)

All the information about schema objects accessed by the query is encoded in a single nested record on the same line. For example, the audit log for an `INSERT ... SELECT` statement records that a select operation occurs on the source table and an insert operation occurs on the destination table. The audit log for a query against a view records the base table accessed by the view, or multiple base tables in the case of a view that includes a join query. Every Impala operation that corresponds to a SQL statement is recorded in the audit logs, whether the operation succeeds or fails. Impala records more information for a successful operation than for a failed one, because an unauthorized query is stopped immediately, before all the query planning is completed.

The information logged for each query includes:

- Client session state:
  - Session ID
  - User name
  - Network address of the client connection
- SQL statement details:
  - Query ID
  - Statement Type - DML, DDL, and so on
  - SQL statement text
  - Execution start time, in local time
  - Execution Status - Details on any errors that were encountered
  - Target Catalog Objects:
    - Object Type - Table, View, or Database
    - Fully qualified object name
    - Privilege - How the object is being used (`SELECT`, `INSERT`, `CREATE`, and so on)

Which Operations Are Audited

The kinds of SQL queries represented in the audit log are:

- Queries that cause data to be returned or changed.

  This includes queries that fail while running but still stream partial results to the user.

- Queries that are prevented due to lack of authorization, even if nothing is returned or changed.
- Queries that Impala can analyze and parse to determine that they are authorized.

  The audit data is recorded immediately after Impala finishes its analysis and before the query is actually executed.

Other than audits for lack of authorization, the audit log does not contain entries for queries that could not be parsed and analyzed, such as errors, cancelled queries, and timed out queries. For example, a query that fails due to a syntax error is not recorded in the audit log. Certain statements in the Impala-shell interpreter, such as `CONNECT`, `SUMMARY`, `PROFILE`, `SET`, and `QUIT`, do not correspond to actual SQL queries; these statements are not reflected in the audit log.

Reviewing the Audit Logs

You typically do not review the audit logs in raw form. The Cloudera Manager Agent periodically transfers the log information into a back-end database where it can be examined in consolidated form. See the Cloudera Navigator documentation for details .

Viewing Lineage Information for Impala Data

Lineage is a feature in the Cloudera Navigator data management component that helps you track where data originated, and how data propagates through the system through SQL statements such as `SELECT`, `INSERT`, and `CREATE TABLE AS SELECT`. Impala is covered by the Cloudera Navigator lineage features in CDH 5.4 / Impala 2.2 and higher.

This type of tracking is important in high-security configurations, especially in highly regulated industries such as healthcare, pharmaceuticals, financial services and intelligence. For such kinds of sensitive data, it is important to know all the places in the system that contain that data or other data derived from it; to verify who has accessed that data; and to be able to doublecheck that the data used to make a decision was processed correctly and not tampered with.

You interact with this feature through ***lineage diagrams*** showing relationships between tables and columns. For instructions about interpreting lineage diagrams, see http://www.cloudera.com/documentation/enterprise/latest/topics/cn_iu_lineage.html.

Column Lineage

Column lineage tracks information in fine detail, at the level of particular columns rather than entire tables.

For example, if you have a table with information derived from web logs, you might copy that data into other tables as part of the ETL process. The ETL operations might involve transformations through expressions and function calls, and rearranging the columns into more or fewer tables (***normalizing*** or ***denormalizing*** the data). Then for reporting, you might issue queries against multiple tables and views. In this example, column lineage helps you determine that data that entered the system as `RAW_LOGS.FIELD1` was then turned into `WEBSITE_REPORTS.IP_ADDRESS` through an `INSERT ... SELECT` statement. Or, conversely, you could start with a reporting query against a view, and trace the origin of the data in a field such as `TOP_10_VISITORS.USER_ID` back to the underlying table and even further back to the point where the data was first loaded into Impala.

When you have tables where you need to track or control access to sensitive information at the column level, see Enabling Sentry Authorization for Impala on page 87 for how to implement column-level security. You set up authorization using the Sentry framework, create views that refer to specific sets of columns, and then assign authorization privileges to those views rather than the underlying tables.

Lineage Data for Impala

The lineage feature is enabled by default. When lineage logging is enabled, the serialized column lineage graph is computed for each query and stored in a specialized log file in JSON format.

Impala records queries in the lineage log if they complete successfully, or fail due to authorization errors. For write operations such as `INSERT` and `CREATE TABLE AS SELECT`, the statement is recorded in the lineage log only if it successfully completes. Therefore, the lineage feature tracks data that was accessed by successful queries, or that was attempted to be accessed by unsuccessful queries that were blocked due to authorization failure. These kinds of queries represent data that really was accessed, or where the attempted access could represent malicious activity.

Impala does not record in the lineage log queries that fail due to syntax errors or that fail or are cancelled before they reach the stage of requesting rows from the result set.

To enable or disable this feature on a system not managed by Cloudera Manager, set or remove the `-lineage_event_log_dir` configuration option for the `impalad` daemon. For information about turning the lineage feature on and off through Cloudera Manager, see http://www.cloudera.com/documentation/enterprise/latest/topics/datamgmt_impala_lineage_log.html.

Impala Security

Impala includes a fine-grained authorization framework for Hadoop, based on the Sentry open source project. Sentry authorization was added in Impala 1.1.0. Together with the Kerberos authentication framework, Sentry takes Hadoop security to a new level needed for the requirements of highly regulated industries such as healthcare, financial services, and government. Impala also includes an auditing capability; Impala generates the audit data, the Cloudera Navigator product consolidates the audit data from all nodes in the cluster, and Cloudera Manager lets you filter, visualize, and produce reports. The auditing feature was added in Impala 1.1.1.

The Impala security features have several objectives. At the most basic level, security prevents accidents or mistakes that could disrupt application processing, delete or corrupt data, or reveal data to unauthorized users. More advanced security features and practices can harden the system against malicious users trying to gain unauthorized access or perform other disallowed operations. The auditing feature provides a way to confirm that no unauthorized access occurred, and detect whether any such attempts were made. This is a critical set of features for production deployments in large organizations that handle important or sensitive data. It sets the stage for multi-tenancy, where multiple applications run concurrently and are prevented from interfering with each other.

The material in this section presumes that you are already familiar with administering secure Linux systems. That is, you should know the general security practices for Linux and Hadoop, and their associated commands and configuration files. For example, you should know how to create Linux users and groups, manage Linux group membership, set Linux and HDFS file permissions and ownership, and designate the default permissions and ownership for new files. You should be familiar with the configuration of the nodes in your Hadoop cluster, and know how to apply configuration changes or run a set of commands across all the nodes.

The security features are divided into these broad categories:

authorization

Which users are allowed to access which resources, and what operations are they allowed to perform? Impala relies on the open source Sentry project for authorization. By default (when authorization is not enabled), Impala does all read and write operations with the privileges of the `impala` user, which is suitable for a development/test environment but not for a secure production environment. When authorization is enabled, Impala uses the OS user ID of the user who runs `impala-shell` or other client program, and associates various privileges with each user. See Enabling Sentry Authorization for Impala on page 87 for details about setting up and managing authorization.

authentication

How does Impala verify the identity of the user to confirm that they really are allowed to exercise the privileges assigned to that user? Impala relies on the Kerberos subsystem for authentication. See Enabling Kerberos Authentication for Impala on page 94 for details about setting up and managing authentication.

auditing

What operations were attempted, and did they succeed or not? This feature provides a way to look back and diagnose whether attempts were made to perform unauthorized operations. You use this information to track down suspicious activity, and to see where changes are needed in authorization policies. The audit data produced by this feature is collected by the Cloudera Manager product and then presented in a user-friendly form by the Cloudera Manager product. See Auditing Impala Operations on page 78 for details about setting up and managing auditing.

Security Guidelines for Impala

The following are the major steps to harden a cluster running Impala against accidents and mistakes, or malicious attackers trying to access sensitive data:

- Secure the `root` account. The `root` user can tamper with the `impalad` daemon, read and write the data files in HDFS, log into other user accounts, and access other system services that are beyond the control of Impala.

- Restrict membership in the `sudoers` list (in the `/etc/sudoers` file). The users who can run the `sudo` command can do many of the same things as the `root` user.
- Ensure the Hadoop ownership and permissions for Impala data files are restricted.
- Ensure the Hadoop ownership and permissions for Impala log files are restricted.
- Ensure that the Impala web UI (available by default on port 25000 on each Impala node) is password-protected. See Impala Web User Interface for Debugging on page 733 for details.
- Create a policy file that specifies which Impala privileges are available to users in particular Hadoop groups (which by default map to Linux OS groups). Create the associated Linux groups using the `groupadd` command if necessary.
- The Impala authorization feature makes use of the HDFS file ownership and permissions mechanism; for background information, see the HDFS Permissions Guide. Set up users and assign them to groups at the OS level, corresponding to the different categories of users with different access levels for various databases, tables, and HDFS locations (URIs). Create the associated Linux users using the `useradd` command if necessary, and add them to the appropriate groups with the `usermod` command.
- Design your databases, tables, and views with database and table structure to allow policy rules to specify simple, consistent rules. For example, if all tables related to an application are inside a single database, you can assign privileges for that database and use the `*` wildcard for the table name. If you are creating views with different privileges than the underlying base tables, you might put the views in a separate database so that you can use the `*` wildcard for the database containing the base tables, while specifying the precise names of the individual views. (For specifying table or database names, you either specify the exact name or `*` to mean all the databases on a server, or all the tables and views in a database.)
- Enable authorization by running the `impalad` daemons with the `-server_name` and `-authorization_policy_file` options on all nodes. (The authorization feature does not apply to the `statestored` daemon, which has no access to schema objects or data files.)
- Set up authentication using Kerberos, to make sure users really are who they say they are.

Securing Impala Data and Log Files

One aspect of security is to protect files from unauthorized access at the filesystem level. For example, if you store sensitive data in HDFS, you specify permissions on the associated files and directories in HDFS to restrict read and write permissions to the appropriate users and groups.

If you issue queries containing sensitive values in the `WHERE` clause, such as financial account numbers, those values are stored in Impala log files in the Linux filesystem and you must secure those files also. For the locations of Impala log files, see Using Impala Logging on page 709.

All Impala read and write operations are performed under the filesystem privileges of the `impala` user. The `impala` user must be able to read all directories and data files that you query, and write into all the directories and data files for `INSERT` and `LOAD DATA` statements. At a minimum, make sure the `impala` user is in the `hive` group so that it can access files and directories shared between Impala and Hive. See User Account Requirements on page 25 for more details.

Setting file permissions is necessary for Impala to function correctly, but is not an effective security practice by itself:

- The way to ensure that only authorized users can submit requests for databases and tables they are allowed to access is to set up Sentry authorization, as explained in Enabling Sentry Authorization for Impala on page 87. With authorization enabled, the checking of the user ID and group is done by Impala, and unauthorized access is blocked by Impala itself. The actual low-level read and write requests are still done by the `impala` user, so you must have appropriate file and directory permissions for that user ID.
- You must also set up Kerberos authentication, as described in Enabling Kerberos Authentication for Impala on page 94, so that users can only connect from trusted hosts. With Kerberos enabled, if someone connects a new

host to the network and creates user IDs that match your privileged IDs, they will be blocked from connecting to Impala at all from that host.

Installation Considerations for Impala Security

Impala 1.1 comes set up with all the software and settings needed to enable security when you run the `impalad` daemon with the new security-related options (`-server_name` and `-authorization_policy_file`). You do not need to change any environment variables or install any additional JAR files. In a cluster managed by Cloudera Manager, you do not need to change any settings in Cloudera Manager.

Securing the Hive Metastore Database

It is important to secure the Hive metastore, so that users cannot access the names or other information about databases and tables the through the Hive client or by querying the metastore database. Do this by turning on Hive metastore security, using the instructions in the CDH Security Guide for securing different Hive components:

- Secure the Hive Metastore.
- In addition, allow access to the metastore only from the HiveServer2 server, and then disable local access to the HiveServer2 server.

Securing the Impala Web User Interface

The instructions in this section presume you are familiar with the `.htpasswd` mechanism commonly used to password-protect pages on web servers.

Password-protect the Impala web UI that listens on port 25000 by default. Set up a `.htpasswd` file in the `$IMPALA_HOME` directory, or start both the `impalad` and `statestored` daemons with the `--webserver_password_file` option to specify a different location (including the filename).

This file should only be readable by the Impala process and machine administrators, because it contains (hashed) versions of passwords. The username / password pairs are not derived from Unix usernames, Kerberos users, or any other system. The `domain` field in the password file must match the domain supplied to Impala by the new command-line option `--webserver_authentication_domain`. The default is `mydomain.com`.

Impala also supports using HTTPS for secure web traffic. To do so, set `--webserver_certificate_file` to refer to a valid `.pem` TLS/SSL certificate file. Impala will automatically start using HTTPS once the TLS/SSL certificate has been read and validated. A `.pem` file is basically a private key, followed by a signed TLS/SSL certificate; make sure to concatenate both parts when constructing the `.pem` file.

If Impala cannot find or parse the `.pem` file, it prints an error message and quits.



Note:

If the private key is encrypted using a passphrase, Impala will ask for that passphrase on startup, which is not useful for a large cluster. In that case, remove the passphrase and make the `.pem` file readable only by Impala and administrators.

When you turn on TLS/SSL for the Impala web UI, the associated URLs change from `http://` prefixes to `https://`. Adjust any bookmarks or application code that refers to those URLs.

Configuring Secure Access for Impala Web Servers

Cloudera Manager supports password-based authentication for secure access to the Impala Catalog Server, Daemon, and StateStoreweb servers.

Authentication for the three types of daemons can be configured independently.

Configuring Password Authentication

1. Navigate to **Clusters** > **Impala Service** > **Configuration**.
2. Search for "password" using the Search box in the **Configuration** tab. This should display the password-related properties (Username and Password properties) for the Impala Daemon, StateStore, and Catalog Server. If there are multiple role groups configured for Impala Daemon instances, the search should display all of them.
3. Enter a username and password into these fields.
4. Click **Save Changes**, and restart the Impala service.

Now when you access the Web UI for the Impala Daemon, StateStore, or Catalog Server, you are asked to log in before access is granted.

Configuring TLS/SSL for Impala Web UI

1. Create or obtain an TLS/SSL certificate.
2. Place the certificate, in `.pem` format, on the hosts where the Impala Catalog Server and StateStore are running, and on each host where an Impala Daemon is running. It can be placed in any location (path) you choose. If all the Impala Daemons are members of the same role group, then the `.pem` file must have the same path on every host.
3. Navigate to **Clusters** > **Impala Service** > **Configuration**.
4. Search for "certificate" using the Search box in the **Configuration** tab. This should display the certificate file location properties for the Impala Catalog Server, Impala Daemon, and StateStore. If there are multiple role groups configured for Impala Daemon instances, the search should display all of them.
5. In the property fields, enter the full path name to the certificate file, fill out the private key file path, and the password for the private key file..
6. Click **Save Changes**, and restart the Impala service.

> **Important:** If Cloudera Manager cannot find the `.pem` file on the host for a specific role instance, that role will fail to start.

When you access the Web UI for the Impala Catalog Server, Impala Daemon, and StateStore, `https` will be used.

Configuring TLS/SSL for Impala

Impala supports TLS/SSL network encryption, between Impala and client programs, and between the Impala-related daemons running on different nodes in the cluster. This feature is important when you also use other features such as Kerberos authentication or Sentry authorization, where credentials are being transmitted back and forth.

> **Important:**
>
> - You can use either Cloudera Manager or the following command-line instructions to complete this configuration.
> - This information applies specifically to the version of Impala shown in the HTML page header or on the PDF title page. If you use an earlier version of CDH, see the documentation for that version located at Cloudera Documentation.

Using Cloudera Manager

To configure Impala to listen for Beeswax and HiveServer2 requests on TLS/SSL-secured ports:

1. Open the Cloudera Manager Admin Console and go to the **Impala** service.
2. Click the **Configuration** tab.
3. Select **Scope** > **Impala (Service-Wide)**.
4. Select **Category** > **Security**.

5. Edit the following properties:

Table 1: Impala SSL Properties

Property	Description
Enable TLS/SSL for Impala	Encrypt communication between clients (like ODBC, JDBC, and the Impala shell) and the Impala daemon using Transport Layer Security (TLS) (formerly known as Secure Socket Layer (SSL)).
Impala TLS/SSL Server Certificate File (PEM Format)	Local path to the X509 certificate that identifies the Impala daemon to clients during TLS/SSL connections. This file must be in PEM format.
Impala TLS/SSL Server Private Key File (PEM Format)	Local path to the private key that matches the certificate specified in the Certificate for Clients. This file must be in PEM format.
Impala TLS/SSL Private Key Password	The password for the private key in the Impala TLS/SSL Server Certificate and Private Key file. If left blank, the private key is not protected by a password.
Impala TLS/SSL CA Certificate	The location on disk of the certificate, in PEM format, used to confirm the authenticity of SSL/TLS servers that the Impala daemons might connect to. Because the Impala daemons connect to each other, this should also include the CA certificate used to sign all the SSL/TLS Certificates. Without this parameter, SSL/TLS between Impala daemons will not be enabled.

6. Click **Save Changes** to commit the changes.
7. Select each scope, one each for "Impala Daemon", "Catalog Server", and "Statestore", and repeat the above steps. Each of these Impala components has its own internal web server that powers the associated web UI with diagnostic information. The configuration setting represents the local path to the X509 certificate that identifies the web server to clients during TLS/SSL connections.
8. Restart the Impala service.

Configuring TLS/SSL Communication for the Impala Shell

Typically, a client program has corresponding configuration properties in Cloudera Manager to verify that it is connecting to the right server. For example, with TLS/SSL enabled for Impala, you use the following options when starting `impala-shell`:

- `--ssl`: enables TLS/SSL for `impala-shell`.
- `--ca_cert`: the local pathname pointing to the third-party CA certificate, or to a copy of the server certificate for self-signed server certificates.

If `--ca_cert` is not set, `impala-shell` enables TLS/SSL, but does not validate the server certificate. This is useful for connecting to a known-good Impala that is only running over TLS/SSL, when a copy of the certificate is not available (such as when debugging customer installations).

For `impala-shell` to successfully connect to an Impala cluster that has the minimum allowed TLS/SSL version set to 1.2 (`--ssl_minimum_version=tlsv1.2`), the Python version on the cluster that `impala-shell` runs on must be 2.7.9 or higher (or a vendor-provided Python version with the required support. Some vendors patched Python 2.7.5 versions on Red Hat Enterprise Linux 7 and derivatives).

Using TLS/SSL with Business Intelligence Tools

You can use Kerberos authentication, TLS/SSL encryption, or both to secure connections from JDBC and ODBC applications to Impala. See Configuring Impala to Work with JDBC on page 725 and Configuring Impala to Work with ODBC on page 723 for details.

Prior to CDH 5.7 / Impala 2.5, the Hive JDBC driver did not support connections that use both Kerberos authentication and SSL encryption. If your cluster is running an older release that has this restriction, to use both of these security features with Impala through a JDBC application, use the Cloudera JDBC Connector as the JDBC driver.

Specifying TLS/SSL Minimum Allowed Version and Ciphers

Depending on your cluster configuration and the security practices in your organization, you might need to restrict the allowed versions of TLS/SSL used by Impala. Older TLS/SSL versions might have vulnerabilities or lack certain features. In CDH 5.13 / Impala 2.10, you can use startup options for the `impalad`, `catalogd`, and `statestored` daemons to specify a minimum allowed version of TLS/SSL.

Specify one of the following values for the `--ssl_minimum_version` configuration setting:

- `tlsv1`: Allow any TLS version of 1.0 or higher. This setting is the default when TLS/SSL is enabled.
- `tlsv1.1`: Allow any TLS version of 1.1 or higher.
- `tlsv1.2`: Allow any TLS version of 1.2 or higher.

Along with specifying the version, you can also specify the allowed set of TLS ciphers by using the `--ssl_cipher_list` configuration setting. The argument to this option is a list of keywords, separated by colons, commas, or spaces, and optionally including other notation. For example:

```
--ssl_cipher_list="RC4-SHA,RC4-MD5"
```

By default, the cipher list is empty, and Impala uses the default cipher list for the underlying platform. See the output of `man ciphers` for the full set of keywords and notation allowed in the argument string.

Enabling Sentry Authorization for Impala

Authorization determines which users are allowed to access which resources, and what operations they are allowed to perform. In Impala 1.1 and higher, you use Apache Sentry for authorization. Sentry adds a fine-grained authorization framework for Hadoop. By default (when authorization is not enabled), Impala does all read and write operations with the privileges of the `impala` user, which is suitable for a development/test environment but not for a secure production environment. When authorization is enabled, Impala uses the OS user ID of the user who runs `impala-shell` or other client program, and associates various privileges with each user.



Note: Sentry is typically used in conjunction with Kerberos authentication that defines which hosts are allowed to connect to each server. Using the combination of Sentry and Kerberos prevents malicious users from being able to connect by creating a named account on an untrusted machine. See Enabling Kerberos Authentication for Impala on page 94 for details about Kerberos authentication.

The Sentry Privilege Model

Privileges can be granted on different objects in the schema and are associated with a level in the object hierarchy. If a privilege is granted on a parent object in the hierarchy, the child object automatically inherits it. This is the same privilege model as Hive and other database systems.

The objects in the Impala schema hierarchy are:

```
Server
    URI
    Database
        Table
            Column
```

The table-level privileges apply to views as well. Anywhere you specify a table name, you can specify a view name instead.

In CDH 5.5 / Impala 2.3 and higher, you can specify privileges for individual columns.

The table below lists the minimum level of privileges and the scope required to execute SQL statements in CDH 6.1 / CDH 5.16 and higher. The following notations are used:

- **ANY** denotes the SELECT, INSERT, CREATE, ***or*** REFRESH privilege.
- **ALL** privilege denotes the SELECT, INSERT, CREATE, ***and*** REFRESH privileges.
- The owner of an object effectively has the ALL privilege on the object.
- The parent levels of the specified scope are implicitly supported. For example, if a privilege is listed with the TABLE scope, the same privilege granted on DATABASE and SERVER will allow the user to execute that specific SQL statement on TABLE.

SQL Statement	Privileges	Scope
SELECT	SELECT	TABLE
WITH SELECT	SELECT	TABLE
EXPLAIN SELECT	SELECT	TABLE
INSERT	INSERT	TABLE
EXPLAIN INSERT	INSERT	TABLE
TRUNCATE	INSERT	TABLE
LOAD	INSERT	TABLE
	ALL	URI
CREATE DATABASE	CREATE	SERVER
CREATE DATABASE LOCATION	CREATE	SERVER
	ALL	URI
CREATE TABLE	CREATE	DATABASE
CREATE TABLE LIKE	CREATE	DATABASE
	SELECT, INSERT, ***or*** REFRESH	TABLE
CREATE TABLE AS SELECT	CREATE	DATABASE
	INSERT	DATABASE
	SELECT	TABLE
EXPLAIN CREATE TABLE AS SELECT	CREATE	DATABASE
	INSERT	DATABASE
	SELECT	TABLE
CREATE TABLE LOCATION	CREATE	TABLE
	ALL	URI
CREATE VIEW	CREATE	DATABASE
	SELECT	TABLE
ALTER DATABASE SET OWNER	ALL WITH GRANT	DATABASE
ALTER TABLE	ALL	TABLE
ALTER TABLE SET LOCATION	ALL	TABLE
	ALL	URI
ALTER TABLE RENAME	CREATE	DATABASE
	ALL	TABLE
ALTER TABLE SET OWNER	ALL WITH GRANT	TABLE

ALTER VIEW	ALL	TABLE
	SELECT	TABLE
ALTER VIEW RENAME	CREATE	DATABASE
	ALL	TABLE
ALTER VIEW SET OWNER	ALL WITH GRANT	VIEW
DROP DATABASE	ALL	DATABASE
DROP TABLE	ALL	TABLE
DROP VIEW	ALL	TABLE
CREATE FUNCTION	CREATE	DATABASE
	ALL	URI
DROP FUNCTION	ALL	DATABASE
COMPUTE STATS	ALL	TABLE
DROP STATS	ALL	TABLE
INVALIDATE METADATA	REFRESH	SERVER
INVALIDATE METADATA <table>	REFRESH	TABLE
REFRESH <table>	REFRESH	TABLE
REFRESH AUTHORIZATION	REFRESH	SERVER
REFRESH FUNCTIONS	REFRESH	DATABASE
COMMENT ON DATABASE	ALL	DATABASE
COMMENT ON TABLE	ALL	TABLE
COMMENT ON VIEW	ALL	TABLE
COMMENT ON COLUMN	ALL	TABLE
DESCRIBE DATABASE	SELECT, INSERT, ***or*** REFRESH	DATABASE
DESCRIBE <table/view>	SELECT, INSERT, ***or*** REFRESH	TABLE
If the user has the SELECT privilege at the COLUMN level, only the columns the user has access will show.	SELECT	COLUMN
USE	ANY	TABLE
SHOW DATABASES	ANY	TABLE
SHOW TABLES	ANY	TABLE
SHOW FUNCTIONS	SELECT, INSERT, ***or*** REFRESH	DATABASE
SHOW PARTITIONS	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW TABLE STATS	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW COLUMN STATS	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW FILES	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW CREATE TABLE	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW CREATE VIEW	SELECT, INSERT, ***or*** REFRESH	TABLE

SHOW CREATE FUNCTION	SELECT, INSERT, ***or*** REFRESH	DATABASE
SHOW RANGE PARTITIONS (Kudu only)	SELECT, INSERT, ***or*** REFRESH	TABLE
UPDATE (Kudu only)	ALL	TABLE
EXPLAIN UPDATE (Kudu only)	ALL	TABLE
UPSERT (Kudu only)	ALL	TABLE
WITH UPSERT (Kudu only)	ALL	TABLE
EXPLAIN UPSERT (Kudu only)	ALL	TABLE
DELETE (Kudu only)	ALL	TABLE
EXPLAIN DELETE (Kudu only)	ALL	TABLE



Note: If a specific privilege is not enabled in Sentry, the `ALL` privilege is required in Impala.

Privileges are managed via the `GRANT` and `REVOKE` SQL statements that requires the Sentry service enabled. The Sentry service stores, retrieves, and manipulates privilege information stored inside the Sentry database.

If you change privileges outside of Impala, e.g. adding a user, removing a user, modifying privileges, you must clear the Impala Catalog server cache by running the `REFRESH AUTHORIZATION` statement. `REFRESH AUTHORIZATION` is not required if you make the changes to privileges within Impala.

Object Ownership in Sentry

Starting in CDH 5.16 and CDH 6.1, Impala supports the ownership on databases, tables, and views. The `CREATE` statements implicitly make the user running the statement the owner of the object. An owner has the `OWNER` privilege if enabled in Sentry. For example, if *User A* creates a database, *foo*, via the `CREATE DATABASE` statement, *User A* now owns the *foo* database and is authorized to perform any operation on the *foo* database.

The `OWNER` privilege is not a grantable or revokable privilege whereas the `ALL` privilege is explicitly granted via the `GRANT` statement.

The object ownership feature is controlled by a Sentry configuration. The `OWNER` privilege is only granted when the feature is enabled in Sentry. When enabled they get the owner privilege, with or without the `GRANT OPTION`, which is also controlled by the Sentry configuration.

An ownership can be transferred to another user or role via the `ALTER DATABASE`, `ALTER TABLE`, or `ALTER VIEW` with the `SET OWNER` clause.

Starting the impalad Daemon with Sentry Authorization Enabled

To run the `impalad` daemon with authorization enabled, you add the following options to the `IMPALA_SERVER_ARGS` declaration in the `/etc/default/impala` configuration file:

- `-server_name`: Turns on Sentry authorization for Impala. The authorization rules refer to a symbolic server name, and you specify the name to use as the argument to the `-server_name` option.
- `--sentry_config`: Specifies the local path to the `sentry-site.xml` configuration file. This setting is required to enable authorization.

For example, you might adapt your `/etc/default/impala` configuration to contain lines like the following. To use the Sentry service:

```
IMPALA_SERVER_ARGS=" \
-server_name=server1 \
...
```

The preceding examples set up a symbolic name of `server1` to refer to the current instance of Impala. This symbolic name is used in the following ways:

- In an environment managed by Cloudera Manager, see Enabling Sentry for Impala in Cloudera Manager on page 91 for setting up Sentry for Impala in Cloud Manager. The values must be the same for both, so that Impala and Hive can share the privilege rules. Restart the Impala and Hive services after setting or changing this value.
- In an environment not managed by Cloudera Manager, you specify this value for the `sentry.hive.server` property in the `sentry-site.xml` configuration file for Hive, as well as in the `-server_name` option for `impalad`.

Now restart the `impalad` daemons on all the nodes.

Enabling Sentry for Impala in Cloudera Manager

To enable the Sentry service for Impala and Hive:

1. Navigate to the **Hive** cluster.
2. In the **Configuration** tab, select **Hive (Service-Wide)** under SCOPE and **Advanced** under CATEGORY.
3. In the **Sentry Service** field, type the Sentry service you specified in the Impala configuration. This is the server name to use when granting server level privileges
4. When using Sentry with the Hive Metastore, you can specify the list of users that are allowed to bypass Sentry Authorization in Hive Metastore. Select **Security** for CATEGORY in the Configuration tab, and specify the users in the **Bypass Sentry Authorization Users** field. These are usually service users that already ensure all activity has been authorized.
5. If in CDH 5, navigate to the **Impala** cluster, and perform the next two steps to disable the policy file-based authorization.
6. In the **Configuration** tab, select **Impala (Service-Wide)** under SCOPE and **Policy File Based Sentry** under **CATEGORY**.
7. Deselect the **Enable Sentry Authorization using Policy Files** parameter when using the Sentry service. Cloudera Manager throws a validation error in CDH 5 if you attempt to configure the Sentry service and policy file at the same time.
8. Restart Impala and Hive.

Using Impala with the Sentry Service (CDH 5.1 or higher only)

When you use the Sentry service, set up privileges through the GRANT and REVOKE statements in either Impala or Hive. Sentry privileges are automatically propagated to both services. Impala added the GRANT and REVOKE statements in CDH 5.2 / Impala 2.0.

For information about using the Impala GRANT and REVOKE statements, see GRANT Statement (CDH 5.2 or higher only) on page 273 and REVOKE Statement (CDH 5.2 or higher only) on page 293.

Changing Privileges

If you make a change to privileges in Sentry from outside of Impala, e.g. adding a user, removing a user, modifying privileges, there are two options to propagate the change:

- Use the `catalogd` flag, `--sentry_catalog_polling_frequency_s` to specify how often to do a Sentry refresh. The flag is set to 60 seconds by default.
- Run the `INVALIDATE METADATA` statement to force a Sentry refresh. `INVALIDATE METADATA` forces a Sentry refresh regardless of the `--sentry_catalog_polling_fequency_s` flag.

If you make a change to privileges within Impala, `INVALIDATE METADATA` is not required.



Warning: As `INVALIDATE METADATA` is an expensive operation, you should use it judiciously.

Granting Privileges on URI

URIs represent the file paths you specify as part of statements such as CREATE EXTERNAL TABLE and LOAD DATA. Typically, you specify what look like UNIX paths, but these locations can also be prefixed with hdfs:// to make clear that they are really URIs. To set privileges for a URI, specify the name of a directory, and the privilege applies to all the files in that directory and any directories underneath it.

URIs must start with hdfs://, s3a://, adl://, or file://. If a URI starts with an absolute path, the path will be appended to the default filesystem prefix. For example, if you specify:

```
GRANT ALL ON URI '/tmp';
```

The above statement effectively becomes the following where the default filesystem is HDFS.

```
GRANT ALL ON URI 'hdfs://localhost:20500/tmp';
```

When defining URIs for HDFS, you must also specify the NameNode. For example:

```
GRANT ALL ON URI file:///path/to/dir TO <role>
GRANT ALL ON URI hdfs://namenode:port/path/to/dir TO <role>
```



Warning: Because the NameNode host and port must be specified, it is strongly recommended that you use High Availability (HA). This ensures that the URI will remain constant even if the NameNode changes. For example:

```
GRANT ALL ON URI hdfs://ha-nn-uri/path/to/dir TO <role>
```

Examples of Setting up Authorization for Security Scenarios

The following examples show how to set up authorization to deal with various scenarios.

A User with No Privileges

If a user has no privileges at all, that user cannot access any schema objects in the system. The error messages do not disclose the names or existence of objects that the user is not authorized to read.

This is the experience you want a user to have if they somehow log into a system where they are not an authorized Impala user. Or in a real deployment, a user might have no privileges because they are not a member of any of the authorized groups.

Examples of Privileges for Administrative Users

In this example, the SQL statements grant the entire_server role all privileges on both the databases and URIs within the server.

```
CREATE ROLE entire_server;
GRANT ROLE entire_server TO GROUP admin_group;
GRANT ALL ON SERVER server1 TO ROLE entire_server;
```

A User with Privileges for Specific Databases and Tables

If a user has privileges for specific tables in specific databases, the user can access those things but nothing else. They can see the tables and their parent databases in the output of SHOW TABLES and SHOW DATABASES, USE the appropriate databases, and perform the relevant actions (SELECT and/or INSERT) based on the table privileges. To actually create a table requires the ALL privilege at the database level, so you might define separate roles for the user that sets up a schema and other users or applications that perform day-to-day operations on the tables.

```
CREATE ROLE one_database;
```

```
GRANT ROLE one_database TO GROUP admin_group;
GRANT ALL ON DATABASE db1 TO ROLE one_database;

CREATE ROLE instructor;
GRANT ROLE instructor TO GROUP trainers;
GRANT ALL ON TABLE db1.lesson TO ROLE instructor;

# This particular course is all about queries, so the students can SELECT but not INSERT
 or CREATE/DROP.
CREATE ROLE student;
GRANT ROLE student TO GROUP visitors;
GRANT SELECT ON TABLE db1.training TO ROLE student;
```

Privileges for Working with External Data Files

When data is being inserted through the `LOAD DATA` statement, or is referenced from an HDFS location outside the normal Impala database directories, the user also needs appropriate permissions on the URIs corresponding to those HDFS locations.

In this example:

- The `external_table` role can insert into and query the Impala table, `external_table.sample`.
- The `staging_dir` role can specify the HDFS path `/user/cloudera/external_data` with the `LOAD DATA` statement. When Impala queries or loads data files, it operates on all the files in that directory, not just a single file, so any Impala `LOCATION` parameters refer to a directory rather than an individual file.

```
CREATE ROLE external_table;
GRANT ROLE external_table TO GROUP cloudera;
GRANT ALL ON TABLE external_table.sample TO ROLE external_table;

CREATE ROLE staging_dir;
GRANT ROLE staging TO GROUP cloudera;
GRANT ALL ON URI 'hdfs://127.0.0.1:8020/user/cloudera/external_data' TO ROLE staging_dir;
```

Separating Administrator Responsibility from Read and Write Privileges

To create a database, you need the full privilege on that database while day-to-day operations on tables within that database can be performed with lower levels of privilege on specific table. Thus, you might set up separate roles for each database or application: an administrative one that could create or drop the database, and a user-level one that can access only the relevant tables.

In this example, the responsibilities are divided between users in 3 different groups:

- Members of the `supergroup` group have the `training_sysadmin` role and so can set up a database named `training`.
- Members of the `cloudera` group have the `instructor` role and so can create, insert into, and query any tables in the `training` database, but cannot create or drop the database itself.
- Members of the `visitor` group have the `student` role and so can query those tables in the `training` database.

```
CREATE ROLE training_sysadmin;
GRANT ROLE training_sysadmin TO GROUP supergroup;
GRANT ALL ON DATABASE training1 TO ROLE training_sysadmin;

CREATE ROLE instructor;
GRANT ROLE instructor TO GROUP cloudera;
GRANT ALL ON TABLE training1.course1 TO ROLE instructor;

CREATE ROLE visitor;
GRANT ROLE student TO GROUP visitor;
GRANT SELECT ON TABLE training1.course1 TO ROLE student;
```

Setting Up Schema Objects for a Secure Impala Deployment

In your role definitions, you must specify privileges at the level of individual databases and tables, or all databases or all tables within a database. To simplify the structure of these rules, plan ahead of time how to name your schema objects so that data with different authorization requirements is divided into separate databases.

If you are adding security on top of an existing Impala deployment, you can rename tables or even move them between databases using the ALTER TABLE statement.

The DEFAULT Database in a Secure Deployment

Because of the extra emphasis on granular access controls in a secure deployment, you should move any important or sensitive information out of the DEFAULT database into a named database. Sometimes you might need to give privileges on the DEFAULT database for administrative reasons, for example, as a place you can reliably specify with a USE statement when preparing to drop a database.

Impala Authentication

Authentication is the mechanism to ensure that only specified hosts and users can connect to Impala. It also verifies that when clients connect to Impala, they are connected to a legitimate server. This feature prevents spoofing such as ***impersonation*** (setting up a phony client system with the same account and group names as a legitimate user) and ***man-in-the-middle attacks*** (intercepting application requests before they reach Impala and eavesdropping on sensitive information in the requests or the results).

Impala supports authentication using either Kerberos or LDAP.



Note: Regardless of the authentication mechanism used, Impala always creates HDFS directories and data files owned by the same user (typically `impala`). To implement user-level access to different databases, tables, columns, partitions, and so on, use the Sentry authorization feature, as explained in Enabling Sentry Authorization for Impala on page 87.

Once you are finished setting up authentication, move on to authorization, which involves specifying what databases, tables, HDFS directories, and so on can be accessed by particular users when they connect through Impala. See Enabling Sentry Authorization for Impala on page 87 for details.

Enabling Kerberos Authentication for Impala

Impala supports Kerberos authentication. For background information on enabling Kerberos authentication, see Enabling Kerberos Authentication for CDH.

When using Impala in a managed environment, Cloudera Manager automatically completes Kerberos configuration.

In Impala 2.0 and later, `user()` returns the full Kerberos principal string, such as `user@example.com`, in a Kerberized environment.



Note: Regardless of the authentication mechanism used, Impala always creates HDFS directories and data files owned by the same user (typically `impala`). To implement user-level access to different databases, tables, columns, partitions, and so on, use the Sentry authorization feature, as explained in Enabling Sentry Authorization for Impala on page 87.

An alternative form of authentication you can use is LDAP, described in Enabling LDAP Authentication for Impala on page 96.

Requirements for Using Impala with Kerberos

Important:

- If you plan to use Impala in your cluster, you must configure your KDC to allow tickets to be renewed, and you must configure `krb5.conf` to request renewable tickets. Typically, you can do this by adding the `max_renewable_life` setting to your realm in `kdc.conf`, and by adding the `renew_lifetime` parameter to the `libdefaults` section of `krb5.conf`.

  For more information about renewable tickets, see the Kerberos documentation.

- The Impala Web UI does not support Kerberos authentication.
- You cannot use the Impala resource management feature on a cluster that has Kerberos authentication enabled.

Impala supports the Cloudera ODBC driver and the Kerberos interface provided. To use Kerberos through the ODBC driver, the host type must be set depending on the level of the ODBD driver:

- `SecImpala` for the ODBC 1.0 driver.
- `SecBeeswax` for the ODBC 1.2 driver.
- Blank for the ODBC 2.0 driver or higher, when connecting to a secure cluster.
- `HS2NoSasl` for the ODBC 2.0 driver or higher, when connecting to a non-secure cluster.

To enable Kerberos in the Impala shell, start the `impala-shell` command using the `-k` flag.

Configuring Impala to Support Kerberos Security

Enabling Kerberos authentication for Impala involves steps that can be summarized as follows:

- Creating service principals for Impala and the HTTP service. Principal names take the form: *`serviceName/fully.qualified.domain.name@KERBEROS.REALM`*
- Creating, merging, and distributing key tab files for these principals.
- Editing `/etc/default/impala` (in cluster not managed by Cloudera Manager), or editing the **Security** settings in the Cloudera Manager interface, to accommodate Kerberos authentication.

Enabling Kerberos for Impala with a Proxy Server

A common configuration for Impala with High Availability is to use a proxy server to submit requests to the actual `impalad` daemons on different hosts in the cluster. This configuration avoids connection problems in case of machine failure, because the proxy server can route new requests through one of the remaining hosts in the cluster. This configuration also helps with load balancing, because the additional overhead of being the “coordinator node” for each query is spread across multiple hosts.

Although you can set up a proxy server with or without Kerberos authentication, typically users set up a secure Kerberized configuration. For information about setting up a proxy server for Impala, including Kerberos-specific steps, see Using Impala through a Proxy for High Availability on page 72.

Enabling Impala Delegation for Kerberos Users

See Configuring Impala Delegation for Hue and BI Tools on page 99 for details about the delegation feature that lets certain users submit queries using the credentials of other users.

Using TLS/SSL with Business Intelligence Tools

You can use Kerberos authentication, TLS/SSL encryption, or both to secure connections from JDBC and ODBC applications to Impala. See Configuring Impala to Work with JDBC on page 725 and Configuring Impala to Work with ODBC on page 723 for details.

Prior to CDH 5.7 / Impala 2.5, the Hive JDBC driver did not support connections that use both Kerberos authentication and SSL encryption. If your cluster is running an older release that has this restriction, to use both of these security features with Impala through a JDBC application, use the Cloudera JDBC Connector as the JDBC driver.

Enabling Access to Internal Impala APIs for Kerberos Users

For applications that need direct access to Impala APIs, without going through the HiveServer2 or Beeswax interfaces, you can specify a list of Kerberos users who are allowed to call those APIs. By default, the `impala` and `hdfs` users are the only ones authorized for this kind of access. Any users not explicitly authorized through the `internal_principals_whitelist` configuration setting are blocked from accessing the APIs. This setting applies to all the Impala-related daemons, although currently it is primarily used for HDFS to control the behavior of the catalog server.

Mapping Kerberos Principals to Short Names for Impala

In CDH 5.8 / Impala 2.6 and higher, Impala recognizes the `auth_to_local` setting, specified through the Cloudera Manager setting **Additional Rules to Map Kerberos Principals to Short Names**. This feature is disabled by default, to avoid an unexpected change in security-related behavior. To enable it, select the **Use HDFS Rules to Map Kerberos Principals to Short Names** checkbox to enable the service-wide `--load_auth_to_local_rules` configuration setting. Then restart the Impala service. See Using Auth-to-Local Rules to Isolate Cluster Users for general information about this feature.

Enabling LDAP Authentication for Impala

Authentication is the process of allowing only specified named users to access the server (in this case, the Impala server). This feature is crucial for any production deployment, to prevent misuse, tampering, or excessive load on the server. Impala uses LDAP for authentication, verifying the credentials of each user who connects through `impala-shell`, Hue, a Business Intelligence tool, JDBC or ODBC application, etc.



Note: Regardless of the authentication mechanism used, Impala always creates HDFS directories and data files owned by the same user (typically `impala`). To implement user-level access to different databases, tables, columns, partitions, and so on, use the Sentry authorization feature, as explained in Enabling Sentry Authorization for Impala on page 87.

An alternative form of authentication you can use is Kerberos, described in Enabling Kerberos Authentication for Impala on page 94.

Requirements for Using Impala with LDAP

Authentication against LDAP servers is available in Impala 1.2.2 and higher. Impala 1.4.0 added the support for secure LDAP authentication through SSL and TLS.

The Impala LDAP support lets you use Impala with systems such as Active Directory that use LDAP behind the scenes.

Consideration for Connections Between Impala Components

Only the connections between clients and Impala can be authenticated by LDAP.

You must use the Kerberos authentication mechanism for connections between internal Impala components, such as between the `impalad`, `statestored`, and `catalogd` daemons. See Enabling Kerberos Authentication for Impala on page 94 on how to set up Kerberos for Impala.

Enabling LDAP in Command Line Interface

To enable LDAP authentication via a command line interface, start the `impalad` with the following startup options for:

`--enable_ldap_auth`

Enables LDAP-based authentication between the client and Impala.

--ldap_uri

Sets the URI of the LDAP server to use. Typically, the URI is prefixed with `ldap://`. You can specify secure SSL-based LDAP transport by using the prefix `ldaps://`. The URI can optionally specify the port, for example: `ldap://ldap_server.example.com:389` or `ldaps://ldap_server.example.com:636`. (389 and 636 are the default ports for non-SSL and SSL LDAP connections, respectively.)

Support for Custom Bind Strings

When Impala connects to LDAP it issues a bind call to the LDAP server to authenticate as the connected user. Impala clients, including the Impala shell, provide the short name of the user to Impala. This is necessary so that Impala can use Sentry for role-based access, which uses short names.

However, LDAP servers often require more complex, structured usernames for authentication. Impala supports three ways of transforming the short name (for example, `'henry'`) to a more complicated string. If necessary, specify one of the following configuration options when starting the `impalad` daemon.

--ldap_domain

Replaces the username with a string *username@ldap_domain*.

--ldap_baseDN

Replaces the username with a "distinguished name" (DN) of the form: `uid=`*userid*`,ldap_baseDN`. (This is equivalent to a Hive option).

--ldap_bind_pattern

This is the most general option, and replaces the username with the string *ldap_bind_pattern* where all instances of the string `#UID` are replaced with *userid*. For example, an `ldap_bind_pattern` of `"user=#UID,OU=foo,CN=bar"` with a username of `henry` will construct a bind name of `"user=henry,OU=foo,CN=bar"`.

The above options are mutually exclusive, and Impala does not start if more than one of these options are specified.

Secure LDAP Connections

To avoid sending credentials over the wire in cleartext, you must configure a secure connection between both the client and Impala, and between Impala and the LDAP server. The secure connection could use SSL or TLS.

Secure LDAP connections through SSL:

For SSL-enabled LDAP connections, specify a prefix of `ldaps://` instead of `ldap://`. Also, the default port for SSL-enabled LDAP connections is 636 instead of 389.

Secure LDAP connections through TLS:

TLS, the successor to the SSL protocol, is supported by most modern LDAP servers. Unlike SSL connections, TLS connections can be made on the same server port as non-TLS connections. To secure all connections using TLS, specify the following flags as startup options to the `impalad` daemon:

--ldap_tls

Tells Impala to start a TLS connection to the LDAP server, and to fail authentication if it cannot be done.

--ldap_ca_certificate="*/path/to/certificate/pem*"

Specifies the location of the certificate in standard `.PEM` format. Store this certificate on the local filesystem, in a location that only the `impala` user and other trusted users can read.

Enabling LDAP in Cloudera Manager

To enable LDAP authentication in Cloudera Manager:

1. In the Impala service, click the **Configuration** tab.
2. In the search box, type **ldap**.

3. Specify the values for the option fields. Each field lists the corresponding Impala startup flag. See the sections above for the corresponding flag if you need more information on a particular field.
4. Click **Save Changes**.
5. Restart the Impala service.

LDAP Authentication for impala-shell

To connect to Impala using LDAP authentication, you specify command-line options to the `impala-shell` command interpreter and enter the password when prompted.

`-l`

Enables LDAP authentication.

`-u`

Sets the user. Per Active Directory, the user is the short username, not the full LDAP distinguished name. If your LDAP settings include a search base, use the `--ldap_bind_pattern` on the `impalad` daemon to translate the short user name from `impala-shell` automatically to the fully qualified name.

`impala-shell` automatically prompts for the password.

See Configuring Impala to Work with JDBC on page 725 for the format to use with the JDBC connection string for servers using LDAP authentication.

Enabling LDAP for Impala in Hue

1. Go to the Hue service.
2. Click the **Configuration** tab.
3. Select **Scope** > **Hue Server**.
4. Select **Category** > **Advanced**.
5. Add the following properties to the **Hue Server Advanced Configuration Snippet (Safety Valve) for hue_safety_valve_server.ini** property.

```
[impala]
auth_username=<LDAP username of Hue user to be authenticated>
auth_password=<LDAP password of Hue user to be authenticated>
```

6. Click **Save Changes**.

Enabling Impala Delegation for LDAP Users

See Configuring Impala Delegation for Hue and BI Tools on page 99 for details about the delegation feature that lets certain users submit queries using the credentials of other users.

LDAP Restrictions for Impala

The LDAP support is preliminary. It currently has only been tested against Active Directory.

Using Multiple Authentication Methods with Impala

Impala 2.0 and later automatically handles both Kerberos and LDAP authentication. Each `impalad` daemon can accept both Kerberos and LDAP requests through the same port. No special actions need to be taken if some users authenticate through Kerberos and some through LDAP.

Prior to Impala 2.0, you had to configure each `impalad` to listen on a specific port depending on the kind of authentication, then configure your network load balancer to forward each kind of request to a DataNode that was set up with the appropriate authentication type. Once the initial request was made using either Kerberos or LDAP authentication, Impala automatically handled the process of coordinating the work across multiple nodes and transmitting intermediate results back to the coordinator node.

Configuring Impala Delegation for Hue and BI Tools

When users submit Impala queries through a separate application, such as Hue or a business intelligence tool, typically all requests are treated as coming from the same user. In Impala 1.2 and higher, Impala supports "delegation" where users whose names you specify can delegate the execution of a query to another user. The query runs with the privileges of the delegated user, not the original authenticated user.

Starting in Impala 3.1 and higher, you can delegate using groups. Instead of listing a large number of delegated users, you can create a group of those users and specify the delegated group name in the Impalad startup option. The client sends the delegated user name, and Impala performs an authorization to see if the delegated user belongs to a delegated group.

The name of the delegated user is passed using the HiveServer2 protocol configuration property `impala.doas.user` when the client connects to Impala.

Currently, the delegation feature is available only for Impala queries submitted through application interfaces such as Hue and BI tools. For example, Impala cannot issue queries using the privileges of the HDFS user.

The delegation feature is enabled by the startup options for `impalad`: `--authorized_proxy_user_config` and `--authorized_proxy_group_config`. See Enabling Delegation in Cloudera Manager on page 100 for enabling delegation in Cloudera Manager.

- The list of authorized users/groups are delimited with `;`
- The list of delegated users/groups are delimited with `,` by default.
- Use the `--authorized_proxy_user_config_delimiter` startup option to override the default user delimiter (the comma character) to another character.
- Use the `--authorized_proxy_group_config_delimiter` startup option to override the default group delimiter (the comma character) to another character.
- A wildcard (`*`) is supported to delegated to any users or any groups, e.g. `--authorized_proxy_group_config=hue=*`. Make sure to use single quotes or escape characters to ensure that any `*` characters do not undergo wildcard expansion when specified in command-line arguments.

When you start Impala with the `--authorized_proxy_user_config=`*`authenticated_user=delegated_user`* or `--authorized_proxy_group_config=`*`authenticated_user=delegated_group`* option:

- Authentication is based on the user on the left hand side (*authenticated_user*).
- Authorization is based on the right hand side user(s) or group(s) (*delegated_user*, *delegated_group*).
- When opening a client connection, the client must provide a delegated username via the HiveServer2 protocol property,`impala.doas.user` or `DelegationUID`.
- It is not necessary for *authenticated_user* to have the permission to access/edit files.
- It is not necessary for the delegated users to have access to the service via Kerberos.
- *delegated_user* and *delegated_group* must exist in the OS.
- For group delegation, use the JNI-based mapping providers for group delegation, such as JniBasedUnixGroupsMappingWithFallback and JniBasedUnixGroupsNetgroupMappingWithFallback.
- ShellBasedUnixGroupsNetgroupMapping and ShellBasedUnixGroupsMapping Hadoop group mapping providers are not supported in Impala group delegation.
- In Impala, `user()` returns *authenticated_user* and `effective_user()` returns the delegated user that the client specified.

The user or group delegation process works as follows:

1. The Impalad daemon starts with one or both of the following options:
   - `--authorized_proxy_user_config=`*`authenticated_user=delegated_user`*
   - `--authorized_proxy_group_config=`*`authenticated_user=delegated_group`*
2. A client connects to Impala via the HiveServer2 protocol with the `impala.doas.user` configuration property, e.g. connected user is *authenticated_user* with `impala.doas.user=`*`delegated_user`*.
3. The client user *`authenticated_user`* sends a request to Impala as the delegated user *delegated_user*.
4. Impala checks authorization:

- In user delegation, Impala checks if *delegated_user* is in the list of authorized delegate users for the user *authenticated_user*.
- In group delegate, Impala checks if *delegated_user* belongs to one of the delegated groups for the user *authenticated_user*, *delegated_group* in this example.

5. If the user is an authorized delegated user for *authenticated_user*, the request is executed as the delegate user *delegated_user*.

See Modifying Impala Startup Options for details about adding or changing `impalad` startup options.

See this blog post for background information about the delegation capability in HiveServer2.

To set up authentication for the delegated users:

- On the server side, configure either user/password authentication through LDAP, or Kerberos authentication, for all the delegated users. See Enabling LDAP Authentication for Impala on page 96 or Enabling Kerberos Authentication for Impala on page 94 for details.
- On the client side, to learn how to enable delegation, consult the documentation for the ODBC driver you are using.

Enabling Delegation in Cloudera Manager

To enable delegation in Cloudera Manager:

1. Navigate to **Clusters** > **Impala** > **Configuration** > **Policy File-Based Sentry**.
2. In the **Proxy User Configuration** field, type the a semicolon-separated list of key=value pairs of authorized proxy users to the user(s) they can impersonate. The list of delegated short user names are delimited with a comma, e.g. **hue=user1, user2**.

   The short user names should be specified. The user names are case-sensitive.

3. In the **Proxy Group Configuration** field, type the a semicolon-separated list of key=value pairs of authorized proxy users to the group(s) they can impersonate. The list of delegated groups are delimited with a comma, e.g. **hue=group1, group2**.

   The group names are case-sensitive.

4. Click **Save Changes** and restart the Impala service.

Impala SQL Language Reference

Impala uses SQL as its query language. To protect user investment in skills development and query design, Impala provides a high degree of compatibility with the Hive Query Language (HiveQL):

- Because Impala uses the same metadata store as Hive to record information about table structure and properties, Impala can access tables defined through the native Impala `CREATE TABLE` command, or tables created using the Hive data definition language (DDL).
- Impala supports data manipulation (DML) statements similar to the DML component of HiveQL.
- Impala provides many built-in functions with the same names and parameter types as their HiveQL equivalents.

Impala supports most of the same statements and clauses as HiveQL, including, but not limited to `JOIN`, `AGGREGATE`, `DISTINCT`, `UNION ALL`, `ORDER BY`, `LIMIT` and (uncorrelated) subquery in the `FROM` clause. Impala also supports `INSERT INTO` and `INSERT OVERWRITE`.

Impala supports data types with the same names and semantics as the equivalent Hive data types: `STRING`, `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `FLOAT`, `DOUBLE`, `BOOLEAN`, `STRING`, `TIMESTAMP`.

For full details about Impala SQL syntax and semantics, see Impala SQL Statements on page 202.

Most HiveQL `SELECT` and `INSERT` statements run unmodified with Impala. For information about Hive syntax not available in Impala, see SQL Differences Between Impala and Hive on page 541.

For a list of the built-in functions available in Impala queries, see Impala Built-In Functions on page 391.

Comments

Impala supports the familiar styles of SQL comments:

- All text from a `--` sequence to the end of the line is considered a comment and ignored. This type of comment can occur on a single line by itself, or after all or part of a statement.
- All text from a `/*` sequence to the next `*/` sequence is considered a comment and ignored. This type of comment can stretch over multiple lines. This type of comment can occur on one or more lines by itself, in the middle of a statement, or before or after a statement.

For example:

```
-- This line is a comment about a table.
create table ...;

/*
This is a multi-line comment about a query.
*/
select ...;

select * from t /* This is an embedded comment about a query. */ where ...;

select * from t -- This is a trailing comment within a multi-line command.
where ...;
```

Data Types

Impala supports a set of data types that you can use for table columns, expression values, and function arguments and return values.



Note: Currently, Impala supports only scalar types, not composite or nested types. Accessing a table containing any columns with unsupported types causes an error.

For the notation to write literals of each of these data types, see Literals on page 167.

Impala supports a limited set of implicit casts to avoid undesired results from unexpected casting behavior.

- Impala does not implicitly cast between string and numeric or Boolean types. Always use CAST( ) for these conversions.
- Impala does perform implicit casts among the numeric types, when going from a smaller or less precise type to a larger or more precise one. For example, Impala will implicitly convert a SMALLINT to a BIGINT or FLOAT, but to convert from DOUBLE to FLOAT or INT to TINYINT requires a call to CAST( ) in the query.
- Impala does perform implicit casts from STRING to TIMESTAMP. Impala has a restricted set of literal formats for the TIMESTAMP data type and the FROM_UNIXTIME( ) format string; see TIMESTAMP Data Type on page 130 for details.

See the topics under this section for full details on implicit and explicit casting for each data type, and see Impala Type Conversion Functions on page 423 for details about the CAST( ) function.

ARRAY Complex Type (CDH 5.5 or higher only)

A complex data type that can represent an arbitrary number of ordered elements. The elements can be scalars or another complex type (ARRAY, STRUCT, or MAP).

Syntax:

```
column_name ARRAY < type >

type ::= primitive_type | complex_type
```

Usage notes:

Because complex types are often used in combination, for example an ARRAY of STRUCT elements, if you are unfamiliar with the Impala complex types, start with Complex Types (CDH 5.5 or higher only) on page 139 for background information and usage examples.

The elements of the array have no names. You refer to the value of the array item using the ITEM pseudocolumn, or its position in the array with the POS pseudocolumn. See ITEM and POS Pseudocolumns on page 153 for information about these pseudocolumns.

Each row can have a different number of elements (including none) in the array for that row.

When an array contains items of scalar types, you can use aggregation functions on the array elements without using join notation. For example, you can find the COUNT( ), AVG( ), SUM( ), and so on of numeric array elements, or the MAX( ) and MIN( ) of any scalar array elements by referring to *table_name*.*array_column* in the FROM clause of the query. When you need to cross-reference values from the array with scalar values from the same row, such as by including a GROUP BY clause to produce a separate aggregated result for each row, then the join clause is required.

A common usage pattern with complex types is to have an array as the top-level type for the column: an array of structs, an array of maps, or an array of arrays. For example, you can model a denormalized table by creating a column that is an ARRAY of STRUCT elements; each item in the array represents a row from a table that would normally be used in a join query. This kind of data structure lets you essentially denormalize tables by associating multiple rows from one table with the matching row in another table.

You typically do not create more than one top-level ARRAY column, because if there is some relationship between the elements of multiple arrays, it is convenient to model the data as an array of another complex type element (either STRUCT or MAP).

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

Added in: CDH 5.5.0 / Impala 2.3.0

Restrictions:

- Columns with this data type can only be used in tables or partitions with the Parquet file format.
- Columns with this data type cannot be used as partition key columns in a partitioned table.
- The COMPUTE STATS statement does not produce any statistics for columns of this data type.
- The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.
- See Limitations and Restrictions for Complex Types on page 143 for a full list of limitations and associated guidelines about complex type columns.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Examples:



Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types on page 160 for the table definitions.

The following example shows how to construct a table with various kinds of ARRAY columns, both at the top level and nested within other complex types. Whenever the ARRAY consists of a scalar value, such as in the PETS column or the CHILDREN field, you can see that future expansion is limited. For example, you could not easily evolve the schema to record the kind of pet or the child's birthday alongside the name. Therefore, it is more common to use an ARRAY whose elements are of STRUCT type, to associate multiple fields with each array element.



Note: Practice the CREATE TABLE and query notation for complex type columns using empty tables, until you can visualize a complex data structure and construct corresponding SQL statements reliably.

```
CREATE TABLE array_demo
(
  id BIGINT,
  name STRING,
-- An ARRAY of scalar type as a top-level column.
  pets ARRAY <STRING>,

-- An ARRAY with elements of complex type (STRUCT).
  places_lived ARRAY < STRUCT <
    place: STRING,
    start_year: INT
  >>,

-- An ARRAY as a field (CHILDREN) within a STRUCT.
-- (The STRUCT is inside another ARRAY, because it is rare
-- for a STRUCT to be a top-level column.)
  marriages ARRAY < STRUCT <
    spouse: STRING,
    children: ARRAY <STRING>
  >>,

-- An ARRAY as the value part of a MAP.
-- The first MAP field (the key) would be a value such as
-- 'Parent' or 'Grandparent', and the corresponding array would
-- represent 2 parents, 4 grandparents, and so on.
  ancestors MAP < STRING, ARRAY <STRING> >
)
STORED AS PARQUET;
```

The following example shows how to examine the structure of a table containing one or more ARRAY columns by using the DESCRIBE statement. You can visualize each ARRAY as its own two-column table, with columns ITEM and POS.

```
DESCRIBE array_demo;
+--------------+--------------------------+
| name         | type                     |
+--------------+--------------------------+
| id           | bigint                   |
| name         | string                   |
| pets         | array<string>            |
| marriages    | array<struct<            |
|              |   spouse:string,         |
|              |   children:array<string> |
|              | >>                       |
| places_lived | array<struct<            |
|              |   place:string,          |
|              |   start_year:int         |
|              | >>                       |
| ancestors    | map<string,array<string>> |
+--------------+--------------------------+

DESCRIBE array_demo.pets;
+------+--------+
| name | type   |
+------+--------+
| item | string |
| pos  | bigint |
+------+--------+

DESCRIBE array_demo.marriages;
+------+--------------------------+
| name | type                     |
+------+--------------------------+
| item | struct<                  |
|      |   spouse:string,         |
|      |   children:array<string> |
|      | >                        |
| pos  | bigint                   |
+------+--------------------------+

DESCRIBE array_demo.places_lived;
+------+------------------+
| name | type             |
+------+------------------+
| item | struct<          |
|      |   place:string,  |
|      |   start_year:int |
|      | >                |
| pos  | bigint           |
+------+------------------+

DESCRIBE array_demo.ancestors;
+-------+---------------+
| name  | type          |
+-------+---------------+
| key   | string        |
| value | array<string> |
+-------+---------------+
```

The following example shows queries involving ARRAY columns containing elements of scalar or complex types. You "unpack" each ARRAY column by referring to it in a join query, as if it were a separate table with ITEM and POS columns. If the array element is a scalar type, you refer to its value using the ITEM pseudocolumn. If the array element is a STRUCT, you refer to the STRUCT fields using dot notation and the field names. If the array element is another ARRAY or a MAP, you use another level of join to unpack the nested collection elements.

```
-- Array of scalar values.
-- Each array element represents a single string, plus we know its position in the array.
SELECT id, name, pets.pos, pets.item FROM array_demo, array_demo.pets;

-- Array of structs.
```

```
-- Now each array element has named fields, possibly of different types.
-- You can consider an ARRAY of STRUCT to represent a table inside another table.
SELECT id, name, places_lived.pos, places_lived.item.place, places_lived.item.start_year
FROM array_demo, array_demo.places_lived;

-- The .ITEM name is optional for array elements that are structs.
-- The following query is equivalent to the previous one, with .ITEM
-- removed from the column references.
SELECT id, name, places_lived.pos, places_lived.place, places_lived.start_year
  FROM array_demo, array_demo.places_lived;

-- To filter specific items from the array, do comparisons against the .POS or .ITEM
-- pseudocolumns, or names of struct fields, in the WHERE clause.
SELECT id, name, pets.item FROM array_demo, array_demo.pets
  WHERE pets.pos in (0, 1, 3);

SELECT id, name, pets.item FROM array_demo, array_demo.pets
  WHERE pets.item LIKE 'Mr. %';

SELECT id, name, places_lived.pos, places_lived.place, places_lived.start_year
  FROM array_demo, array_demo.places_lived
WHERE places_lived.place like '%California%';
```

Related information:

Complex Types (CDH 5.5 or higher only) on page 139, STRUCT Complex Type (CDH 5.5 or higher only) on page 124, MAP Complex Type (CDH 5.5 or higher only) on page 118

BIGINT Data Type

An 8-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name BIGINT
```

Range: -9223372036854775808 .. 9223372036854775807. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a floating-point type (FLOAT or DOUBLE) automatically. Use CAST( ) to convert to TINYINT, SMALLINT, INT, STRING, or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting --use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Examples:

```
CREATE TABLE t1 (x BIGINT);
SELECT CAST(1000 AS BIGINT);
```

Usage notes:

BIGINT is a convenient type to use for column declarations because you can use any kind of integer values in INSERT statements and they are promoted to BIGINT where necessary. However, BIGINT also requires the most bytes of any integer type on disk and in memory, meaning your queries are not as efficient and scalable as possible if you overuse this type. Therefore, prefer to use the smallest integer type with sufficient range to hold all input values, and CAST( ) when necessary to the appropriate type.

For a convenient and automated way to check the bounds of the BIGINT type, call the functions MIN_BIGINT( ) and MAX_BIGINT( ).

If an integer value is too large to be represented as a BIGINT, use a DECIMAL instead with sufficient digits of precision.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Partitioning: Prefer to use this type for a partition key column. Impala can process the numeric type more efficiently than a STRING representation of the value.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as an 8-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Related information:

Numeric Literals on page 167, TINYINT Data Type on page 136, SMALLINT Data Type on page 122, INT Data Type on page 117, BIGINT Data Type on page 105, DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only) on page 109, Impala Mathematical Functions on page 397

BOOLEAN Data Type

A data type used in CREATE TABLE and ALTER TABLE statements, representing a single true/false choice.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name BOOLEAN
```

Range: TRUE or FALSE. Do not use quotation marks around the TRUE and FALSE literal values. You can write the literal values in uppercase, lowercase, or mixed case. The values queried from a table are always returned in lowercase, `true` or `false`.

Conversions: Impala does not automatically convert any other type to BOOLEAN. All conversions must use an explicit call to the CAST( ) function.

You can use CAST( ) to convert any integer or floating-point type to BOOLEAN: a value of 0 represents `false`, and any non-zero value is converted to `true`.

```
SELECT CAST(42 AS BOOLEAN) AS nonzero_int, CAST(99.44 AS BOOLEAN) AS nonzero_decimal,
  CAST(000 AS BOOLEAN) AS zero_int, CAST(0.0 AS BOOLEAN) AS zero_decimal;
+-------------+-----------------+----------+--------------+
| nonzero_int | nonzero_decimal | zero_int | zero_decimal |
+-------------+-----------------+----------+--------------+
| true        | true            | false    | false        |
+-------------+-----------------+----------+--------------+
```

When you cast the opposite way, from BOOLEAN to a numeric type, the result becomes either 1 or 0:

```
SELECT CAST(true AS INT) AS true_int, CAST(true AS DOUBLE) AS true_double,
  CAST(false AS INT) AS false_int, CAST(false AS DOUBLE) AS false_double;
+----------+-------------+-----------+--------------+
| true_int | true_double | false_int | false_double |
+----------+-------------+-----------+--------------+
| 1        | 1           | 0         | 0            |
+----------+-------------+-----------+--------------+
```

You can cast DECIMAL values to BOOLEAN, with the same treatment of zero and non-zero values as the other numeric types. You cannot cast a BOOLEAN to a DECIMAL.

You cannot cast a STRING value to BOOLEAN, although you can cast a BOOLEAN value to STRING, returning `'1'` for `true` values and `'0'` for `false` values.

Although you can cast a TIMESTAMP to a BOOLEAN or a BOOLEAN to a TIMESTAMP, the results are unlikely to be useful. Any non-zero TIMESTAMP (that is, any value other than 1970-01-01 00:00:00) becomes TRUE when converted to BOOLEAN, while 1970-01-01 00:00:00 becomes FALSE. A value of FALSE becomes 1970-01-01 00:00:00 when converted to BOOLEAN, and TRUE becomes one second past this epoch date, that is, 1970-01-01 00:00:01.

NULL considerations: An expression of this type produces a NULL value if any argument of the expression is NULL.

Partitioning:

Do not use a BOOLEAN column as a partition key. Although you can create such a table, subsequent operations produce errors:

```
[localhost:21000] > create table truth_table (assertion string) partitioned by (truth
boolean);
[localhost:21000] > insert into truth_table values ('Pigs can fly',false);
ERROR: AnalysisException: INSERT into table with BOOLEAN partition column (truth) is
not supported: partitioning.truth_table
```

Examples:

```
SELECT 1 < 2;
SELECT 2 = 5;
SELECT 100 < NULL, 100 > NULL;
CREATE TABLE assertions (claim STRING, really BOOLEAN);
INSERT INTO assertions VALUES
  ("1 is less than 2", 1 < 2),
  ("2 is the same as 5", 2 = 5),
  ("Grass is green", true),
  ("The moon is made of green cheese", false);
SELECT claim FROM assertions WHERE really = TRUE;
```

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Kudu considerations:

Currently, the data types BOOLEAN, FLOAT, and DOUBLE cannot be used for primary key columns in Kudu tables.

Related information: Boolean Literals on page 169, SQL Operators on page 171, Impala Conditional Functions on page 457

CHAR Data Type (CDH 5.2 or higher only)

A fixed-length character type, padded with trailing spaces if necessary to achieve the specified length. If values are longer than the specified length, Impala truncates any trailing characters.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name CHAR(length)
```

The maximum *length* you can specify is 255.

Semantics of trailing spaces:

- When you store a CHAR value shorter than the specified length in a table, queries return the value padded with trailing spaces if necessary; the resulting value has the same length as specified in the column definition.
- Leading spaces in CHAR are preserved within the data file.

- If you store a CHAR value containing trailing spaces in a table, those trailing spaces are not stored in the data file. When the value is retrieved by a query, the result could have a different number of trailing spaces. That is, the value includes however many spaces are needed to pad it to the specified length of the column.
- If you compare two CHAR values that differ only in the number of trailing spaces, those values are considered identical.
- When comparing or processing CHAR values:
  - CAST( ) truncates any longer string to fit within the defined length. For example:

```
SELECT CAST('x' AS CHAR(4)) = CAST('x        ' AS CHAR(4)); -- Returns TRUE.
```

  - If a CHAR value is shorter than the specified length, it is padded on the right with spaces until it matches the specified length.
  - CHAR_LENGTH( ) returns the length including any trailing spaces.
  - LENGTH( ) returns the length excluding trailing spaces.
  - CONCAT( ) returns the length including trailing spaces.

Partitioning: This type can be used for partition key columns. Because of the efficiency advantage of numeric values over character-based values, if the partition key is a string representation of a number, prefer to use an integer type with sufficient range (INT, BIGINT, and so on) where practical.

HBase considerations: This data type cannot be used with HBase tables.

Parquet considerations:

- This type can be read from and written to Parquet files.
- There is no requirement for a particular level of Parquet.
- Parquet files generated by Impala and containing this type can be freely interchanged with other components such as Hive and MapReduce.
- Any trailing spaces, whether implicitly or explicitly specified, are not written to the Parquet data files.
- Parquet data files might contain values that are longer than allowed by the CHAR(*n*) length limit. Impala ignores any extra trailing characters when it processes those values during a query.

Text table considerations:

Text data files might contain values that are longer than allowed for a particular CHAR(*n*) column. Any extra trailing characters are ignored when Impala processes those values during a query. Text data files can also contain values that are shorter than the defined length limit, and Impala pads them with trailing spaces up to the specified length. Any text data files produced by Impala INSERT statements do not include any trailing blanks for CHAR columns.

Avro considerations:

The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In CDH 5.7 / Impala 2.5 and higher, Impala truncates CHAR and VARCHAR values in Avro tables to (2**31)-1 bytes. If a query encounters a STRING value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Compatibility:

This type is available using CDH 5.2 / Impala 2.0 or higher.

Some other database systems make the length specification optional. For Impala, the length is required.

Internal details: Represented in memory as a byte array with the same size as the length specification. Values that are shorter than the specified length are padded on the right with trailing spaces.

Added in: CDH 5.2.0 / Impala 2.0.0

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

UDF considerations: This type cannot be used for the argument or return type of a user-defined function (UDF) or user-defined aggregate function (UDA).

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Performance consideration:

The CHAR type currently does not have the Impala Codegen support, and we recommend using VARCHAR or STRING over CHAR as the performance gain of Codegen outweighs the benefits of fixed width CHAR.

Restrictions:

Because the blank-padding behavior requires allocating the maximum length for each value in memory, for scalability reasons, you should avoid declaring CHAR columns that are much longer than typical values in that column.

All data in CHAR and VARCHAR columns must be in a character encoding that is compatible with UTF-8. If you have binary data from another database system (that is, a BLOB type), use a STRING column to hold it.

When an expression compares a CHAR with a STRING or VARCHAR, the CHAR value is implicitly converted to STRING first, with trailing spaces preserved.

This behavior differs from other popular database systems. To get the expected result of TRUE, cast the expressions on both sides to CHAR values of the appropriate length. For example:

```
SELECT CAST("foo  " AS CHAR(5)) = CAST('foo' AS CHAR(3)); -- Returns TRUE.
```

This behavior is subject to change in future releases.

Related information:

STRING Data Type on page 123, VARCHAR Data Type (CDH 5.2 or higher only) on page 137, String Literals on page 168, Impala String Functions on page 462

DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only)

The DECIMAL data type is a numeric data type with fixed scale and precision.

The data type is useful for storing and doing operations on precise decimal values.

Syntax:

```
DECIMAL[(precision[, scale])]
```

Precision:

precision represents the total number of digits that can be represented regardless of the location of the decimal point.

This value must be between 1 and 38, specified as an integer literal.

The default precision is 9.

Scale:

scale represents the number of fractional digits.

This value must be less than or equal to the precision, specified as an integer literal.

The default scale is 0.

When the precision and the scale are omitted, a DECIMAL is treated as DECIMAL(9, 0).

Range:

The range of DECIMAL type is -10^38 +1 through 10^38 –1.

The largest value is represented by DECIMAL(38, 0).

The most precise fractional value (between 0 and 1, or 0 and -1) is represented by DECIMAL(38, 38), with 38 digits to the right of the decimal point. The value closest to 0 would be .0000...1 (37 zeros and the final 1). The value closest to 1 would be .999... (9 repeated 38 times).

Memory and disk storage:

Only the precision determines the storage size for DECIMAL values, and the scale setting has no effect on the storage size. The following table describes the in-memory storage once the values are loaded into memory.

Precision	In-memory Storage
1 - 9	4 bytes
10 - 18	8 bytes
19 - 38	16 bytes

The on-disk representation varies depending on the file format of the table.

Text, RCFile, and SequenceFile tables use ASCII-based formats as below:

- Leading zeros are not stored.
- Trailing zeros are stored.
- Each DECIMAL value takes up as many bytes as the precision of the value, plus:
  - One extra byte if the decimal point is present.
  - One extra byte for negative values.

Parquet and Avro tables use binary formats and offer more compact storage for DECIMAL values. In these tables, Impala stores each value in fewer bytes where possible depending on the precision specified for the DECIMAL column. To conserve space in large tables, use the smallest-precision DECIMAL type.

Precision and scale in arithmetic operations:

For all arithmetic operations, the resulting precision is at most 38.

If the resulting precision would be greater than 38, Impala truncates the result from the back, but keeps at least 6 fractional digits in scale and rounds.

For example, `DECIMAL(38, 20) * DECIMAL(38, 20)` returns `DECIMAL(38, 6)`. According to the table below, the resulting precision and scale would be `(77, 40)`, but they are higher than the maximum precision and scale for DECIMAL. So, Impala sets the precision to the maximum allowed 38, and truncates the scale to 6.

When you use DECIMAL values in arithmetic operations, the precision and scale of the result value are determined as follows. For better readability, the following terms are used in the table below:

- P1, P2: Input precisions
- S1, S2: Input scales
- L1, L2: Leading digits in input DECIMALs, i.e., L1 = P1 - S1 and L2 = P2 - S2

Operation	Resulting Precision	Resulting Scale
Addition and Subtraction	max (L1, L2) + max (S1, S2) + 1 1 is for carry-over.	max (S1, S2)
Multiplication	P1 + P2 + 1	S1 + S2
Division	L1 + S2 + max (S1 + P2 + 1, 6)	max (S1 + P2 + 1, 6)
Modulo	min (L1, L2) + max (S1, S2)	max (S1, S2)

Precision and scale in functions:

When you use DECIMAL values in built-in functions, the precision and scale of the result value are determined as follows:

- The result of the SUM aggregate function on a DECIMAL value is:
  - Precision: 38
  - Scale: The same scale as the input column
- The result of AVG aggregate function on a DECIMAL value is:
  - Precision: 38
  - Scale: max(Scale of input column, 6)

Implicit conversions in DECIMAL assignments:

Impala enforces strict conversion rules in decimal assignments like in INSERT and UNION statements, or in functions like COALESCE.

If there is not enough precision and scale in the destination, Impala fails with an error.

Impala performs implicit conversions between DECIMAL and other numeric types as below:

- DECIMAL is implicitly converted to DOUBLE or FLOAT when necessary even with a loss of precision. It can be necessary, for example when inserting a DECIMAL value into a DOUBLE column. For example:

```
CREATE TABLE flt(c FLOAT);
INSERT INTO flt SELECT CAST(1e37 AS DECIMAL(38, 0));
SELECT CAST(c AS DECIMAL(38, 0)) FROM flt;

Result: 99999999338158125107115063762579619 84
```

  The result has a loss of information due to implicit casting. This is why we discourage using the DOUBLE and FLOAT types in general.

- DOUBLE and FLOAT cannot be implicitly converted to DECIMAL. An error is returned.
- DECIMAL is implicitly converted to DECIMAL if all digits fit in the resulting DECIMAL.

  For example, the following query returns an error because the resulting type that guarantees that all digits fit cannot be determined .

```
SELECT GREATEST (CAST(1 AS DECIMAL(38, 0)), CAST(2 AS DECIMAL(38, 37)));
```

- Integer values can be implicitly converted to DECIMAL when there is enough room in the DECIMAL to guarantee that all digits fit. The integer types require the following numbers of digits to the left of the decimal point when converted to DECIMAL:
  - BIGINT: 19 digits
  - INT: 10 digits
  - SMALLINT: 5 digits
  - TINYINT: 3 digits

  For example:

```
CREATE TABLE decimals_10_8 (x DECIMAL(10, 8));
INSERT INTO decimals_10_8 VALUES (CAST(1 AS TINYINT));
```

  The above INSERT statement fails because TINYINT requires room for 3 digits to the left of the decimal point in the DECIMAL.

```
CREATE TABLE decimals_11_8(x DECIMAL(11, 8));
INSERT INTO decimals_11_8 VALUES (CAST(1 AS TINYINT));
```

The above INSERT statement succeeds because there is enough room for 3 digits to the left of the decimal point that TINYINT requires.

In UNION, the resulting precision and scales are determined as follows.

- Precision: max (L1, L2) + max (S1, S2)

  If the resulting type does not fit in the DECIMAL type, an error is returned. See the first example below.

- Scale: max (S1, S2)

Examples for UNION:

- DECIMAL(20, 0) UNION DECIMAL(20, 20) would require a DECIMAL(40, 20) to fit all the digits. Since this is larger than the max precision for DECIMAL, Impala returns an error. One way to fix the error is to cast both operands to the desired type, for example DECIMAL(38, 18).
- DECIMAL(20, 2) UNION DECIMAL(8, 6) returns DECIMAL(24, 6).
- INT UNION DECIMAL(9, 4) returns DECIMAL(14, 4).

  INT has the precision 10 and the scale 0, so it is treated as DECIMAL(10, 0) UNION DECIMAL(9. 4).

Casting between DECIMAL and other data types:

To avoid potential conversion errors, use CAST to explicitly convert between DECIMAL and other types in decimal assignments like in INSERT and UNION statements, or in functions like COALESCE:

- You can cast the following types to DECIMAL: FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING
- You can cast DECIMAL to the following types: FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING, BOOLEAN, TIMESTAMP

Impala performs CAST between DECIMAL and other numeric types as below:

- Precision: If you cast a value with bigger precision than the precision of the destination type, Impala returns an error. For example, CAST(123456 AS DECIMAL(3,0)) returns an error because all digits do not fit into DECIMAL(3, 0)
- Scale: If you cast a value with more fractional digits than the scale of the destination type, the fractional digits are rounded. For example, CAST(1.239 AS DECIMAL(3, 2)) returns 1.24.

Casting STRING to DECIMAL:

You can cast STRING of numeric characters in columns, literals, or expressions to DECIMAL as long as number fits within the specified target DECIMAL type without overflow.

- If scale in STRING > scale in DECIMAL, the fractional digits are rounded to the DECIMAL scale.

  For example, CAST('98.678912' AS DECIMAL(15, 1)) returns 98.7.

- If # leading digits in STRING > # leading digits in DECIMAL, an error is returned.

  For example, CAST('123.45' AS DECIMAL(2, 2)) returns an error.

Exponential notation is supported when casting from STRING.

For example, CAST('1.0e6' AS DECIMAL(32, 0)) returns 1000000.

Casting any non-numeric value, such as 'ABC' to the DECIMAL type returns an error.

Casting DECIMAL to TIMESTAMP:

Casting a DECIMAL value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970).

DECIMAL vs FLOAT consideration:

The FLOAT and DOUBLE types can cause problems or unexpected behavior due to inability to precisely represent certain fractional values, for example dollar and cents values for currency. You might find output values slightly different than you inserted, equality tests that do not match precisely, or unexpected values for GROUP BY columns. The DECIMAL type can help reduce unexpected behavior and rounding errors, but at the expense of some performance overhead for assignments and comparisons.

Literals and expressions:

- Numeric literals without a decimal point
  - The literals are treated as the smallest integer that would fit the literal. For example, 111 is a TINYINT, and 1111 is a SMALLINT.
  - Large literals that do not fit into any integer type are treated as DECIMAL.
  - The literals too large to fit into a DECIMAL(38, 0) are treated as DOUBLE.
- Numeric literals with a decimal point
  - The literal with less than 38 digits are treated as DECIMAL.
  - The literals with 38 or more digits are treated as a DOUBLE.
- Exponential notation is supported in DECIMAL literals.
- To represent a very large or precise DECIMAL value as a literal, for example one that contains more digits than can be represented by a BIGINT literal, use a quoted string or a floating-point value for the number and CAST the string to the desired DECIMAL type.

  For example: CAST('999999999999999999999999999999' AS DECIMAL(38, 5)))

File format considerations:

The DECIMAL data type can be stored in any of the file formats supported by Impala:

- Impala can query Avro, RCFile, or SequenceFile tables that contain DECIMAL columns, created by other Hadoop components.
- Impala can query and insert into Kudu tables that contain DECIMAL columns. Kudu supports the DECIMAL type in CDH 6.1 and higher.
- The DECIMAL data type is fully compatible with HBase tables.
- The DECIMAL data type is fully compatible with Parquet tables.
- Values of the DECIMAL data type are potentially larger in text tables than in tables using Parquet or other binary formats.

UDF consideration:

When writing a C++ UDF, use the DecimalVal data type defined in /usr/include/impala_udf/udf.h.

Changing precision and scale:

You can issue an ALTER TABLE ... REPLACE COLUMNS statement to change the precision and scale of an existing DECIMAL column.

- For text-based formats (text, RCFile, and SequenceFile tables)
  - If the values in the column fit within the new precision and scale, they are returned correctly by a query.
  - If any values that do not fit within the new precision and scale:
    - Impala returns an error if the query option ABORT_ON_ERROR is set to true.
    - Impala returns a NULL and warning that conversion failed if the query option ABORT_ON_ERROR is set to false.
  - Leading zeros do not count against the precision value, but trailing zeros after the decimal point do.

- For binary formats (Parquet and Avro tables)
  - Although an ALTER TABLE ... REPLACE COLUMNS statement that changes the precision or scale of a DECIMAL column succeeds, any subsequent attempt to query the changed column results in a fatal error. This is because the metadata about the columns is stored in the data files themselves, and ALTER TABLE does not actually make any updates to the data files. The other unaltered columns can still be queried successfully.
  - If the metadata in the data files disagrees with the metadata in the metastore database, Impala cancels the query.

Partitioning:

Using a DECIMAL column as a partition key provides you a better match between the partition key values and the HDFS directory names than using a DOUBLE or FLOAT partitioning column.

Column statistics considerations:

Because the DECIMAL type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Compatibility with older version of DECIMAL:

This version of DECIMAL type is the default in CDH 6.0 / Impala 3.0 and higher. The key differences between this version of DECIMAL and the previous DECIMAL V1 in Impala 2.x include the following.

	DECIMAL in CDH 6.0 / Impala 3.0 or higher	**DECIMAL in CDH 5.15 / Impala 2.12 or lower**
Overall behavior	Returns either the result or an error.	Returns either the result or NULL with a warning.
Overflow behavior	Aborts with an error.	Issues a warning and returns NULL.
Truncation / rounding behavior in arithmetic	Truncates and rounds digits from the back.	Truncates digits from the front.
String cast	Truncates from the back and rounds.	Truncates from the back.

If you need to continue using the first version of the DECIMAL type for the backward compatibility of your queries, set the DECIMAL_V2 query option to FALSE:

```
SET DECIMAL_V2=FALSE;
```

Compatibility with other databases:

Use the DECIMAL data type in Impala for applications where you used the NUMBER data type in Oracle.

The Impala DECIMAL type does not support the Oracle idioms of * for scale.

The Impala DECIMAL type does not support negative values for precision.

DOUBLE Data Type

A double precision floating-point data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name DOUBLE
```

Range: 4.94065645841246544e-324d .. 1.79769313486231570e+308, positive or negative

Precision: 15 to 17 significant digits, depending on usage. The number of significant digits does not depend on the position of the decimal point.

Representation: The values are stored in 8 bytes, using IEEE 754 Double Precision Binary Floating Point format.

Conversions: Impala does not automatically convert DOUBLE to any other type. You can use CAST() to convert DOUBLE values to FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING, TIMESTAMP, or BOOLEAN. You can use exponential notation in DOUBLE literals or when casting from STRING, for example 1.0e6 to represent one million. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting --use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

The data type REAL is an alias for DOUBLE.

Impala does not evaluate NaN (not a number) as equal to any other numeric values, including other NaN values. For example, the following statement, which evaluates equality between two NaN values, returns false:

```
SELECT CAST('nan' AS DOUBLE)=CAST('nan' AS DOUBLE);
```

Examples:

```
CREATE TABLE t1 (x DOUBLE);
SELECT CAST(1000.5 AS DOUBLE);
```

Partitioning: Because fractional values of this type are not always represented precisely, when this type is used for a partition key column, the underlying HDFS directories might not be named exactly as you expect. Prefer to partition on a DECIMAL column instead.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as an 8-byte value.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM() and AVG() for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

The inability to exactly represent certain floating-point values means that DECIMAL is sometimes a better choice than DOUBLE or FLOAT when precision is critical, particularly when transferring data from other database systems that use different representations or file formats.

Kudu considerations:

Currently, the data types BOOLEAN, FLOAT, and DOUBLE cannot be used for primary key columns in Kudu tables.

Related information:

Numeric Literals on page 167, Impala Mathematical Functions on page 397, FLOAT Data Type on page 116

FLOAT Data Type

A single precision floating-point data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name FLOAT
```

Range: 1.40129846432481707e-45 .. 3.40282346638528860e+38, positive or negative

Precision: 6 to 9 significant digits, depending on usage. The number of significant digits does not depend on the position of the decimal point.

Representation: The values are stored in 4 bytes, using IEEE 754 Single Precision Binary Floating Point format.

Conversions: Impala automatically converts FLOAT to more precise DOUBLE values, but not the other way around. You can use CAST( ) to convert FLOAT values to TINYINT, SMALLINT, INT, BIGINT, STRING, TIMESTAMP, or BOOLEAN. You can use exponential notation in FLOAT literals or when casting from STRING, for example 1.0e6 to represent one million. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting --use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

Impala does not evaluate NaN (not a number) as equal to any other numeric values, including other NaN values. For example, the following statement, which evaluates equality between two NaN values, returns false:

```
SELECT CAST('nan' AS FLOAT)=CAST('nan' AS FLOAT);
```

Examples:

```
CREATE TABLE t1 (x FLOAT);
SELECT CAST(1000.5 AS FLOAT);
```

Partitioning: Because fractional values of this type are not always represented precisely, when this type is used for a partition key column, the underlying HDFS directories might not be named exactly as you expect. Prefer to partition on a DECIMAL column instead.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 4-byte value.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM( ) and AVG( ) for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

The inability to exactly represent certain floating-point values means that DECIMAL is sometimes a better choice than DOUBLE or FLOAT when precision is critical, particularly when transferring data from other database systems that use different representations or file formats.

Kudu considerations:

Currently, the data types BOOLEAN, FLOAT, and DOUBLE cannot be used for primary key columns in Kudu tables.

Related information:

Numeric Literals on page 167, Impala Mathematical Functions on page 397, DOUBLE Data Type on page 114

INT Data Type

A 4-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name INT
```

Range: -2147483648 .. 2147483647. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a larger integer type (BIGINT) or a floating-point type (FLOAT or DOUBLE) automatically. Use CAST( ) to convert to TINYINT, SMALLINT, STRING, or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting --use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

The data type INTEGER is an alias for INT.

For a convenient and automated way to check the bounds of the INT type, call the functions MIN_INT( ) and MAX_INT( ).

If an integer value is too large to be represented as a INT, use a BIGINT instead.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Examples:

```
CREATE TABLE t1 (x INT);
SELECT CAST(1000 AS INT);
```

Partitioning: Prefer to use this type for a partition key column. Impala can process the numeric type more efficiently than a STRING representation of the value.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations:

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 4-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Related information:

Numeric Literals on page 167, TINYINT Data Type on page 136, SMALLINT Data Type on page 122, INT Data Type on page 117, BIGINT Data Type on page 105, DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only) on page 109, Impala Mathematical Functions on page 397

MAP Complex Type (CDH 5.5 or higher only)

A complex data type representing an arbitrary set of key-value pairs. The key part is a scalar type, while the value part can be a scalar or another complex type (ARRAY, STRUCT, or MAP).

Syntax:

```
column_name MAP < primitive_type, type >

type ::= primitive_type | complex_type
```

Usage notes:

Because complex types are often used in combination, for example an ARRAY of STRUCT elements, if you are unfamiliar with the Impala complex types, start with Complex Types (CDH 5.5 or higher only) on page 139 for background information and usage examples.

The MAP complex data type represents a set of key-value pairs. Each element of the map is indexed by a primitive type such as BIGINT or STRING, letting you define sequences that are not continuous or categories with arbitrary names. You might find it convenient for modelling data produced in other languages, such as a Python dictionary or Java HashMap, where a single scalar value serves as the lookup key.

In a big data context, the keys in a map column might represent a numeric sequence of events during a manufacturing process, or TIMESTAMP values corresponding to sensor observations. The map itself is inherently unordered, so you choose whether to make the key values significant (such as a recorded TIMESTAMP) or synthetic (such as a random global universal ID).



Note: Behind the scenes, the MAP type is implemented in a similar way as the ARRAY type. Impala does not enforce any uniqueness constraint on the KEY values, and the KEY values are processed by looping through the elements of the MAP rather than by a constant-time lookup. Therefore, this type is primarily for ease of understanding when importing data and algorithms from non-SQL contexts, rather than optimizing the performance of key lookups.

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

Added in: CDH 5.5.0 / Impala 2.3.0

Restrictions:

- Columns with this data type can only be used in tables or partitions with the Parquet file format.
- Columns with this data type cannot be used as partition key columns in a partitioned table.
- The COMPUTE STATS statement does not produce any statistics for columns of this data type.
- The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.
- See Limitations and Restrictions for Complex Types on page 143 for a full list of limitations and associated guidelines about complex type columns.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Examples:

Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types on page 160 for the table definitions.

The following example shows a table with various kinds of MAP columns, both at the top level and nested within other complex types. Each row represents information about a specific country, with complex type fields of various levels of nesting to represent different information associated with the country: factual measurements such as area and population, notable people in different categories, geographic features such as cities, points of interest within each city, and mountains with associated facts. Practice the CREATE TABLE and query notation for complex type columns using empty tables, until you can visualize a complex data structure and construct corresponding SQL statements reliably.

```
create TABLE map_demo
(
  country_id BIGINT,

-- Numeric facts about each country, looked up by name.
-- For example, 'Area':1000, 'Population':999999.
-- Using a MAP instead of a STRUCT because there could be
-- a different set of facts for each country.
  metrics MAP <STRING, BIGINT>,

-- MAP whose value part is an ARRAY.
-- For example, the key 'Famous Politicians' could represent an array of 10 elements,
-- while the key 'Famous Actors' could represent an array of 20 elements.
  notables MAP <STRING, ARRAY <STRING>>,

-- MAP that is a field within a STRUCT.
-- (The STRUCT is inside another ARRAY, because it is rare
-- for a STRUCT to be a top-level column.)
-- For example, city #1 might have points of interest with key 'Zoo',
-- representing an array of 3 different zoos.
-- City #2 might have completely different kinds of points of interest.
-- Because the set of field names is potentially large, and most entries could be blank,
-- a MAP makes more sense than a STRUCT to represent such a sparse data structure.
  cities ARRAY < STRUCT <
    name: STRING,
    points_of_interest: MAP <STRING, ARRAY <STRING>>
  >>,

-- MAP that is an element within an ARRAY. The MAP is inside a STRUCT field to associate
-- the mountain name with all the facts about the mountain.
-- The "key" of the map (the first STRING field) represents the name of some fact whose
 value
-- can be expressed as an integer, such as 'Height', 'Year First Climbed', and so on.
  mountains ARRAY < STRUCT < name: STRING, facts: MAP <STRING, INT > > >
)
STORED AS PARQUET;
```

```
DESCRIBE map_demo;
+------------+-------------------------------------------------+
| name       | type                                            |
+------------+-------------------------------------------------+
| country_id | bigint                                          |
| metrics    | map<string,bigint>                              |
| notables   | map<string,array<string>>                       |
| cities     | array<struct<                                   |
|            |   name:string,                                  |
|            |   points_of_interest:map<string,array<string>> |
|            | >>                                              |
| mountains  | array<struct<                                   |
|            |   name:string,                                  |
|            |   facts:map<string,int>                         |
|            | >>                                              |
+------------+-------------------------------------------------+

DESCRIBE map_demo.metrics;
```

```
+-------+--------+
| name  | type   |
+-------+--------+
| key   | string |
| value | bigint |
+-------+--------+

DESCRIBE map_demo.notables;
+-------+---------------+
| name  | type          |
+-------+---------------+
| key   | string        |
| value | array<string> |
+-------+---------------+

DESCRIBE map_demo.notables.value;
+------+--------+
| name | type   |
+------+--------+
| item | string |
| pos  | bigint |
+------+--------+

DESCRIBE map_demo.cities;
+------+-------------------------------------------------+
| name | type                                            |
+------+-------------------------------------------------+
| item | struct<                                         |
|      |   name:string,                                  |
|      |   points_of_interest:map<string,array<string>> |
|      | >                                               |
| pos  | bigint                                          |
+------+-------------------------------------------------+

DESCRIBE map_demo.cities.item.points_of_interest;
+-------+---------------+
| name  | type          |
+-------+---------------+
| key   | string        |
| value | array<string> |
+-------+---------------+

DESCRIBE map_demo.cities.item.points_of_interest.value;
+------+--------+
| name | type   |
+------+--------+
| item | string |
| pos  | bigint |
+------+--------+

DESCRIBE map_demo.mountains;
+------+-------------------------+
| name | type                    |
+------+-------------------------+
| item | struct<                 |
|      |   name:string,          |
|      |   facts:map<string,int> |
|      | >                       |
| pos  | bigint                  |
+------+-------------------------+

DESCRIBE map_demo.mountains.item.facts;
+-------+--------+
| name  | type   |
+-------+--------+
| key   | string |
| value | int    |
+-------+--------+
```

The following example shows a table that uses a variety of data types for the MAP "key" field. Typically, you use BIGINT or STRING to use numeric or character-based keys without worrying about exceeding any size or length constraints.

```
CREATE TABLE map_demo_obscure
(
  id BIGINT,
  m1 MAP <INT, INT>,
  m2 MAP <SMALLINT, INT>,
  m3 MAP <TINYINT, INT>,
  m4 MAP <TIMESTAMP, INT>,
  m5 MAP <BOOLEAN, INT>,
  m6 MAP <CHAR(5), INT>,
  m7 MAP <VARCHAR(25), INT>,
  m8 MAP <FLOAT, INT>,
  m9 MAP <DOUBLE, INT>,
  m10 MAP <DECIMAL(12,2), INT>
)
STORED AS PARQUET;
```

```
CREATE TABLE celebrities (name STRING, birth_year MAP < STRING, SMALLINT >) STORED AS
PARQUET;
-- A typical row might represent values with 2 different birth years, such as:
-- ("Joe Movie Star", { "real": 1972, "claimed": 1977 })

CREATE TABLE countries (name STRING, famous_leaders MAP < INT, STRING >) STORED AS
PARQUET;
-- A typical row might represent values with different leaders, with key values
corresponding to their numeric sequence, such as:
-- ("United States", { 1: "George Washington", 3: "Thomas Jefferson", 16: "Abraham
Lincoln" })

CREATE TABLE airlines (name STRING, special_meals MAP < STRING, MAP < STRING, STRING >
 >) STORED AS PARQUET;
-- A typical row might represent values with multiple kinds of meals, each with several
 components:
-- ("Elegant Airlines",
--   {
--     "vegetarian": { "breakfast": "pancakes", "snack": "cookies", "dinner": "rice
pilaf" },
--     "gluten free": { "breakfast": "oatmeal", "snack": "fruit", "dinner": "chicken"
}
--   } )
```

Related information:

Complex Types (CDH 5.5 or higher only) on page 139, ARRAY Complex Type (CDH 5.5 or higher only) on page 102, STRUCT Complex Type (CDH 5.5 or higher only) on page 124

REAL Data Type

An alias for the DOUBLE data type. See DOUBLE Data Type on page 114 for details.

Examples:

These examples show how you can use the type names REAL and DOUBLE interchangeably, and behind the scenes Impala treats them always as DOUBLE.

```
[localhost:21000] > create table r1 (x real);
[localhost:21000] > describe r1;
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | double |         |
+------+--------+---------+
[localhost:21000] > insert into r1 values (1.5), (cast (2.2 as double));
[localhost:21000] > select cast (1e6 as real);
+--------------------------+
| cast(1000000.0 as double) |
```

```
+--------------------------+
| 1000000                  |
+--------------------------+
```

SMALLINT Data Type

A 2-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name SMALLINT
```

Range: -32768 .. 32767. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a larger integer type (INT or BIGINT) or a floating-point type (FLOAT or DOUBLE) automatically. Use CAST( ) to convert to TINYINT, STRING, or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting --use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

For a convenient and automated way to check the bounds of the SMALLINT type, call the functions MIN_SMALLINT( ) and MAX_SMALLINT( ).

If an integer value is too large to be represented as a SMALLINT, use an INT instead.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Examples:

```
CREATE TABLE t1 (x SMALLINT);
SELECT CAST(1000 AS SMALLINT);
```

Parquet considerations:

Physically, Parquet files represent TINYINT and SMALLINT values as 32-bit integers. Although Impala rejects attempts to insert out-of-range values into such columns, if you create a new table with the CREATE TABLE ... LIKE PARQUET syntax, any TINYINT or SMALLINT columns in the original table turn into INT columns in the new table.

Partitioning: Prefer to use this type for a partition key column. Impala can process the numeric type more efficiently than a STRING representation of the value.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 2-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Related information:

Numeric Literals on page 167, TINYINT Data Type on page 136, SMALLINT Data Type on page 122, INT Data Type on page 117, BIGINT Data Type on page 105, DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only) on page 109, Impala Mathematical Functions on page 397

STRING Data Type

A data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE and ALTER TABLE statements:

```
column_name STRING
```

Length:

If you need to manipulate string values with precise or maximum lengths, in Impala 2.0 and higher you can declare columns as VARCHAR(*max_length*) or CHAR(*length*), but for best performance use STRING where practical.

Take the following considerations for STRING lengths:

- The hard limit on the size of a STRING and the total size of a row is 2 GB.

  If a query tries to process or create a string larger than this limit, it will return an error to the user.

- The limit is 1 GB on STRING when writing to Parquet files.
- Queries operating on strings with 32 KB or less will work reliably and will not hit significant performance or memory problems (unless you have very complex queries, very many columns, etc.)
- Performance and memory consumption may degrade with strings larger than 32 KB.
- The row size, i.e. the total size of all string and other columns, is subject to lower limits at various points in query execution that support spill-to-disk. A few examples for lower row size limits are:
  - Rows coming from the right side of any hash join
  - Rows coming from either side of a hash join that spills to disk
  - Rows being sorted by the SORT operator without a limit
  - Rows in a grouping aggregation

  In CDH 5.12 and lower, the default limit of the row size in the above cases is 8 MB.

  In CDH 5.13 and higher, the max row size is configurable on a per-query basis with the MAX_ROW_SIZE query option. Rows up to MAX_ROW_SIZE (which defaults to 512 KB) can always be processed in the above cases. Rows larger than MAX_ROW_SIZE are processed on a best-effort basis. See MAX_ROW_SIZE for more details.

Character sets:

For full support in all Impala subsystems, restrict string values to the ASCII character set. Although some UTF-8 character data can be stored in Impala and retrieved through queries, UTF-8 strings containing non-ASCII characters are not guaranteed to work properly in combination with many SQL aspects, including but not limited to:

- String manipulation functions.
- Comparison operators.
- The ORDER BY clause.
- Values in partition key columns.

For any national language aspects such as collation order or interpreting extended ASCII variants such as ISO-8859-1 or ISO-8859-2 encodings, Impala does not include such metadata with the table definition. If you need to sort, manipulate, or display data depending on those national language characteristics of string data, use logic on the application side.

Conversions:

- Impala does not automatically convert STRING to any numeric type. Impala does automatically convert STRING to TIMESTAMP if the value matches one of the accepted TIMESTAMP formats; see TIMESTAMP Data Type on page 130 for details.
- You can use CAST( ) to convert STRING values to TINYINT, SMALLINT, INT, BIGINT, FLOAT, DOUBLE, or TIMESTAMP.

- You cannot directly cast a STRING value to BOOLEAN. You can use a CASE expression to evaluate string values such as 'T', 'true', and so on and return Boolean true and false values as appropriate.
- You can cast a BOOLEAN value to STRING, returning '1' for true values and '0' for false values.

Partitioning:

Although it might be convenient to use STRING columns for partition keys, even when those columns contain numbers, for performance and scalability it is much better to use numeric columns as partition keys whenever practical. Although the underlying HDFS directory name might be the same in either case, the in-memory storage for the partition key columns is more compact, and computations are faster, if partition key columns such as YEAR, MONTH, DAY and so on are declared as INT, SMALLINT, and so on.

Zero-length strings: For purposes of clauses such as DISTINCT and GROUP BY, Impala considers zero-length strings (""), NULL, and space to all be different values.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Avro considerations:

The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In CDH 5.7 / Impala 2.5 and higher, Impala truncates CHAR and VARCHAR values in Avro tables to (2**31)-1 bytes. If a query encounters a STRING value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Column statistics considerations: Because the values of this type have variable size, none of the column statistics fields are filled in until you run the COMPUTE STATS statement.

Examples:

The following examples demonstrate double-quoted and single-quoted string literals, and required escaping for quotation marks within string literals:

```
SELECT 'I am a single-quoted string';
SELECT "I am a double-quoted string";
SELECT 'I\'m a single-quoted string with an apostrophe';
SELECT "I\'m a double-quoted string with an apostrophe";
SELECT 'I am a "short" single-quoted string containing quotes';
SELECT "I am a \"short\" double-quoted string containing quotes";
```

The following examples demonstrate calls to string manipulation functions to concatenate strings, convert numbers to strings, or pull out substrings:

```
SELECT CONCAT("Once upon a time, there were ", CAST(3 AS STRING), ' little pigs.');
SELECT SUBSTR("hello world",7,5);
```

The following examples show how to perform operations on STRING columns within a table:

```
CREATE TABLE t1 (s1 STRING, s2 STRING);
INSERT INTO t1 VALUES ("hello", 'world'), (CAST(7 AS STRING), "wonders");
SELECT s1, s2, length(s1) FROM t1 WHERE s2 LIKE 'w%';
```

Related information:

String Literals on page 168, CHAR Data Type (CDH 5.2 or higher only) on page 107, VARCHAR Data Type (CDH 5.2 or higher only) on page 137, Impala String Functions on page 462, Impala Date and Time Functions on page 424

STRUCT Complex Type (CDH 5.5 or higher only)

A complex data type, representing multiple fields of a single item. Frequently used as the element type of an ARRAY or the VALUE part of a MAP.

Syntax:

```
column_name STRUCT < name : type [COMMENT 'comment_string'], ... >

type ::= primitive_type | complex_type
```

The names and number of fields within the STRUCT are fixed. Each field can be a different type. A field within a STRUCT can also be another STRUCT, or an ARRAY or a MAP, allowing you to create nested data structures with a maximum nesting depth of 100.

A STRUCT can be the top-level type for a column, or can itself be an item within an ARRAY or the value part of the key-value pair in a MAP.

When a STRUCT is used as an ARRAY element or a MAP value, you use a join clause to bring the ARRAY or MAP elements into the result set, and then refer to *array_name*`.ITEM.`*field* or *map_name*`.VALUE.`*field*. In the case of a STRUCT directly inside an ARRAY or MAP, you can omit the `.ITEM` and `.VALUE` pseudocolumns and refer directly to *array_name*`.`*field* or *map_name*`.`*field*.

Usage notes:

Because complex types are often used in combination, for example an ARRAY of STRUCT elements, if you are unfamiliar with the Impala complex types, start with Complex Types (CDH 5.5 or higher only) on page 139 for background information and usage examples.

A STRUCT is similar conceptually to a table row: it contains a fixed number of named fields, each with a predefined type. To combine two related tables, while using complex types to minimize repetition, the typical way to represent that data is as an ARRAY of STRUCT elements.

Because a STRUCT has a fixed number of named fields, it typically does not make sense to have a STRUCT as the type of a table column. In such a case, you could just make each field of the STRUCT into a separate column of the table. The STRUCT type is most useful as an item of an ARRAY or the value part of the key-value pair in a MAP. A nested type column with a STRUCT at the lowest level lets you associate a variable number of row-like objects with each row of the table.

The STRUCT type is straightforward to reference within a query. You do not need to include the STRUCT column in a join clause or give it a table alias, as is required for the ARRAY and MAP types. You refer to the individual fields using dot notation, such as *struct_column_name*`.`*field_name*, without any pseudocolumn such as `ITEM` or `VALUE`.

You can pass a multi-part qualified name to `DESCRIBE` to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table `T1` contains an ARRAY column `A1`, you could issue the statement `DESCRIBE t1.a1`. If table `T1` contained a STRUCT column `S1`, and a field `F1` within the STRUCT was a MAP, you could issue the statement `DESCRIBE t1.s1.f1`. An ARRAY is shown as a two-column table, with `ITEM` and `POS` columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

Internal details:

Within the Parquet data file, the values for each STRUCT field are stored adjacent to each other, so that they can be encoded and compressed using all the Parquet techniques for storing sets of similar or repeated values. The adjacency applies even when the STRUCT values are part of an ARRAY or MAP. During a query, Impala avoids unnecessary I/O by reading only the portions of the Parquet data file containing the requested STRUCT fields.

Added in: CDH 5.5.0 / Impala 2.3.0

Restrictions:

- Columns with this data type can only be used in tables or partitions with the Parquet file format.
- Columns with this data type cannot be used as partition key columns in a partitioned table.
- The `COMPUTE STATS` statement does not produce any statistics for columns of this data type.
- The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.

- See Limitations and Restrictions for Complex Types on page 143 for a full list of limitations and associated guidelines about complex type columns.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Examples:



Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types on page 160 for the table definitions.

The following example shows a table with various kinds of STRUCT columns, both at the top level and nested within other complex types. Practice the CREATE TABLE and query notation for complex type columns using empty tables, until you can visualize a complex data structure and construct corresponding SQL statements reliably.

```
CREATE TABLE struct_demo
(
  id BIGINT,
  name STRING,

-- A STRUCT as a top-level column. Demonstrates how the table ID column
-- and the ID field within the STRUCT can coexist without a name conflict.
  employee_info STRUCT < employer: STRING, id: BIGINT, address: STRING >,

-- A STRUCT as the element type of an ARRAY.
  places_lived ARRAY < STRUCT <street: STRING, city: STRING, country: STRING >>,

-- A STRUCT as the value portion of the key-value pairs in a MAP.
  memorable_moments MAP < STRING, STRUCT < year: INT, place: STRING, details: STRING
>>,

-- A STRUCT where one of the fields is another STRUCT.
  current_address STRUCT < street_address: STRUCT <street_number: INT, street_name:
STRING, street_type: STRING>, country: STRING, postal_code: STRING >
)
STORED AS PARQUET;
```

The following example shows how to examine the structure of a table containing one or more STRUCT columns by using the DESCRIBE statement. You can visualize each STRUCT as its own table, with columns named the same as each field of the STRUCT. If the STRUCT is nested inside another complex type, such as ARRAY, you can extend the qualified name passed to DESCRIBE until the output shows just the STRUCT fields.

```
DESCRIBE struct_demo;
+-------------------+--------------------------+
| name              | type                     |
+-------------------+--------------------------+
| id                | bigint                   |
| name              | string                   |
| employee_info     | struct<                  |
|                   |   employer:string,       |
|                   |   id:bigint,             |
|                   |   address:string         |
|                   | >                        |
| places_lived      | array<struct<            |
|                   |   street:string,         |
|                   |   city:string,           |
|                   |   country:string         |
|                   | >>                       |
| memorable_moments | map<string,struct<       |
|                   |   year:int,              |
|                   |   place:string,          |
|                   |   details:string         |
|                   | >>                       |
| current_address   | struct<                  |
```

```
|                   |   street_address:struct< |
|                   |     street_number:int,   |
|                   |     street_name:string,  |
|                   |     street_type:string   |
|                   |   >,                     |
|                   |   country:string,        |
|                   |   postal_code:string     |
|                   | >                        |
+-------------------+--------------------------+
```

The top-level column EMPLOYEE_INFO is a STRUCT. Describing *table_name*.*struct_name* displays the fields of the STRUCT as if they were columns of a table:

```
DESCRIBE struct_demo.employee_info;
+----------+--------+
| name     | type   |
+----------+--------+
| employer | string |
| id       | bigint |
| address  | string |
+----------+--------+
```

Because PLACES_LIVED is a STRUCT inside an ARRAY, the initial DESCRIBE shows the structure of the ARRAY:

```
DESCRIBE struct_demo.places_lived;
+------+------------------+
| name | type             |
+------+------------------+
| item | struct<          |
|      |   street:string, |
|      |   city:string,   |
|      |   country:string |
|      | >                |
| pos  | bigint           |
+------+------------------+
```

Ask for the details of the ITEM field of the ARRAY to see just the layout of the STRUCT:

```
DESCRIBE struct_demo.places_lived.item;
+---------+--------+
| name    | type   |
+---------+--------+
| street  | string |
| city    | string |
| country | string |
+---------+--------+
```

Likewise, MEMORABLE_MOMENTS has a STRUCT inside a MAP, which requires an extra level of qualified name to see just the STRUCT part:

```
DESCRIBE struct_demo.memorable_moments;
+-------+------------------+
| name  | type             |
+-------+------------------+
| key   | string           |
| value | struct<          |
|       |   year:int,      |
|       |   place:string,  |
|       |   details:string |
|       | >                |
+-------+------------------+
```

For a MAP, ask to see the VALUE field to see the corresponding STRUCT fields in a table-like structure:

```
DESCRIBE struct_demo.memorable_moments.value;
+---------+--------+
| name    | type   |
+---------+--------+
| year    | int    |
| place   | string |
| details | string |
+---------+--------+
```

For a STRUCT inside a STRUCT, we can see the fields of the outer STRUCT:

```
DESCRIBE struct_demo.current_address;
+----------------+-----------------------+
| name           | type                  |
+----------------+-----------------------+
| street_address | struct<               |
|                |   street_number:int,  |
|                |   street_name:string, |
|                |   street_type:string  |
|                | >                     |
| country        | string                |
| postal_code    | string                |
+----------------+-----------------------+
```

Then we can use a further qualified name to see just the fields of the inner STRUCT:

```
DESCRIBE struct_demo.current_address.street_address;
+---------------+--------+
| name          | type   |
+---------------+--------+
| street_number | int    |
| street_name   | string |
| street_type   | string |
+---------------+--------+
```

The following example shows how to examine the structure of a table containing one or more STRUCT columns by using the DESCRIBE statement. You can visualize each STRUCT as its own table, with columns named the same as each field of the STRUCT. If the STRUCT is nested inside another complex type, such as ARRAY, you can extend the qualified name passed to DESCRIBE until the output shows just the STRUCT fields.

```
DESCRIBE struct_demo;
+-------------------+--------------------------+---------+
| name              | type                     | comment |
+-------------------+--------------------------+---------+
| id                | bigint                   |         |
| name              | string                   |         |
| employee_info     | struct<                  |         |
|                   |   employer:string,       |         |
|                   |   id:bigint,             |         |
|                   |   address:string         |         |
|                   | >                        |         |
| places_lived      | array<struct<            |         |
|                   |   street:string,         |         |
|                   |   city:string,           |         |
|                   |   country:string         |         |
|                   | >>                       |         |
| memorable_moments | map<string,struct<       |         |
|                   |   year:int,              |         |
|                   |   place:string,          |         |
|                   |   details:string         |         |
|                   | >>                       |         |
| current_address   | struct<                  |         |
|                   |   street_address:struct< |         |
|                   |     street_number:int,   |         |
```

```
|                   |     street_name:string,  |         |
|                   |     street_type:string   |         |
|                   |   >,                     |         |
|                   |   country:string,        |         |
|                   |   postal_code:string     |         |
|                   | >                        |         |
+-------------------+--------------------------+---------+

SELECT id, employee_info.id FROM struct_demo;

SELECT id, employee_info.id AS employee_id FROM struct_demo;

SELECT id, employee_info.id AS employee_id, employee_info.employer
  FROM struct_demo;

SELECT id, name, street, city, country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, places_lived.pos, places_lived.street, places_lived.city,
places_lived.country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, pl.pos, pl.street, pl.city, pl.country
  FROM struct_demo, struct_demo.places_lived AS pl;

SELECT id, name, places_lived.pos, places_lived.street, places_lived.city,
places_lived.country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, pos, street, city, country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, memorable_moments.key,
  memorable_moments.value.year,
  memorable_moments.value.place,
  memorable_moments.value.details
FROM struct_demo, struct_demo.memorable_moments
WHERE memorable_moments.key IN ('Birthday','Anniversary','Graduation');

SELECT id, name, mm.key, mm.value.year, mm.value.place, mm.value.details
  FROM struct_demo, struct_demo.memorable_moments AS mm
WHERE mm.key IN ('Birthday','Anniversary','Graduation');

SELECT id, name, memorable_moments.key, memorable_moments.value.year,
  memorable_moments.value.place, memorable_moments.value.details
FROM struct_demo, struct_demo.memorable_moments
WHERE key IN ('Birthday','Anniversary','Graduation');

SELECT id, name, key, value.year, value.place, value.details
  FROM struct_demo, struct_demo.memorable_moments
WHERE key IN ('Birthday','Anniversary','Graduation');

SELECT id, name, key, year, place, details
  FROM struct_demo, struct_demo.memorable_moments
WHERE key IN ('Birthday','Anniversary','Graduation');

SELECT id, name,
  current_address.street_address.street_number,
  current_address.street_address.street_name,
  current_address.street_address.street_type,
  current_address.country,
  current_address.postal_code
FROM struct_demo;
```

For example, this table uses a struct that encodes several data values for each phone number associated with a person. Each person can have a variable-length array of associated phone numbers, and queries can refer to the category field to locate specific home, work, mobile, and so on kinds of phone numbers.

```
CREATE TABLE contact_info_many_structs
(
  id BIGINT, name STRING,
```

```
  phone_numbers ARRAY < STRUCT <category:STRING, country_code:STRING, area_code:SMALLINT,
 full_number:STRING, mobile:BOOLEAN, carrier:STRING > >
) STORED AS PARQUET;
```

Because structs are naturally suited to composite values where the fields have different data types, you might use them to decompose things such as addresses:

```
CREATE TABLE contact_info_detailed_address
(
  id BIGINT, name STRING,
  address STRUCT < house_number:INT, street:STRING, street_type:STRING, apartment:STRING,
 city:STRING, region:STRING, country:STRING >
);
```

In a big data context, splitting out data fields such as the number part of the address and the street name could let you do analysis on each field independently. For example, which streets have the largest number range of addresses, what are the statistical properties of the street names, which areas have a higher proportion of “Roads”, “Courts” or “Boulevards”, and so on.

Related information:

Complex Types (CDH 5.5 or higher only) on page 139, ARRAY Complex Type (CDH 5.5 or higher only) on page 102, MAP Complex Type (CDH 5.5 or higher only) on page 118

TIMESTAMP Data Type

In Impala, the `TIMESTAMP` data type holds a value of date and time. It can be decomposed into year, month, day, hour, minute and seconds fields, but with no time zone information available, it does not correspond to any specific point in time.

Internally, the resolution of the time portion of a `TIMESTAMP` value is in nanoseconds.

Syntax:

In the column definition of a `CREATE TABLE` statement:

```
column_name TIMESTAMP
```

```
timestamp [+ | -] INTERVAL interval
DATE_ADD (timestamp, INTERVAL interval time_unit)
```

Range: 1400-01-01 to 9999-12-31

Out of range `TIMESTAMP` values are converted to NULL.

The range of Impala `TIMESTAMP` is different from the Hive `TIMESTAMP` type. Refer to Hive documentation for detail.

INTERVAL expressions:

You can perform date arithmetic by adding or subtracting a specified number of time units, using the `INTERVAL` keyword and the + operator, the - operator, `date_add()` or `date_sub()`.

The following units are supported for *`time_unit`* in the `INTERVAL` clause:

- `YEAR[S]`
- `MONTH[S]`
- `WEEK[S]`
- `DAY[S]`
- `HOUR[S]`
- `MINUTE[S]`
- `SECOND[S]`

- MILLISECOND[S]
- MICROSECOND[S]
- NANOSECOND[S]

You can only specify one time unit in each interval expression, for example INTERVAL 3 DAYS or INTERVAL 25 HOURS, but you can produce any granularity by adding together successive INTERVAL values, such as *timestamp_value* + INTERVAL 3 WEEKS - INTERVAL 1 DAY + INTERVAL 10 MICROSECONDS.

Internal details: Represented in memory as a 16-byte value.

Time zones:

By default, Impala stores and interprets TIMESTAMP values in UTC time zone when writing to data files, reading from data files, or converting to and from system time values through functions.

When you set the --use_local_tz_for_unix_timestamp_conversions startup flag to TRUE, Impala treats the TIMESTAMP values specified in the local time zone. The local time zone is determined in the following order with the TIMESTAMP query option takes the highest precedence:

1. The TIMESTAMP query option
2. $TZ environment variable
3. System time zone where the impalad coordinator runs

The --use_local_tz_for_unix_timestamp_conversions setting can be used to fix discrepancy in INTERVAL operations. For example, a TIMESTAMP + INTERVAL *n-hours* can be affected by Daylight Saving Time, which Impala does not consider by default as these operations are applied as if the timestamp was in UTC. You can use the --use_local_tz_for_unix_timestamp_conversions setting to fix the issue.

See Customizing Time Zones on page 135 for configuring to use custom time zone database and aliases.

See Impala Date and Time Functions for the list of functions affected by the --use_local_tz_for_unix_timestamp_conversions setting.

Time zone handling between Impala and Hive:

Interoperability between Hive and Impala is different depending on the file format.

- *Text*

  For text tables, TIMESTAMP values can be written and read interchangeably by Impala and Hive as Hive reads and writes TIMESTAMP values without converting with respect to time zones.

- *Parquet*

  Note: This section only applies to INT96 TIMESTAMP. See Data Type Considerations for Parquet Tables on page 655 for information about Parquet data types.

  When Hive writes to Parquet data files, the TIMESTAMP values are normalized to UTC from the local time zone of the host where the data was written. On the other hand, Impala does not make any time zone adjustment when it writes or reads INT96 TIMESTAMP values to Parquet files. This difference in time zone handling can cause potentially inconsistent results when Impala processes TIMESTAMP values in the Parquet files written by Hive.

  To avoid incompatibility problems or having to code workarounds, you can specify one or both of these impalad startup flags:

  - --use_local_tz_for_unix_timestamp_conversions=true
  - --convert_legacy_hive_parquet_utc_timestamps=true

  When the --convert_legacy_hive_parquet_utc_timestamps setting is enabled, Impala recognizes the Parquet data files written by Hive, and applies the same UTC-to-local-timezone conversion logic during the query as Hive does.

  In CDH 6.0 / Impala 3.0 and lower, the --convert_legacy_hive_parquet_utc_timestamps setting had a severe impact on multi-threaded performance. The new time zone implementation in CDH 6.1 eliminated most

of the performance overhead and made Impala scale well to multiple threads. The `--convert_legacy_hive_parquet_utc_timestamps` setting is turned off by default for a performance reason. To avoid unexpected incompatibility problems, you should turn on the option when processing `TIMESTAMP` columns in Parquet files written by Hive.

Hive currently cannot write `INT64 TIMESTAMP` values.

In CDH 6.2 and higher, `INT64 TIMESTAMP` values annotated with the `TIMESTAMP_MILLIS` or `TIMESTAMP_MICROS` `OriginalType` are assumed to be always UTC normalized, so the UTC to local conversion will be always done. `INT64 TIMESTAMP` annotated with the `TIMESTAMP` `LogicalType` specifies whether UTC to local conversion is necessary depending on the Parquet metadata.

Conversions:

Impala automatically converts `STRING` literals of the correct format into `TIMESTAMP` values. Timestamp values are accepted in the format `'yyyy-MM-dd HH:mm:ss.SSSSSS'`, and can consist of just the date, or just the time, with or without the fractional second portion. For example, you can specify `TIMESTAMP` values such as `'1966-07-30'`, `'08:30:00'`, or `'1985-09-25 17:45:30.005'`.

Leading zeroes are not required in the numbers representing the date component, such as month and date, or the time component, such as hour, minute, and second. For example, Impala accepts both `'2018-1-1 01:02:03'` and `'2018-01-01 1:2:3'` as valid.

In `STRING` to `TIMESTAMP` conversions, leading and trailing white spaces, such as a space, a tab, a newline, or a carriage return, are ignored. For example, Impala treats the following as equivalent: '1999-12-01 01:02:03 ', ' 1999-12-01 01:02:03', '1999-12-01 01:02:03\r\n\t'.

When you convert or cast a `STRING` literal to `TIMESTAMP`, you can use the following separators between the date part and the time part:

- One or more space characters

  Example: `CAST('2001-01-09 01:05:01' AS TIMESTAMP)`

- The character “T”

  Example: `CAST('2001-01-09T01:05:01' AS TIMESTAMP)`

Casting an integer or floating-point value `N` to `TIMESTAMP` produces a value that is `N` seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting `--use_local_tz_for_unix_timestamp_conversions=true` is in effect, the resulting `TIMESTAMP` represents a date and time in the local time zone.

In Impala 1.3 and higher, the `FROM_UNIXTIME()` and `UNIX_TIMESTAMP()` functions allow a wider range of format strings, with more flexibility in element order, repetition of letter placeholders, and separator characters. In CDH 5.5 / Impala 2.3 and higher, the `UNIX_TIMESTAMP()` function also allows a numeric timezone offset to be specified as part of the input string. See Impala Date and Time Functions on page 424 for details.

In Impala 2.2.0 and higher, built-in functions that accept or return integers representing `TIMESTAMP` values use the `BIGINT` type for parameters and return values, rather than `INT`. This change lets the date and time functions avoid an overflow error that would otherwise occur on January 19th, 2038 (known as the “Year 2038 problem” or “Y2K38 problem”). This change affects the `FROM_UNIXTIME()` and `UNIX_TIMESTAMP()` functions. You might need to change application code that interacts with these functions, change the types of columns that store the return values, or add `CAST()` calls to SQL statements that call these functions.

Partitioning:

Although you cannot use a `TIMESTAMP` column as a partition key, you can extract the individual years, months, days, hours, and so on and partition based on those columns. Because the partition key column values are represented in HDFS directory names, rather than as fields in the data files themselves, you can also keep the original `TIMESTAMP`

values if desired, without duplicating data or wasting storage space. See Partition Key Columns on page 630 for more details on partitioning with date and time values.

```
[localhost:21000] > create table timeline (event string) partitioned by (happened
timestamp);
ERROR: AnalysisException: Type 'TIMESTAMP' is not supported as partition-column type in
 column: happened
```

NULL considerations: Casting any unrecognized STRING value to this type produces a NULL value.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet consideration: INT96 encoded Parquet timestamps are supported in Impala. INT64 timestamps are supported in CDH 6.2 and higher.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Kudu considerations:

In CDH 5.12 / Impala 2.9 and higher, you can include TIMESTAMP columns in Kudu tables, instead of representing the date and time as a BIGINT value. The behavior of TIMESTAMP for Kudu tables has some special considerations:

- Any nanoseconds in the original 96-bit value produced by Impala are not stored, because Kudu represents date/time columns using 64-bit values. The nanosecond portion of the value is rounded, not truncated. Therefore, a TIMESTAMP value that you store in a Kudu table might not be bit-for-bit identical to the value returned by a query.
- The conversion between the Impala 96-bit representation and the Kudu 64-bit representation introduces some performance overhead when reading or writing TIMESTAMP columns. You can minimize the overhead during writes by performing inserts through the Kudu API. Because the overhead during reads applies to each query, you might continue to use a BIGINT column to represent date/time values in performance-critical applications.
- The Impala TIMESTAMP type has a narrower range for years than the underlying Kudu data type. Impala can represent years 1400-9999. If year values outside this range are written to a Kudu table by a non-Impala client, Impala returns NULL by default when reading those TIMESTAMP values during a query. Or, if the ABORT_ON_ERROR query option is enabled, the query fails when it encounters a value with an out-of-range year.

Restrictions:

If you cast a STRING with an unrecognized format to a TIMESTAMP, the result is NULL rather than an error. Make sure to test your data pipeline to be sure any textual date and time values are in a format that Impala TIMESTAMP can recognize.

Currently, Avro tables cannot contain TIMESTAMP columns. If you need to store date and time values in Avro tables, as a workaround you can use a STRING representation of the values, convert the values to BIGINT with the UNIX_TIMESTAMP( ) function, or create separate numeric columns for individual date and time fields using the EXTRACT( ) function.

Examples:

The following examples demonstrate using TIMESTAMP values with built-in functions:

```
select cast('1966-07-30' as timestamp);
select cast('1985-09-25 17:45:30.005' as timestamp);
select cast('08:30:00' as timestamp);
select hour('1970-01-01 15:30:00');         -- Succeeds, returns 15.
select hour('1970-01-01 15:30');            -- Returns NULL because seconds field
required.
select hour('1970-01-01 27:30:00');         -- Returns NULL because hour value out of
range.
```

```
select dayofweek('2004-06-13');            -- Returns 1, representing Sunday.
select dayname('2004-06-13');              -- Returns 'Sunday'.
select date_add('2004-06-13', 365);       -- Returns 2005-06-13 with zeros for hh:mm:ss
 fields.
select day('2004-06-13');                  -- Returns 13.
select datediff('1989-12-31','1984-09-01'); -- How many days between these 2 dates?
select now();                              -- Returns current date and time in local
timezone.
```

The following examples demonstrate using TIMESTAMP values with HDFS-backed tables:

```
create table dates_and_times (t timestamp);
insert into dates_and_times values
  ('1966-07-30'), ('1985-09-25 17:45:30.005'), ('08:30:00'), (now());
```

The following examples demonstrate using TIMESTAMP values with Kudu tables:

```
create table timestamp_t (x int primary key, s string, t timestamp, b bigint)
  partition by hash (x) partitions 16
  stored as kudu;

-- The default value of now() has microsecond precision, so the final 3 digits
-- representing nanoseconds are all zero.
insert into timestamp_t values (1, cast(now() as string), now(), unix_timestamp(now()));

-- Values with 1-499 nanoseconds are rounded down in the Kudu TIMESTAMP column.
insert into timestamp_t values (2, cast(now() + interval 100 nanoseconds as string),
now() + interval 100 nanoseconds, unix_timestamp(now() + interval 100 nanoseconds));
insert into timestamp_t values (3, cast(now() + interval 499 nanoseconds as string),
now() + interval 499 nanoseconds, unix_timestamp(now() + interval 499 nanoseconds));

-- Values with 500-999 nanoseconds are rounded up in the Kudu TIMESTAMP column.
insert into timestamp_t values (4, cast(now() + interval 500 nanoseconds as string),
now() + interval 500 nanoseconds, unix_timestamp(now() + interval 500 nanoseconds));
insert into timestamp_t values (5, cast(now() + interval 501 nanoseconds as string),
now() + interval 501 nanoseconds, unix_timestamp(now() + interval 501 nanoseconds));

-- The string representation shows how underlying Impala TIMESTAMP can have nanosecond
 precision.
-- The TIMESTAMP column shows how timestamps in a Kudu table are rounded to microsecond
 precision.
-- The BIGINT column represents seconds past the epoch and so if not affected much by
nanoseconds.
select s, t, b from timestamp_t order by t;
+-------------------------------+-------------------------------+------------+
| s                             | t                             | b          |
+-------------------------------+-------------------------------+------------+
| 2017-05-31 15:30:05.107157000 | 2017-05-31 15:30:05.107157000 | 1496244605 |
| 2017-05-31 15:30:28.868151100 | 2017-05-31 15:30:28.868151000 | 1496244628 |
| 2017-05-31 15:34:33.674692499 | 2017-05-31 15:34:33.674692000 | 1496244873 |
| 2017-05-31 15:35:04.769166500 | 2017-05-31 15:35:04.769167000 | 1496244904 |
| 2017-05-31 15:35:33.033082501 | 2017-05-31 15:35:33.033083000 | 1496244933 |
+-------------------------------+-------------------------------+------------+
```

Added in: Available in all versions of Impala.

Related information:

- Timestamp Literals on page 169.
- To convert to or from different date formats, or perform date arithmetic, use the date and time functions described in Impala Date and Time Functions on page 424. In particular, the `from_unixtime()` function requires a case-sensitive format string such as `"yyyy-MM-dd HH:mm:ss.SSSS"`, matching one of the allowed variations of a TIMESTAMP value (date plus time, only date, only time, optional fractional seconds).
- See SQL Differences Between Impala and Hive on page 541 for details about differences in TIMESTAMP handling between Impala and Hive.

Customizing Time Zones

Starting in CDH 6.1 / Impala 3.1, you can customize the time zone definitions used in Impala.

- By default, Impala uses the OS's time zone database located in `/usr/share/zoneinfo`. This directory contains the IANA timezone database in a compiled binary format. The contents of the `zoneinfo` directory is controlled by the OS's package manager.
- Use the following start-up flags managed as `impalad` safety valves in Cloudera Manager.
  - `--hdfs_zone_info_zip`: This flag allows Impala administrators to specify a custom timezone database. The flag should be set to a shared (not necessarily HDFS) path that points to a zip archive of a custom IANA timezone database. The timezone database is expected to be in a compiled binary format. If the startup flag is set, Impala will use the specified timezone database instead of the default `/usr/share/zoneinfo` database. The timezone db upgrade process is described in detail below.
  - `--hdfs_zone_alias_conf`: This flag allows Impala administrators to specify definitions for custom timezone aliases. The flag should be set to a shared (not necessarily HDFS) path that specifies a config file containing custom timezone alias definitions. This config file can be used as a workaround for users who want to keep using their legacy timezone names. Configuring custom aliases is described in detail below.

Upgrading custom IANA time zone database:

1. Download latest IANA time zone database distribution:

```
git clone https://github.com/eggert/tz
```

   Alternatively, download a specific tzdb version from:

```
 https://www.iana.org/time-zones/repository
```

2. Build timezone tools:

```
cd tz
make TOPDIR=tzdata install
```

3. Generate the compiled binary time zone database:

```
./zic -d ./tzdata/etc/zoneinfo africa antarctica asia australasia backward backzone
etcetera europe factory northamerica pacificnew southamerica systemv
```

4. Create zip archive:

```
pushd ./tzdata/etc
zip -r zoneinfo.zip zoneinfo
popd
```

5. Copy the time zone database to HDFS:

```
hdfs dfs -mkdir -p /tzdb/latest
hdfs dfs -copyFromLocal ./tzdata/etc/zoneinfo.zip /tzdb/latest
```

6. Set the `--hdfs_zone_info_zip` startup flag to `/tzdb/latest/zoneinfo.zip` as an `impalad` safety valve.
7. Perform a full restart of Impala service.

Configuring custom time zone aliases:

1. Create a `tzalias.conf` config file that contains time zone alias definitions formatted as *`ALIAS = DEFINITION`*. For example:

```
#
# Define aliases for existing timezone names:
```

```
#
Universal Coordinated Time = UTC
Mideast/Riyadh89 = Asia/Riyadh
PDT = America/Los_Angeles
#
# Define aliases as UTC offsets in seconds:
#
GMT-01:00 = 3600
GMT+01:00 = -3600
```

2. Copy the config file to HDFS:

```
hdfs dfs -mkdir -p /tzdb
hdfs dfs -copyFromLocal tzalias.conf /tzdb
```

3. Set the `--hdfs_zone_alias_conf` startup flag to `/tzdb/tzalias.conf` as an `impalad` safety valve.
4. Perform a full restart of Impala service.

Added in: CDH 6.1 / Impala 3.1

TINYINT Data Type

A 1-byte integer data type used in `CREATE TABLE` and `ALTER TABLE` statements.

Syntax:

In the column definition of a `CREATE TABLE` statement:

```
column_name TINYINT
```

Range: -128 .. 127. There is no `UNSIGNED` subtype.

Conversions: Impala automatically converts to a larger integer type (`SMALLINT`, `INT`, or `BIGINT`) or a floating-point type (`FLOAT` or `DOUBLE`) automatically. Use `CAST()` to convert to `STRING` or `TIMESTAMP`. Casting an integer or floating-point value `N` to `TIMESTAMP` produces a value that is `N` seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting `--use_local_tz_for_unix_timestamp_conversions=true` is in effect, the resulting `TIMESTAMP` represents a date and time in the local time zone.

Impala does not return column overflows as `NULL`, so that customers can distinguish between `NULL` data and overflow conditions similar to how they do so with traditional database systems. Impala returns the largest or smallest value in the range for the type. For example, valid values for a `tinyint` range from -128 to 127. In Impala, a `tinyint` with a value of -200 returns -128 rather than `NULL`. A `tinyint` with a value of 200 returns 127.

Usage notes:

For a convenient and automated way to check the bounds of the `TINYINT` type, call the functions `MIN_TINYINT()` and `MAX_TINYINT()`.

If an integer value is too large to be represented as a `TINYINT`, use a `SMALLINT` instead.

NULL considerations: Casting any non-numeric value to this type produces a `NULL` value.

Examples:

```
CREATE TABLE t1 (x TINYINT);
SELECT CAST(100 AS TINYINT);
```

Parquet considerations:

Physically, Parquet files represent `TINYINT` and `SMALLINT` values as 32-bit integers. Although Impala rejects attempts to insert out-of-range values into such columns, if you create a new table with the `CREATE TABLE ... LIKE PARQUET` syntax, any `TINYINT` or `SMALLINT` columns in the original table turn into `INT` columns in the new table.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 1-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Related information:

Numeric Literals on page 167, TINYINT Data Type on page 136, SMALLINT Data Type on page 122, INT Data Type on page 117, BIGINT Data Type on page 105, DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only) on page 109, Impala Mathematical Functions on page 397

VARCHAR Data Type (CDH 5.2 or higher only)

A variable-length character type, truncated during processing if necessary to fit within the specified length.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name VARCHAR(max_length)
```

The maximum length you can specify is 65,535.

Partitioning: This type can be used for partition key columns. Because of the efficiency advantage of numeric values over character-based values, if the partition key is a string representation of a number, prefer to use an integer type with sufficient range (INT, BIGINT, and so on) where practical.

HBase considerations: This data type cannot be used with HBase tables.

Parquet considerations:

- This type can be read from and written to Parquet files.
- There is no requirement for a particular level of Parquet.
- Parquet files generated by Impala and containing this type can be freely interchanged with other components such as Hive and MapReduce.
- Parquet data files can contain values that are longer than allowed by the VARCHAR(*n*) length limit. Impala ignores any extra trailing characters when it processes those values during a query.

Text table considerations:

Text data files can contain values that are longer than allowed by the VARCHAR(*n*) length limit. Any extra trailing characters are ignored when Impala processes those values during a query.

Avro considerations:

The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In CDH 5.7 / Impala 2.5 and higher, Impala truncates CHAR and VARCHAR values in Avro tables to (2**31)-1 bytes. If a query encounters a STRING value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Schema evolution considerations:

You can use ALTER TABLE ... CHANGE to switch column data types to and from VARCHAR. You can convert from STRING to VARCHAR(*n*), or from VARCHAR(*n*) to STRING, or from CHAR(*n*) to VARCHAR(*n*), or from VARCHAR(*n*) to CHAR(*n*). When switching back and forth between VARCHAR and CHAR, you can also change the length value. This schema evolution works the same for tables using any file format. If a table contains values longer than the maximum length defined for a VARCHAR column, Impala does not return an error. Any extra trailing characters are ignored when Impala processes those values during a query.

Compatibility:

This type is available in CDH 5.2 / Impala 2.0 or higher.

Internal details: Represented in memory as a byte array with the minimum size needed to represent each value.

Added in: CDH 5.2.0 / Impala 2.0.0

Column statistics considerations: Because the values of this type have variable size, none of the column statistics fields are filled in until you run the COMPUTE STATS statement.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Restrictions:

All data in CHAR and VARCHAR columns must be in a character encoding that is compatible with UTF-8. If you have binary data from another database system (that is, a BLOB type), use a STRING column to hold it.

Examples:

The following examples show how long and short VARCHAR values are treated. Values longer than the maximum specified length are truncated by CAST( ), or when queried from existing data files. Values shorter than the maximum specified length are represented as the actual length of the value, with no extra padding as seen with CHAR values.

```
create table varchar_1 (s varchar(1));
create table varchar_4 (s varchar(4));
create table varchar_20 (s varchar(20));

insert into varchar_1 values (cast('a' as varchar(1))), (cast('b' as varchar(1))),
(cast('hello' as varchar(1))), (cast('world' as varchar(1)));
insert into varchar_4 values (cast('a' as varchar(4))), (cast('b' as varchar(4))),
(cast('hello' as varchar(4))), (cast('world' as varchar(4)));
insert into varchar_20 values (cast('a' as varchar(20))), (cast('b' as varchar(20))),
(cast('hello' as varchar(20))), (cast('world' as varchar(20)));

select * from varchar_1;
+---+
| s |
+---+
| a |
| b |
| h |
| w |
+---+
select * from varchar_4;
+------+
| s    |
+------+
| a    |
| b    |
| hell |
| worl |
+------+
[localhost:21000] > select * from varchar_20;
+-------+
| s     |
+-------+
| a     |
| b     |
| hello |
| world |
+-------+
select concat('[',s,']') as s from varchar_20;
+---------+
| s       |
+---------+
| [a]     |
| [b]     |
| [hello] |
| [world] |
+---------+
```

The following example shows how identical VARCHAR values compare as equal, even if the columns are defined with different maximum lengths. Both tables contain 'a' and 'b' values. The longer 'hello' and 'world' values from the VARCHAR_20 table were truncated when inserted into the VARCHAR_1 table.

```
select s from varchar_1 join varchar_20 using (s);
+-------+
| s     |
+-------+
| a     |
| b     |
+-------+
```

The following examples show how VARCHAR values are freely interchangeable with STRING values in contexts such as comparison operators and built-in functions:

```
select length(cast('foo' as varchar(100))) as length;
+--------+
| length |
+--------+
| 3      |
+--------+
select cast('xyz' as varchar(5)) > cast('abc' as varchar(10)) as greater;
+---------+
| greater |
+---------+
| true    |
+---------+
```

UDF considerations: This type cannot be used for the argument or return type of a user-defined function (UDF) or user-defined aggregate function (UDA).

Related information:

STRING Data Type on page 123, CHAR Data Type (CDH 5.2 or higher only) on page 107, String Literals on page 168, Impala String Functions on page 462

Complex Types (CDH 5.5 or higher only)

Complex types (also referred to as ***nested types***) let you represent multiple data values within a single row/column position. They differ from the familiar column types such as BIGINT and STRING, known as ***scalar types*** or ***primitive types***, which represent a single data value within a given row/column position. Impala supports the complex types ARRAY, MAP, and STRUCT in CDH 5.5 / Impala 2.3 and higher. The Hive UNION type is not currently supported.

Once you understand the basics of complex types, refer to the individual type topics when you need to refresh your memory about syntax and examples:

- ARRAY Complex Type (CDH 5.5 or higher only) on page 102
- STRUCT Complex Type (CDH 5.5 or higher only) on page 124
- MAP Complex Type (CDH 5.5 or higher only) on page 118

Benefits of Impala Complex Types

The reasons for using Impala complex types include the following:

- You already have data produced by Hive or other non-Impala component that uses the complex type column names. You might need to convert the underlying data to Parquet to use it with Impala.
- Your data model originates with a non-SQL programming language or a NoSQL data management system. For example, if you are representing Python data expressed as nested lists, dictionaries, and tuples, those data structures correspond closely to Impala ARRAY, MAP, and STRUCT types.
- Your analytic queries involving multiple tables could benefit from greater locality during join processing. By packing more related data items within each HDFS data block, complex types let join queries avoid the network overhead of the traditional Hadoop shuffle or broadcast join techniques.

The Impala complex type support produces result sets with all scalar values, and the scalar components of complex types can be used with all SQL clauses, such as GROUP BY, ORDER BY, all kinds of joins, subqueries, and inline views. The ability to process complex type data entirely in SQL reduces the need to write application-specific code in Java or other programming languages to deconstruct the underlying data structures.

Overview of Impala Complex Types

The ARRAY and MAP types are closely related: they represent collections with arbitrary numbers of elements, where each element is the same type. In contrast, STRUCT groups together a fixed number of items into a single element. The parts of a STRUCT element (the ***fields***) can be of different types, and each field has a name.

The elements of an ARRAY or MAP, or the fields of a STRUCT, can also be other complex types. You can construct elaborate data structures with up to 100 levels of nesting. For example, you can make an ARRAY whose elements are STRUCTs. Within each STRUCT, you can have some fields that are ARRAY, MAP, or another kind of STRUCT. The Impala documentation uses the terms complex and nested types interchangeably; for simplicity, it primarily uses the term complex types, to encompass all the properties of these types.

When visualizing your data model in familiar SQL terms, you can think of each ARRAY or MAP as a miniature table, and each STRUCT as a row within such a table. By default, the table represented by an ARRAY has two columns, POS to represent ordering of elements, and ITEM representing the value of each element. Likewise, by default, the table represented by a MAP encodes key-value pairs, and therefore has two columns, KEY and VALUE.

The ITEM and VALUE names are only required for the very simplest kinds of ARRAY and MAP columns, ones that hold only scalar values. When the elements within the ARRAY or MAP are of type STRUCT rather than a scalar type, then the result set contains columns with names corresponding to the STRUCT fields rather than ITEM or VALUE.

You write most queries that process complex type columns using familiar join syntax, even though the data for both sides of the join resides in a single table. The join notation brings together the scalar values from a row with the values from the complex type columns for that same row. The final result set contains all scalar values, allowing you to do all the familiar filtering, aggregation, ordering, and so on for the complex data entirely in SQL or using business intelligence tools that issue SQL queries.

Behind the scenes, Impala ensures that the processing for each row is done efficiently on a single host, without the network traffic involved in broadcast or shuffle joins. The most common type of join query for tables with complex type columns is INNER JOIN, which returns results only in those cases where the complex type contains some elements. Therefore, most query examples in this section use either the INNER JOIN clause or the equivalent comma notation.



Note:

Although Impala can query complex types that are present in Parquet files, Impala currently cannot create new Parquet files containing complex types. Therefore, the discussion and examples presume that you are working with existing Parquet data produced through Hive, Spark, or some other source. See Constructing Parquet Files with Complex Columns Using Hive on page 162 for examples of constructing Parquet data files with complex type columns.

For learning purposes, you can create empty tables with complex type columns and practice query syntax, even if you do not have sample data with the required structure.

Design Considerations for Complex Types

When planning to use Impala complex types, and designing the Impala schema, first learn how this kind of schema differs from traditional table layouts from the relational database and data warehousing fields. Because you might have already encountered complex types in a Hadoop context while using Hive for ETL, also learn how to write high-performance analytic queries for complex type data using Impala SQL syntax.

How Complex Types Differ from Traditional Data Warehouse Schemas

Complex types let you associate arbitrary data structures with a particular row. If you are familiar with schema design for relational database management systems or data warehouses, a schema with complex types has the following differences:

- Logically, related values can now be grouped tightly together in the same table.

  In traditional data warehousing, related values were typically arranged in one of two ways:

  – Split across multiple normalized tables. Foreign key columns specified which rows from each table were associated with each other. This arrangement avoided duplicate data and therefore the data was compact, but join queries could be expensive because the related data had to be retrieved from separate locations. (In the case of distributed Hadoop queries, the joined tables might even be transmitted between different hosts in a cluster.)

  – Flattened into a single denormalized table. Although this layout eliminated some potential performance issues by removing the need for join queries, the table typically became larger because values were repeated. The extra data volume could cause performance issues in other parts of the workflow, such as longer ETL cycles or more expensive full-table scans during queries.

  Complex types represent a middle ground that addresses these performance and volume concerns. By physically locating related data within the same data files, complex types increase locality and reduce the expense of join queries. By associating an arbitrary amount of data with a single row, complex types avoid the need to repeat lengthy values such as strings. Because Impala knows which complex type values are associated with each row, you can save storage by avoiding artificial foreign key values that are only used for joins. The flexibility of the `STRUCT`, `ARRAY`, and `MAP` types lets you model familiar constructs such as fact and dimension tables from a data warehouse, and wide tables representing sparse matrixes.

Physical Storage for Complex Types

Physically, the scalar and complex columns in each row are located adjacent to each other in the same Parquet data file, ensuring that they are processed on the same host rather than being broadcast across the network when cross-referenced within a query. This co-location simplifies the process of copying, converting, and backing all the columns up at once. Because of the column-oriented layout of Parquet files, you can still query only the scalar columns of a table without imposing the I/O penalty of reading the (possibly large) values of the composite columns.

Within each Parquet data file, the constituent parts of complex type columns are stored in column-oriented format:

- Each field of a `STRUCT` type is stored like a column, with all the scalar values adjacent to each other and encoded, compressed, and so on using the Parquet space-saving techniques.
- For an `ARRAY` containing scalar values, all those values (represented by the `ITEM` pseudocolumn) are stored adjacent to each other.
- For a `MAP`, the values of the `KEY` pseudocolumn are stored adjacent to each other. If the `VALUE` pseudocolumn is a scalar type, its values are also stored adjacent to each other.
- If an `ARRAY` element, `STRUCT` field, or `MAP` `VALUE` part is another complex type, the column-oriented storage applies to the next level down (or the next level after that, and so on for deeply nested types) where the final elements, fields, or values are of scalar types.

The numbers represented by the `POS` pseudocolumn of an `ARRAY` are not physically stored in the data files. They are synthesized at query time based on the order of the `ARRAY` elements associated with each row.

File Format Support for Impala Complex Types

Currently, Impala queries support complex type data only in the Parquet file format. See Using the Parquet File Format with Impala Tables on page 643 for details about the performance benefits and physical layout of this file format.

Because Impala does not parse the data structures containing nested types for unsupported formats such as text, Avro, SequenceFile, or RCFile, you cannot use data files in these formats with Impala, even if the query does not refer to the nested type columns. Also, if a table using an unsupported format originally contained nested type columns, and then those columns were dropped from the table using `ALTER TABLE ... DROP COLUMN`, any existing data files in the table still contain the nested type data and Impala queries on that table will generate errors.

The one exception to the preceding rule is `COUNT(*)` queries on RCFile tables that include complex types. Such queries are allowed in CDH 5.8 / Impala 2.6 and higher.

You can perform DDL operations for tables involving complex types in most file formats other than Parquet. You cannot create tables in Impala with complex types using text files.

You can have a partitioned table with complex type columns that uses a non-Parquet format, and use `ALTER TABLE` to change the file format to Parquet for individual partitions. When you put Parquet data files into those partitions, Impala can execute queries against that data as long as the query does not involve any of the non-Parquet partitions.

If you use the `parquet-tools` command to examine the structure of a Parquet data file that includes complex types, you see that both `ARRAY` and `MAP` are represented as a `Bag` in Parquet terminology, with all fields marked `Optional` because Impala allows any column to be nullable.

Impala supports either 2-level and 3-level encoding within each Parquet data file. When constructing Parquet data files outside Impala, use either encoding style but do not mix 2-level and 3-level encoding within the same data file.

Choosing Between Complex Types and Normalized Tables

Choosing between multiple normalized fact and dimension tables, or a single table containing complex types, is an important design decision.

- If you are coming from a traditional database or data warehousing background, you might be familiar with how to split up data between tables. Your business intelligence tools might already be optimized for dealing with this kind of multi-table scenario through join queries.
- If you are pulling data from Impala into an application written in a programming language that has data structures analogous to the complex types, such as Python or Java, complex types in Impala could simplify data interchange and improve understandability and reliability of your program logic.
- You might already be faced with existing infrastructure or receive high volumes of data that assume one layout or the other. For example, complex types are popular with web-oriented applications, for example to keep information about an online user all in one place for convenient lookup and analysis, or to deal with sparse or constantly evolving data fields.
- If some parts of the data change over time while related data remains constant, using multiple normalized tables lets you replace certain parts of the data without reloading the entire data set. Conversely, if you receive related data all bundled together, such as in JSON files, using complex types can save the overhead of splitting the related items across multiple tables.
- From a performance perspective:
  - In Parquet tables, Impala can skip columns that are not referenced in a query, avoiding the I/O penalty of reading the embedded data. When complex types are nested within a column, the data is physically divided at a very granular level; for example, a query referring to data nested multiple levels deep in a complex type column does not have to read all the data from that column, only the data for the relevant parts of the column type hierarchy.
  - Complex types avoid the possibility of expensive join queries when data from fact and dimension tables is processed in parallel across multiple hosts. All the information for a row containing complex types is typically to be in the same data block, and therefore does not need to be transmitted across the network when joining fields that are all part of the same row.
  - The tradeoff with complex types is that fewer rows fit in each data block. Whether it is better to have more data blocks with fewer rows, or fewer data blocks with many rows, depends on the distribution of your data and the characteristics of your query workload. If the complex columns are rarely referenced, using them might lower efficiency. If you are seeing low parallelism due to a small volume of data (relatively few data blocks) in each table partition, increasing the row size by including complex columns might produce more data blocks and thus spread the work more evenly across the cluster. See Scalability Considerations for Impala on page 605 for more on this advanced topic.

Differences Between Impala and Hive Complex Types

Impala can query Parquet tables containing `ARRAY`, `STRUCT`, and `MAP` columns produced by Hive. There are some differences to be aware of between the Impala SQL and HiveQL syntax for complex types, primarily for queries.

Impala supports a subset of the syntax that Hive supports for specifying ARRAY, STRUCT, and MAP types in the CREATE TABLE statements.

Because Impala STRUCT columns include user-specified field names, you use the NAMED_STRUCT( ) constructor in Hive rather than the STRUCT( ) constructor when you populate an Impala STRUCT column using a Hive INSERT statement.

The Hive UNION type is not currently supported in Impala.

While Impala usually aims for a high degree of compatibility with HiveQL query syntax, Impala syntax differs from Hive for queries involving complex types. The differences are intended to provide extra flexibility for queries involving these kinds of tables.

- Impala uses dot notation for referring to element names or elements within complex types, and join notation for cross-referencing scalar columns with the elements of complex types within the same row, rather than the LATERAL VIEW clause and EXPLODE( ) function of HiveQL.
- Using join notation lets you use all the kinds of join queries with complex type columns. For example, you can use a LEFT OUTER JOIN, LEFT ANTI JOIN, or LEFT SEMI JOIN query to evaluate different scenarios where the complex columns do or do not contain any elements.
- You can include references to collection types inside subqueries and inline views. For example, you can construct a FROM clause where one of the “tables” is a subquery against a complex type column, or use a subquery against a complex type column as the argument to an IN or EXISTS clause.
- The Impala pseudocolumn POS lets you retrieve the position of elements in an array along with the elements themselves, equivalent to the POSEXPLODE( ) function of HiveQL. You do not use index notation to retrieve a single array element in a query; the join query loops through the array elements and you use WHERE clauses to specify which elements to return.
- Join clauses involving complex type columns do not require an ON or USING clause. Impala implicitly applies the join key so that the correct array entries or map elements are associated with the correct row from the table.
- Impala does not currently support the UNION complex type.

Limitations and Restrictions for Complex Types

Complex type columns can only be used in tables or partitions with the Parquet file format.

Complex type columns cannot be used as partition key columns in a partitioned table.

When you use complex types with the ORDER BY, GROUP BY, HAVING, or WHERE clauses, you cannot refer to the column name by itself. Instead, you refer to the names of the scalar values within the complex type, such as the ITEM, POS, KEY, or VALUE pseudocolumns, or the field names from a STRUCT.

The maximum depth of nesting for complex types is 100 levels.

The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.

For ideal performance and scalability, use small or medium-sized collections, where all the complex columns contain at most a few hundred megabytes per row. Remember, all the columns of a row are stored in the same HDFS data block, whose size in Parquet files typically ranges from 256 MB to 1 GB.

Including complex type columns in a table introduces some overhead that might make queries that do not reference those columns somewhat slower than Impala queries against tables without any complex type columns. Expect at most a 2x slowdown compared to tables that do not have any complex type columns.

Currently, the COMPUTE STATS statement does not collect any statistics for columns containing complex types. Impala uses heuristics to construct execution plans involving complex type columns.

Currently, Impala built-in functions and user-defined functions cannot accept complex types as parameters or produce them as function return values. (When the complex type values are materialized in an Impala result set, the result set contains the scalar components of the values, such as the POS or ITEM for an ARRAY, the KEY or VALUE for a MAP, or the fields of a STRUCT; these scalar data items *can* be used with built-in functions and UDFs as usual.)

Impala currently cannot write new data files containing complex type columns. Therefore, although the SELECT statement works for queries involving complex type columns, you cannot use a statement form that writes data to complex type columns, such as CREATE TABLE AS SELECT or INSERT ... SELECT. To create data files containing complex type data, use the Hive INSERT statement, or another ETL mechanism such as MapReduce jobs, Spark jobs, Pig, and so on.

Currently, Impala can query complex type columns only from Parquet tables or Parquet partitions within partitioned tables. Although you can use complex types in tables with Avro, text, and other file formats as part of your ETL pipeline, for example as intermediate tables populated through Hive, doing analytics through Impala requires that the data eventually ends up in a Parquet table. The requirement for Parquet data files means that you can use complex types with Impala tables hosted on other kinds of file storage systems such as Isilon and Amazon S3, but you cannot use Impala to query complex types from HBase tables. See File Format Support for Impala Complex Types on page 141 for more details.

Using Complex Types in SQL

When using complex types through SQL in Impala, you learn the notation for < > delimiters for the complex type columns in CREATE TABLE statements, and how to construct join queries to "unpack" the scalar values nested inside the complex data structures. You might need to condense a traditional RDBMS or data warehouse schema into a smaller number of Parquet tables, and use Hive, Spark, Pig, or other mechanism outside Impala to populate the tables with data.

Complex Type Syntax for DDL Statements

The definition of *data_type*, as seen in the CREATE TABLE and ALTER TABLE statements, now includes complex types in addition to primitive types:

```
  primitive_type
| array_type
| map_type
| struct_type
```

Unions are not currently supported.

Array, struct, and map column type declarations are specified in the CREATE TABLE statement. You can also add or change the type of complex columns through the ALTER TABLE statement.

Currently, Impala queries allow complex types only in tables that use the Parquet format. If an Impala query encounters complex types in a table or partition using another file format, the query returns a runtime error.

You can use ALTER TABLE ... SET FILEFORMAT PARQUET to change the file format of an existing table containing complex types to Parquet, after which Impala can query it. Make sure to load Parquet files into the table after changing the file format, because the ALTER TABLE ... SET FILEFORMAT statement does not convert existing data to the new file format.

Partitioned tables can contain complex type columns. All the partition key columns must be scalar types.

Because use cases for Impala complex types require that you already have Parquet data files produced outside of Impala, you can use the Impala CREATE TABLE LIKE PARQUET syntax to produce a table with columns that match the structure of an existing Parquet file, including complex type columns for nested data structures. Remember to include the STORED AS PARQUET clause in this case, because even with CREATE TABLE LIKE PARQUET, the default file format of the resulting table is still text.

You cannot use the CREATE TABLE AS SELECT syntax to create a table with nested type columns because the complex columns are omitted from the result set of an Impala SELECT * or SELECT *col_name* query, and because Impala currently does not support writing Parquet files with complex type columns,



Note:

Once you have a table set up with complex type columns, use the DESCRIBE and SHOW CREATE TABLE statements to see the correct notation with < and > delimiters and comma and colon separators within the complex type definitions. If you do not have existing data with the same layout as the table, you can query the empty table to practice with the notation for the SELECT statement. In the SELECT list, you use dot notation and pseudocolumns such as ITEM, KEY, and VALUE for referring to items within the complex type columns. In the FROM clause, you use join notation to construct table aliases for any referenced ARRAY and MAP columns.

For example, when defining a table that holds contact information, you might represent phone numbers differently depending on the expected layout and relationships of the data, and how well you can predict those properties in advance.

Here are different ways that you might represent phone numbers in a traditional relational schema, with equivalent representations using complex types.

The traditional, simplest way to represent phone numbers in a relational table is to store all contact info in a single table, with all columns having scalar types, and each potential phone number represented as a separate column. In this example, each person can only have these 3 types of phone numbers. If the person does not have a particular kind of phone number, the corresponding column is NULL for that row.

```
CREATE TABLE contacts_fixed_phones
(
    id BIGINT
  , name STRING
  , address STRING
  , home_phone STRING
  , work_phone STRING
  , mobile_phone STRING
) STORED AS PARQUET;
```

Figure 1: Traditional Relational Representation of Phone Numbers: Single Table

Using a complex type column to represent the phone numbers adds some extra flexibility. Now there could be an unlimited number of phone numbers. Because the array elements have an order but not symbolic names, you could decide in advance that phone_number[0] is the home number, [1] is the work number, [2] is the mobile number, and so on. (In subsequent examples, you will see how to create a more flexible naming scheme using other complex type variations, such as a MAP or an ARRAY where each element is a STRUCT.)

```
CREATE TABLE contacts_array_of_phones
(
    id BIGINT
  , name STRING
  , address STRING
  , phone_number ARRAY < STRING >
) STORED AS PARQUET;
```

Figure 2: An Array of Phone Numbers

Another way to represent an arbitrary set of phone numbers is with a MAP column. With a MAP, each element is associated with a key value that you specify, which could be a numeric, string, or other scalar type. This example uses a STRING key to give each phone number a name, such as 'home' or 'mobile'. A query could filter the data based on the key values, or display the key values in reports.

```
CREATE TABLE contacts_unlimited_phones
(
  id BIGINT, name STRING, address STRING, phone_number MAP < STRING,STRING >
```

```
) STORED AS PARQUET;
```

Figure 3: A Map of Phone Numbers

If you are an experienced database designer, you already know how to work around the limitations of the single-table schema from Figure 1: Traditional Relational Representation of Phone Numbers: Single Table on page 145. By normalizing the schema, with the phone numbers in their own table, you can associate an arbitrary set of phone numbers with each person, and associate additional details with each phone number, such as whether it is a home, work, or mobile phone.

The flexibility of this approach comes with some drawbacks. Reconstructing all the data for a particular person requires a join query, which might require performance tuning on Hadoop because the data from each table might be transmitted from a different host. Data management tasks such as backups and refreshing the data require dealing with multiple tables instead of a single table.

This example illustrates a traditional database schema to store contact info normalized across 2 tables. The fact table establishes the identity and basic information about person. A dimension table stores information only about phone numbers, using an ID value to associate each phone number with a person ID from the fact table. Each person can have 0, 1, or many phones; the categories are not restricted to a few predefined ones; and the phone table can contain as many columns as desired, to represent all sorts of details about each phone number.

```
CREATE TABLE fact_contacts (id BIGINT, name STRING, address STRING) STORED AS PARQUET;
CREATE TABLE dim_phones
(
    contact_id BIGINT
  , category STRING
  , international_code STRING
  , area_code STRING
  , exchange STRING
  , extension STRING
  , mobile BOOLEAN
  , carrier STRING
  , current BOOLEAN
  , service_start_date TIMESTAMP
  , service_end_date TIMESTAMP
)
STORED AS PARQUET;
```

Figure 4: Traditional Relational Representation of Phone Numbers: Normalized Tables

To represent a schema equivalent to the one from Figure 4: Traditional Relational Representation of Phone Numbers: Normalized Tables on page 146 using complex types, this example uses an ARRAY where each array element is a STRUCT. As with the earlier complex type examples, each person can have an arbitrary set of associated phone numbers. Making each array element into a STRUCT lets us associate multiple data items with each phone number, and give a separate name and type to each data item. The STRUCT fields of the ARRAY elements reproduce the columns of the dimension table from the previous example.

You can do all the same kinds of queries with the complex type schema as with the normalized schema from the previous example. The advantages of the complex type design are in the areas of convenience and performance. Now your backup and ETL processes only deal with a single table. When a query uses a join to cross-reference the information about a person with their associated phone numbers, all the relevant data for each row resides in the same HDFS data block, meaning each row can be processed on a single host without requiring network transmission.

```
CREATE TABLE contacts_detailed_phones
(
  id BIGINT, name STRING, address STRING
    , phone ARRAY < STRUCT <
        category: STRING
      , international_code: STRING
      , area_code: STRING
      , exchange: STRING
```

```
      , extension: STRING
      , mobile: BOOLEAN
      , carrier: STRING
      , current: BOOLEAN
      , service_start_date: TIMESTAMP
      , service_end_date: TIMESTAMP
    >>
) STORED AS PARQUET;
```

Figure 5: Phone Numbers Represented as an Array of Structs

SQL Statements that Support Complex Types

The Impala SQL statements that support complex types are currently CREATE TABLE, ALTER TABLE, DESCRIBE, LOAD DATA, and SELECT. That is, currently Impala can create or alter tables containing complex type columns, examine the structure of a table containing complex type columns, import existing data files containing complex type columns into a table, and query Parquet tables containing complex types.

Impala currently cannot write new data files containing complex type columns. Therefore, although the SELECT statement works for queries involving complex type columns, you cannot use a statement form that writes data to complex type columns, such as CREATE TABLE AS SELECT or INSERT ... SELECT. To create data files containing complex type data, use the Hive INSERT statement, or another ETL mechanism such as MapReduce jobs, Spark jobs, Pig, and so on.

DDL Statements and Complex Types

Column specifications for complex or nested types use < and > delimiters:

```
-- What goes inside the < > for an ARRAY is a single type, either a scalar or another
-- complex type (ARRAY, STRUCT, or MAP).
CREATE TABLE array_t
(
  id BIGINT,
  a1 ARRAY <STRING>,
  a2 ARRAY <BIGINT>,
  a3 ARRAY <TIMESTAMP>,
  a4 ARRAY <STRUCT <f1: STRING, f2: INT, f3: BOOLEAN>>
)
STORED AS PARQUET;

-- What goes inside the < > for a MAP is two comma-separated types specifying the types
 of the key-value pair:
-- a scalar type representing the key, and a scalar or complex type representing the
value.
CREATE TABLE map_t
(
  id BIGINT,
  m1 MAP <STRING, STRING>,
  m2 MAP <STRING, BIGINT>,
  m3 MAP <BIGINT, STRING>,
  m4 MAP <BIGINT, BIGINT>,
  m5 MAP <STRING, ARRAY <STRING>>
)
STORED AS PARQUET;

-- What goes inside the < > for a STRUCT is a comma-separated list of fields, each field
 defined as
-- name:type. The type can be a scalar or a complex type. The field names for each STRUCT
 do not clash
-- with the names of table columns or fields in other STRUCTs. A STRUCT is most often
used inside
-- an ARRAY or a MAP rather than as a top-level column.
CREATE TABLE struct_t
(
  id BIGINT,
  s1 STRUCT <f1: STRING, f2: BIGINT>,
  s2 ARRAY <STRUCT <f1: INT, f2: TIMESTAMP>>,
  s3 MAP <BIGINT, STRUCT <name: STRING, birthday: TIMESTAMP>>
)
```

```
STORED AS PARQUET;
```

Queries and Complex Types

The result set of an Impala query always contains all scalar types; the elements and fields within any complex type queries must be “unpacked” using join queries. A query cannot directly retrieve the entire value for a complex type column. Impala returns an error in this case. Queries using `SELECT *` are allowed for tables with complex types, but the columns with complex types are skipped.

The following example shows how referring directly to a complex type column returns an error, while `SELECT *` on the same table succeeds, but only retrieves the scalar columns.



Note: Many of the complex type examples refer to tables such as `CUSTOMER` and `REGION` adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types on page 160 for the table definitions.

```
SELECT c_orders FROM customer LIMIT 1;
ERROR: AnalysisException: Expr 'c_orders' in select list returns a complex type
'ARRAY<STRUCT<o_orderkey:BIGINT,o_orderstatus:STRING, ...
l_receiptdate:STRING,l_shipinstruct:STRING,l_shipmode:STRING,l_comment:STRING>>>>'.
Only scalar types are allowed in the select list.

-- Original column has several scalar and one complex column.
DESCRIBE customer;
+--------------+-----------------------------------+
| name         | type                              |
+--------------+-----------------------------------+
| c_custkey    | bigint                            |
| c_name       | string                            |
...
| c_orders     | array<struct<                     |
|              |   o_orderkey:bigint,              |
|              |   o_orderstatus:string,           |
|              |   o_totalprice:decimal(12,2),     |
...
|              | >>                                |
+--------------+-----------------------------------+

-- When we SELECT * from that table, only the scalar columns come back in the result
set.
CREATE TABLE select_star_customer STORED AS PARQUET AS SELECT * FROM customer;
+------------------------+
| summary                |
+------------------------+
| Inserted 150000 row(s) |
+------------------------+

-- The c_orders column, being of complex type, was not included in the SELECT * result
 set.
DESC select_star_customer;
+--------------+---------------+
| name         | type          |
+--------------+---------------+
| c_custkey    | bigint        |
| c_name       | string        |
| c_address    | string        |
| c_nationkey  | smallint      |
| c_phone      | string        |
| c_acctbal    | decimal(12,2) |
| c_mktsegment | string        |
| c_comment    | string        |
+--------------+---------------+
```

References to fields within STRUCT columns use dot notation. If the field name is unambiguous, you can omit qualifiers such as table name, column name, or even the ITEM or VALUE pseudocolumn names for STRUCT elements inside an ARRAY or a MAP.

```
SELECT id, address.city FROM customers WHERE address.zip = 94305;
```

References to elements within ARRAY columns use the ITEM pseudocolumn:

```
select r_name, r_nations.item.n_name from region, region.r_nations limit 7;
+--------+----------------+
| r_name | item.n_name    |
+--------+----------------+
| EUROPE | UNITED KINGDOM |
| EUROPE | RUSSIA         |
| EUROPE | ROMANIA        |
| EUROPE | GERMANY        |
| EUROPE | FRANCE         |
| ASIA   | VIETNAM        |
| ASIA   | CHINA          |
+--------+----------------+
```

References to fields within MAP columns use the KEY and VALUE pseudocolumns. In this example, once the query establishes the alias MAP_FIELD for a MAP column with a STRING key and an INT value, the query can refer to MAP_FIELD.KEY and MAP_FIELD.VALUE, which have zero, one, or many instances for each row from the containing table.

```
DESCRIBE table_0;
+---------+-----------------------+
| name    | type                  |
+---------+-----------------------+
| field_0 | string                |
| field_1 | map<string,int>       |
...

SELECT field_0, map_field.key, map_field.value
  FROM table_0, table_0.field_1 AS map_field
WHERE length(field_0) = 1
LIMIT 10;
+---------+-----------+-------+
| field_0 | key       | value |
+---------+-----------+-------+
| b       | gshsgkvd  | NULL  |
| b       | twrtcxj6  | 18    |
| b       | 2vp5      | 39    |
| b       | fh0s      | 13    |
| v       | 2         | 41    |
| v       | 8b58mz    | 20    |
| v       | hw        | 16    |
| v       | 65l388pyt | 29    |
| v       | 03k68g91z | 30    |
| v       | r2hlg5b   | NULL  |
+---------+-----------+-------+
```

When complex types are nested inside each other, you use a combination of joins, pseudocolumn names, and dot notation to refer to specific fields at the appropriate level. This is the most frequent form of query syntax for complex columns, because the typical use case involves two levels of complex types, such as an ARRAY of STRUCT elements.

```
SELECT id, phone_numbers.area_code FROM contact_info_many_structs INNER JOIN
contact_info_many_structs.phone_numbers phone_numbers LIMIT 3;
```

You can express relationships between ARRAY and MAP columns at different levels as joins. You include comparison operators between fields at the top level and within the nested type columns so that Impala can do the appropriate join operation.



Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types on page 160 for the table definitions.

For example, the following queries work equivalently. They each return customer and order data for customers that have at least one order.

```
SELECT c.c_name, o.o_orderkey FROM customer c, c.c_orders o LIMIT 5;
+--------------------+------------+
| c_name             | o_orderkey |
+--------------------+------------+
| Customer#000072578 | 558821     |
| Customer#000072578 | 2079810    |
| Customer#000072578 | 5768068    |
| Customer#000072578 | 1805604    |
| Customer#000072578 | 3436389    |
+--------------------+------------+

SELECT c.c_name, o.o_orderkey FROM customer c INNER JOIN c.c_orders o LIMIT 5;
+--------------------+------------+
| c_name             | o_orderkey |
+--------------------+------------+
| Customer#000072578 | 558821     |
| Customer#000072578 | 2079810    |
| Customer#000072578 | 5768068    |
| Customer#000072578 | 1805604    |
| Customer#000072578 | 3436389    |
+--------------------+------------+
```

The following query using an outer join returns customers that have orders, plus customers with no orders (no entries in the C_ORDERS array):

```
SELECT c.c_custkey, o.o_orderkey
  FROM customer c LEFT OUTER JOIN c.c_orders o
LIMIT 5;
+-----------+------------+
| c_custkey | o_orderkey |
+-----------+------------+
| 60210     | NULL       |
| 147873    | NULL       |
| 72578     | 558821     |
| 72578     | 2079810    |
| 72578     | 5768068    |
+-----------+------------+
```

The following query returns *only* customers that have no orders. (With LEFT ANTI JOIN or LEFT SEMI JOIN, the query can only refer to columns from the left-hand table, because by definition there is no matching information in the right-hand table.)

```
SELECT c.c_custkey, c.c_name
  FROM customer c LEFT ANTI JOIN c.c_orders o
LIMIT 5;
+-----------+--------------------+
| c_custkey | c_name             |
+-----------+--------------------+
| 60210     | Customer#000060210 |
| 147873    | Customer#000147873 |
| 141576    | Customer#000141576 |
| 85365     | Customer#000085365 |
| 70998     | Customer#000070998 |
+-----------+--------------------+
```

You can also perform correlated subqueries to examine the properties of complex type columns for each row in the result set.

Count the number of orders per customer. Note the correlated reference to the table alias C. The COUNT(*) operation applies to all the elements of the C_ORDERS array for the corresponding row, avoiding the need for a GROUP BY clause.

```
select c_name, howmany FROM customer c, (SELECT COUNT(*) howmany FROM c.c_orders) v
limit 5;
+--------------------+---------+
| c_name             | howmany |
+--------------------+---------+
| Customer#000030065 | 15      |
| Customer#000065455 | 18      |
| Customer#000113644 | 21      |
| Customer#000111078 | 0       |
| Customer#000024621 | 0       |
+--------------------+---------+
```

Count the number of orders per customer, ignoring any customers that have not placed any orders:

```
SELECT c_name, howmany_orders
FROM
  customer c,
  (SELECT COUNT(*) howmany_orders FROM c.c_orders) subq1
WHERE howmany_orders > 0
LIMIT 5;
+--------------------+----------------+
| c_name             | howmany_orders |
+--------------------+----------------+
| Customer#000072578 | 7              |
| Customer#000046378 | 26             |
| Customer#000069815 | 11             |
| Customer#000079058 | 12             |
| Customer#000092239 | 26             |
+--------------------+----------------+
```

Count the number of line items in each order. The reference to C.C_ORDERS in the FROM clause is needed because the O_ORDERKEY field is a member of the elements in the C_ORDERS array. The subquery labelled SUBQ1 is correlated: it is re-evaluated for the C_ORDERS.O_LINEITEMS array from each row of the CUSTOMERS table.

```
SELECT c_name, o_orderkey, howmany_line_items
FROM
  customer c,
  c.c_orders t2,
  (SELECT COUNT(*) howmany_line_items FROM c.c_orders.o_lineitems) subq1
WHERE howmany_line_items > 0
LIMIT 5;
+--------------------+------------+--------------------+
| c_name             | o_orderkey | howmany_line_items |
+--------------------+------------+--------------------+
| Customer#000020890 | 1884930    | 95                 |
| Customer#000020890 | 4570754    | 95                 |
| Customer#000020890 | 3771072    | 95                 |
| Customer#000020890 | 2555489    | 95                 |
| Customer#000020890 | 919171     | 95                 |
+--------------------+------------+--------------------+
```

Get the number of orders, the average order price, and the maximum items in any order per customer. For this example, the subqueries labelled SUBQ1 and SUBQ2 are correlated: they are re-evaluated for each row from the original CUSTOMER table, and only apply to the complex columns associated with that row.

```
SELECT c_name, howmany, average_price, most_items
FROM
  customer c,
  (SELECT COUNT(*) howmany, AVG(o_totalprice) average_price FROM c.c_orders) subq1,
  (SELECT MAX(l_quantity) most_items FROM c.c_orders.o_lineitems ) subq2
LIMIT 5;
+--------------------+---------+---------------+------------+
| c_name             | howmany | average_price | most_items |
+--------------------+---------+---------------+------------+
| Customer#000030065 | 15      | 128908.34     | 50.00      |
```

```
| Customer#000088191 | 0       | NULL          | NULL       |
| Customer#000101555 | 10      | 164250.31     | 50.00      |
| Customer#000022092 | 0       | NULL          | NULL       |
| Customer#000036277 | 27      | 166040.06     | 50.00      |
+--------------------+---------+---------------+------------+
```

For example, these queries show how to access information about the ARRAY elements within the CUSTOMER table from the “nested TPC-H” schema, starting with the initial ARRAY elements and progressing to examine the STRUCT fields of the ARRAY, and then the elements nested within another ARRAY of STRUCT:

```
-- How many orders does each customer have?
-- The type of the ARRAY column doesn't matter, this is just counting the elements.
SELECT c_custkey, count(*)
  FROM customer, customer.c_orders
GROUP BY c_custkey
LIMIT 5;
+-----------+----------+
| c_custkey | count(*) |
+-----------+----------+
| 61081     | 21       |
| 115987    | 15       |
| 69685     | 19       |
| 109124    | 15       |
| 50491     | 12       |
+-----------+----------+

-- How many line items are part of each customer order?
-- Now we examine a field from a STRUCT nested inside the ARRAY.
SELECT c_custkey, c_orders.o_orderkey, count(*)
  FROM customer, customer.c_orders c_orders, c_orders.o_lineitems
GROUP BY c_custkey, c_orders.o_orderkey
LIMIT 5;
+-----------+------------+----------+
| c_custkey | o_orderkey | count(*) |
+-----------+------------+----------+
| 63367     | 4985959    | 7        |
| 53989     | 1972230    | 2        |
| 143513    | 5750498    | 5        |
| 17849     | 4857989    | 1        |
| 89881     | 1046437    | 1        |
+-----------+------------+----------+

-- What are the line items in each customer order?
-- One of the STRUCT fields inside the ARRAY is another
-- ARRAY containing STRUCT elements. The join finds
-- all the related items from both levels of ARRAY.
SELECT c_custkey, o_orderkey, l_partkey
  FROM customer, customer.c_orders, c_orders.o_lineitems
LIMIT 5;
+-----------+------------+-----------+
| c_custkey | o_orderkey | l_partkey |
+-----------+------------+-----------+
| 113644    | 2738497    | 175846    |
| 113644    | 2738497    | 27309     |
| 113644    | 2738497    | 175873    |
| 113644    | 2738497    | 88559     |
| 113644    | 2738497    | 8032      |
+-----------+------------+-----------+
```

Pseudocolumns for ARRAY and MAP Types

Each element in an ARRAY type has a position, indexed starting from zero, and a value. Each element in a MAP type represents a key-value pair. Impala provides pseudocolumns that let you retrieve this metadata as part of a query, or filter query results by including such things in a WHERE clause. You refer to the pseudocolumns as part of qualified column names in queries:

- ITEM: The value of an array element. If the ARRAY contains STRUCT elements, you can refer to either *array_name*.ITEM.*field_name* or use the shorthand *array_name*.*field_name*.
- POS: The position of an element within an array.

- KEY: The value forming the first part of a key-value pair in a map. It is not necessarily unique.
- VALUE: The data item forming the second part of a key-value pair in a map. If the VALUE part of the MAP element is a STRUCT, you can refer to either *map_name*.VALUE.*field_name* or use the shorthand *map_name*.*field_name*.

ITEM and POS Pseudocolumns

When an ARRAY column contains STRUCT elements, you can refer to a field within the STRUCT using a qualified name of the form *array_column*.*field_name*. If the ARRAY contains scalar values, Impala recognizes the special name *array_column*.ITEM to represent the value of each scalar array element. For example, if a column contained an ARRAY where each element was a STRING, you would use *array_name*.ITEM to refer to each scalar value in the SELECT list, or the WHERE or other clauses.

This example shows a table with two ARRAY columns whose elements are of the scalar type STRING. When referring to the values of the array elements in the SELECT list, WHERE clause, or ORDER BY clause, you use the ITEM pseudocolumn because within the array, the individual elements have no defined names.

```
create TABLE persons_of_interest
(
person_id BIGINT,
aliases ARRAY <STRING>,
associates ARRAY <STRING>,
real_name STRING
)
STORED AS PARQUET;

-- Get all the aliases of each person.
SELECT real_name, aliases.ITEM
  FROM persons_of_interest, persons_of_interest.aliases
ORDER BY real_name, aliases.item;

-- Search for particular associates of each person.
SELECT real_name, associates.ITEM
  FROM persons_of_interest, persons_of_interest.associates
WHERE associates.item LIKE '% MacGuffin';
```

Because an array is inherently an ordered data structure, Impala recognizes the special name *array_column*.POS to represent the numeric position of each element within the array. The POS pseudocolumn lets you filter or reorder the result set based on the sequence of array elements.

The following example uses a table from a flattened version of the TPC-H schema. The REGION table only has a few rows, such as one row for Europe and one for Asia. The row for each region represents all the countries in that region as an ARRAY of STRUCT elements:

```
[localhost:21000] > desc region;
+-------------+-------------------------------------------------------------------+
| name        | type                                                              |
+-------------+-------------------------------------------------------------------+
| r_regionkey | smallint                                                          |
| r_name      | string                                                            |
| r_comment   | string                                                            |
| r_nations   | array<struct<n_nationkey:smallint,n_name:string,n_comment:string>> |
+-------------+-------------------------------------------------------------------+
```

To find the countries within a specific region, you use a join query. To find out the order of elements in the array, you also refer to the POS pseudocolumn in the select list:

```
[localhost:21000] > SELECT r1.r_name, r2.n_name, r2.POS
                  > FROM region r1 INNER JOIN r1.r_nations r2
                  > WHERE r1.r_name = 'ASIA';
+--------+-----------+-----+
| r_name | n_name    | pos |
+--------+-----------+-----+
| ASIA   | VIETNAM   | 0   |
| ASIA   | CHINA     | 1   |
| ASIA   | JAPAN     | 2   |
```

```
| ASIA   | INDONESIA | 3   |
| ASIA   | INDIA     | 4   |
+--------+-----------+-----+
```

Once you know the positions of the elements, you can use that information in subsequent queries, for example to change the ordering of results from the complex type column or to filter certain elements from the array:

```
[localhost:21000] > SELECT r1.r_name, r2.n_name, r2.POS
                  > FROM region r1 INNER JOIN r1.r_nations r2
                  > WHERE r1.r_name = 'ASIA'
                  > ORDER BY r2.POS DESC;
+--------+-----------+-----+
| r_name | n_name    | pos |
+--------+-----------+-----+
| ASIA   | INDIA     | 4   |
| ASIA   | INDONESIA | 3   |
| ASIA   | JAPAN     | 2   |
| ASIA   | CHINA     | 1   |
| ASIA   | VIETNAM   | 0   |
+--------+-----------+-----+
[localhost:21000] > SELECT r1.r_name, r2.n_name, r2.POS
                  > FROM region r1 INNER JOIN r1.r_nations r2
                  > WHERE r1.r_name = 'ASIA' AND r2.POS BETWEEN 1 and 3;
+--------+-----------+-----+
| r_name | n_name    | pos |
+--------+-----------+-----+
| ASIA   | CHINA     | 1   |
| ASIA   | JAPAN     | 2   |
| ASIA   | INDONESIA | 3   |
+--------+-----------+-----+
```

KEY and VALUE Pseudocolumns

The MAP data type is suitable for representing sparse or wide data structures, where each row might only have entries for a small subset of named fields. Because the element names (the map keys) vary depending on the row, a query must be able to refer to both the key and the value parts of each key-value pair. The KEY and VALUE pseudocolumns let you refer to the parts of the key-value pair independently within the query, as *map_column*.KEY and *map_column*.VALUE.

The KEY must always be a scalar type, such as STRING, BIGINT, or TIMESTAMP. It can be NULL. Values of the KEY field are not necessarily unique within the same MAP. You apply any required DISTINCT, GROUP BY, and other clauses in the query, and loop through the result set to process all the values matching any specified keys.

The VALUE can be either a scalar type or another complex type. If the VALUE is a STRUCT, you can construct a qualified name *map_column*.VALUE.*struct_field* to refer to the individual fields inside the value part. If the VALUE is an ARRAY or another MAP, you must include another join condition that establishes a table alias for *map_column*.VALUE, and then construct another qualified name using that alias, for example *table_alias*.ITEM or *table_alias*.KEY and *table_alias*.VALUE

The following example shows different ways to access a MAP column using the KEY and VALUE pseudocolumns. The DETAILS column has a STRING first part with short, standardized values such as 'Recurring', 'Lucid', or 'Anxiety'. This is the "key" that is used to look up particular kinds of elements from the MAP. The second part, also a STRING, is a longer free-form explanation. Impala gives you the standard pseudocolumn names KEY and VALUE for the two parts, and you apply your own conventions and interpretations to the underlying values.



Note: If you find that the single-item nature of the VALUE makes it difficult to model your data accurately, the solution is typically to add some nesting to the complex type. For example, to have several sets of key-value pairs, make the column an ARRAY whose elements are MAP. To make a set of key-value pairs that holds more elaborate information, make a MAP column whose VALUE part contains an ARRAY or a STRUCT.

```
CREATE TABLE dream_journal
(
```

```
  dream_id BIGINT,
  details MAP <STRING,STRING>
)
STORED AS PARQUET;


-- What are all the types of dreams that are recorded?
SELECT DISTINCT details.KEY FROM dream_journal, dream_journal.details;

-- How many lucid dreams were recorded?
-- Because there is no GROUP BY, we count the 'Lucid' keys across all rows.
SELECT COUNT(details.KEY)
  FROM dream_journal, dream_journal.details
WHERE details.KEY = 'Lucid';

-- Print a report of a subset of dreams, filtering based on both the lookup key
-- and the detailed value.
SELECT dream_id, details.KEY AS "Dream Type", details.VALUE AS "Dream Summary"
  FROM dream_journal, dream_journal.details
WHERE
  details.KEY IN ('Happy', 'Pleasant', 'Joyous')
  AND details.VALUE LIKE '%childhood%';
```

The following example shows a more elaborate version of the previous table, where the VALUE part of the MAP entry is a STRUCT rather than a scalar type. Now instead of referring to the VALUE pseudocolumn directly, you use dot notation to refer to the STRUCT fields inside it.

```
CREATE TABLE better_dream_journal
(
  dream_id BIGINT,
  details MAP <STRING,STRUCT <summary: STRING, when_happened: TIMESTAMP, duration:
DECIMAL(5,2), woke_up: BOOLEAN> >
)
STORED AS PARQUET;


-- Do more elaborate reporting and filtering by examining multiple attributes within
the same dream.
SELECT dream_id, details.KEY AS "Dream Type", details.VALUE.summary AS "Dream Summary",
 details.VALUE.duration AS "Duration"
  FROM better_dream_journal, better_dream_journal.details
WHERE
  details.KEY IN ('Anxiety', 'Nightmare')
  AND details.VALUE.duration > 60
  AND details.VALUE.woke_up = TRUE;

-- Remember that if the ITEM or VALUE contains a STRUCT, you can reference
-- the STRUCT fields directly without the .ITEM or .VALUE qualifier.
SELECT dream_id, details.KEY AS "Dream Type", details.summary AS "Dream Summary",
details.duration AS "Duration"
  FROM better_dream_journal, better_dream_journal.details
WHERE
  details.KEY IN ('Anxiety', 'Nightmare')
  AND details.duration > 60
  AND details.woke_up = TRUE;
```

Loading Data Containing Complex Types

Because the Impala INSERT statement does not currently support creating new data with complex type columns, or copying existing complex type values from one table to another, you primarily use Impala to query Parquet tables with complex types where the data was inserted through Hive, or create tables with complex types where you already have existing Parquet data files.

If you have created a Hive table with the Parquet file format and containing complex types, use the same table for Impala queries with no changes. If you have such a Hive table in some other format, use a Hive CREATE TABLE AS SELECT ... STORED AS PARQUET or INSERT ... SELECT statement to produce an equivalent Parquet table that Impala can query.

If you have existing Parquet data files containing complex types, located outside of any Impala or Hive table, such as data files created by Spark jobs, you can use an Impala CREATE TABLE ... STORED AS PARQUET statement, followed by an Impala LOAD DATA statement to move the data files into the table. As an alternative, you can use an Impala CREATE EXTERNAL TABLE statement to create a table pointing to the HDFS directory that already contains the data files.

Perhaps the simplest way to get started with complex type data is to take a denormalized table containing duplicated values, and use an INSERT ... SELECT statement to copy the data into a Parquet table and condense the repeated values into complex types. With the Hive INSERT statement, you use the COLLECT_LIST(), NAMED_STRUCT(), and MAP() constructor functions within a GROUP BY query to produce the complex type values. COLLECT_LIST() turns a sequence of values into an ARRAY. NAMED_STRUCT() uses the first, third, and so on arguments as the field names for a STRUCT, to match the field names from the CREATE TABLE statement.



Note: Because Hive currently cannot construct individual rows using complex types through the INSERT ... VALUES syntax, you prepare the data in flat form in a separate table, then copy it to the table with complex columns using INSERT ... SELECT and the complex type constructors. See Constructing Parquet Files with Complex Columns Using Hive on page 162 for examples.

Using Complex Types as Nested Types

The ARRAY, STRUCT, and MAP types can be the top-level types for "nested type" columns. That is, each of these types can contain other complex or scalar types, with multiple levels of nesting to a maximum depth of 100. For example, you can have an array of structures, a map containing other maps, a structure containing an array of other structures, and so on. At the lowest level, there are always scalar types making up the fields of a STRUCT, elements of an ARRAY, and keys and values of a MAP.

Schemas involving complex types typically use some level of nesting for the complex type columns.

For example, to model a relationship like a dimension table and a fact table, you typically use an ARRAY where each array element is a STRUCT. The STRUCT fields represent what would traditionally be columns in a separate joined table. It makes little sense to use a STRUCT as the top-level type for a column, because you could just make the fields of the STRUCT into regular table columns.

Perhaps the only use case for a top-level STRUCT would be to to allow STRUCT fields with the same name as columns to coexist in the same table. The following example shows how a table could have a column named ID, and two separate STRUCT fields also named ID. Because the STRUCT fields are always referenced using qualified names, the identical ID names do not cause a conflict.

```
CREATE TABLE struct_namespaces
(
  id BIGINT
  , s1 STRUCT < id: BIGINT, field1: STRING >
  , s2 STRUCT < id: BIGINT, when_happened: TIMESTAMP >
)
STORED AS PARQUET;

select id, s1.id, s2.id from struct_namespaces;
```

It is common to make the value portion of each key-value pair in a MAP a STRUCT, ARRAY of STRUCT, or other complex type variation. That way, each key in the MAP can be associated with a flexible and extensible data structure. The key values are not predefined ahead of time (other than by specifying their data type). Therefore, the MAP can accommodate a rapidly evolving schema, or sparse data structures where each row contains only a few data values drawn from a large set of possible choices.

Although you can use an ARRAY of scalar values as the top-level column in a table, such a simple array is typically of limited use for analytic queries. The only property of the array elements, aside from the element value, is the ordering sequence available through the POS pseudocolumn. To record any additional item about each array element, such as a TIMESTAMP or a symbolic name, you use an ARRAY of STRUCT rather than of scalar values.

If you are considering having multiple ARRAY or MAP columns, with related items under the same position in each ARRAY or the same key in each MAP, prefer to use a STRUCT to group all the related items into a single ARRAY or MAP.

Doing so avoids the additional storage overhead and potential duplication of key values from having an extra complex type column. Also, because each ARRAY or MAP that you reference in the query SELECT list requires an additional join clause, minimizing the number of complex type columns also makes the query easier to read and maintain, relying more on dot notation to refer to the relevant fields rather than a sequence of join clauses.

For example, here is a table with several complex type columns all at the top level and containing only scalar types. To retrieve every data item for the row requires a separate join for each ARRAY or MAP column. The fields of the STRUCT can be referenced using dot notation, but there is no real advantage to using the STRUCT at the top level rather than just making separate columns FIELD1 and FIELD2.

```
CREATE TABLE complex_types_top_level
(
  id BIGINT,
  a1 ARRAY<INT>,
  a2 ARRAY<STRING>,
  s STRUCT<field1: INT, field2: STRING>,
-- Numeric lookup key for a string value.
  m1 MAP<INT,STRING>,
-- String lookup key for a numeric value.
  m2 MAP<STRING,INT>
)
STORED AS PARQUET;

describe complex_types_top_level;
+------+-----------------+
| name | type            |
+------+-----------------+
| id   | bigint          |
| a1   | array<int>      |
| a2   | array<string>   |
| s    | struct<         |
|      |   field1:int,   |
|      |   field2:string |
|      | >               |
| m1   | map<int,string> |
| m2   | map<string,int> |
+------+-----------------+

select
  id,
  a1.item,
  a2.item,
  s.field1,
  s.field2,
  m1.key,
  m1.value,
  m2.key,
  m2.value
from
  complex_types_top_level,
  complex_types_top_level.a1,
  complex_types_top_level.a2,
  complex_types_top_level.m1,
  complex_types_top_level.m2;
```

For example, here is a table with columns containing an ARRAY of STRUCT, a MAP where each key value is a STRUCT, and a MAP where each key value is an ARRAY of STRUCT.

```
CREATE TABLE nesting_demo
(
  user_id BIGINT,
  family_members ARRAY < STRUCT < name: STRING, email: STRING, date_joined: TIMESTAMP
>>,
  foo map < STRING, STRUCT < f1: INT, f2: INT, f3: TIMESTAMP, f4: BOOLEAN >>,
  gameplay MAP < STRING , ARRAY < STRUCT <
    name: STRING, highest: BIGINT, lives_used: INT, total_spent: DECIMAL(16,2)
  >>>
)
```

```
STORED AS PARQUET;
```

The DESCRIBE statement rearranges the < and > separators and the field names within each STRUCT for easy readability:

```
DESCRIBE nesting_demo;
+----------------+-----------------------------+
| name           | type                        |
+----------------+-----------------------------+
| user_id        | bigint                      |
| family_members | array<struct<               |
|                |   name:string,              |
|                |   email:string,             |
|                |   date_joined:timestamp     |
|                | >>                          |
| foo            | map<string,struct<          |
|                |   f1:int,                   |
|                |   f2:int,                   |
|                |   f3:timestamp,             |
|                |   f4:boolean                |
|                | >>                          |
| gameplay       | map<string,array<struct<    |
|                |   name:string,              |
|                |   highest:bigint,           |
|                |   lives_used:int,           |
|                |   total_spent:decimal(16,2) |
|                | >>>                         |
+----------------+-----------------------------+
```

To query the complex type columns, you use join notation to refer to the lowest-level scalar values. If the value is an ARRAY element, the fully qualified name includes the ITEM pseudocolumn. If the value is inside a MAP, the fully qualified name includes the KEY or VALUE pseudocolumn. Each reference to a different ARRAY or MAP (even if nested inside another complex type) requires an additional join clause.

```
SELECT
-- The lone scalar field doesn't require any dot notation or join clauses.
    user_id
-- Retrieve the fields of a STRUCT inside an ARRAY.
-- The FAMILY_MEMBERS name refers to the FAMILY_MEMBERS table alias defined later in
the FROM clause.
  , family_members.item.name
  , family_members.item.email
  , family_members.item.date_joined
-- Retrieve the KEY and VALUE fields of a MAP, with the value being a STRUCT consisting
 of more fields.
-- The FOO name refers to the FOO table alias defined later in the FROM clause.
  , foo.key
  , foo.value.f1
  , foo.value.f2
  , foo.value.f3
  , foo.value.f4
-- Retrieve the KEY fields of a MAP, and expand the VALUE part into ARRAY items consisting
 of STRUCT fields.
-- The GAMEPLAY name refers to the GAMEPLAY table alias defined later in the FROM clause
 (referring to the MAP item).
-- The GAME_N name refers to the GAME_N table alias defined later in the FROM clause
(referring to the ARRAY
-- inside the MAP item's VALUE part.)
  , gameplay.key
  , game_n.name
  , game_n.highest
  , game_n.lives_used
  , game_n.total_spent
FROM
    nesting_demo
  , nesting_demo.family_members AS family_members
  , nesting_demo.foo AS foo
  , nesting_demo.gameplay AS gameplay
```

```
  , nesting_demo.gameplay.value AS game_n;
```

Once you understand the notation to refer to a particular data item in the SELECT list, you can use the same qualified name to refer to that data item in other parts of the query, such as the WHERE clause, ORDER BY or GROUP BY clauses, or calls to built-in functions. For example, you might frequently retrieve the VALUE part of each MAP item in the SELECT list, while choosing the specific MAP items by running comparisons against the KEY part in the WHERE clause.

Accessing Complex Type Data in Flattened Form Using Views

The layout of complex and nested types is largely a physical consideration. The complex type columns reside in the same data files rather than in separate normalized tables, for your convenience in managing related data sets and performance in querying related data sets. You can use views to treat tables with complex types as if they were flattened. By putting the join logic and references to the complex type columns in the view definition, you can query the same tables using existing queries intended for tables containing only scalar columns. This technique also lets you use tables with complex types with BI tools that are not aware of the data types and query notation for accessing complex type columns.

For example, the variation of the TPC-H schema containing complex types has a table REGION. This table has 5 rows, corresponding to 5 regions such as NORTH AMERICA and AFRICA. Each row has an ARRAY column, where each array item is a STRUCT containing details about a country in that region.

```
DESCRIBE region;
+-------------+-------------------------+
| name        | type                    |
+-------------+-------------------------+
| r_regionkey | smallint                |
| r_name      | string                  |
| r_comment   | string                  |
| r_nations   | array<struct<           |
|             |   n_nationkey:smallint, |
|             |   n_name:string,        |
|             |   n_comment:string      |
|             | >>                      |
+-------------+-------------------------+
```

The same data could be represented in traditional denormalized form, as a single table where the information about each region is repeated over and over, alongside the information about each country. The nested complex types let us avoid the repetition, while still keeping the data in a single table rather than normalizing across multiple tables.

To use this table with a JDBC or ODBC application that expected scalar columns, we could create a view that represented the result set as a set of scalar columns (three columns from the original table, plus three more from the STRUCT fields of the array elements). In the following examples, any column with an R_* prefix is taken unchanged from the original table, while any column with an N_* prefix is extracted from the STRUCT inside the ARRAY.

```
CREATE VIEW region_view AS
  SELECT
    r_regionkey,
    r_name,
    r_comment,
    array_field.item.n_nationkey AS n_nationkey,
    array_field.item.n_name AS n_name,
    array_field.n_comment AS n_comment
FROM
  region, region.r_nations AS array_field;
```

Then we point the application queries at the view rather than the original table. From the perspective of the view, there are 25 rows in the result set, one for each nation in each region, and queries can refer freely to fields related to the region or the nation.

```
-- Retrieve info such as the nation name from the original R_NATIONS array elements.
select n_name from region_view where r_name in ('EUROPE', 'ASIA');
+----------------+
```

```
| n_name         |
+----------------+
| UNITED KINGDOM |
| RUSSIA         |
| ROMANIA        |
| GERMANY        |
| FRANCE         |
| VIETNAM        |
| CHINA          |
| JAPAN          |
| INDONESIA      |
| INDIA          |
+----------------+

-- UNITED STATES in AMERICA and UNITED KINGDOM in EUROPE.
SELECT DISTINCT r_name FROM region_view WHERE n_name LIKE 'UNITED%';
+---------+
| r_name  |
+---------+
| AMERICA |
| EUROPE  |
+---------+

-- For conciseness, we only list some view columns in the SELECT list.
-- SELECT * would bring back all the data, unlike SELECT *
-- queries on the original table with complex type columns.
SELECT r_regionkey, r_name, n_nationkey, n_name FROM region_view LIMIT 7;
+-------------+--------+-------------+----------------+
| r_regionkey | r_name | n_nationkey | n_name         |
+-------------+--------+-------------+----------------+
| 3           | EUROPE | 23          | UNITED KINGDOM |
| 3           | EUROPE | 22          | RUSSIA         |
| 3           | EUROPE | 19          | ROMANIA        |
| 3           | EUROPE | 7           | GERMANY        |
| 3           | EUROPE | 6           | FRANCE         |
| 2           | ASIA   | 21          | VIETNAM        |
| 2           | ASIA   | 18          | CHINA          |
+-------------+--------+-------------+----------------+
```

Tutorials and Examples for Complex Types

The following examples illustrate the query syntax for some common use cases involving complex type columns.

Sample Schema and Data for Experimenting with Impala Complex Types

The tables used for earlier examples of complex type syntax are trivial ones with no actual data. The more substantial examples of the complex type feature use these tables, adapted from the schema used for TPC-H testing:

```
SHOW TABLES;
+----------+
| name     |
+----------+
| customer |
| part     |
| region   |
| supplier |
+----------+

DESCRIBE customer;
+--------------+-----------------------------------+
| name         | type                              |
+--------------+-----------------------------------+
| c_custkey    | bigint                            |
| c_name       | string                            |
| c_address    | string                            |
| c_nationkey  | smallint                          |
| c_phone      | string                            |
| c_acctbal    | decimal(12,2)                     |
| c_mktsegment | string                            |
| c_comment    | string                            |
| c_orders     | array<struct<                     |
|              |   o_orderkey:bigint,              |
```

```
|              |   o_orderstatus:string,                 |
|              |   o_totalprice:decimal(12,2),           |
|              |   o_orderdate:string,                   |
|              |   o_orderpriority:string,               |
|              |   o_clerk:string,                       |
|              |   o_shippriority:int,                   |
|              |   o_comment:string,                     |
|              |   o_lineitems:array<struct<             |
|              |     l_partkey:bigint,                   |
|              |     l_suppkey:bigint,                   |
|              |     l_linenumber:int,                   |
|              |     l_quantity:decimal(12,2),           |
|              |     l_extendedprice:decimal(12,2),      |
|              |     l_discount:decimal(12,2),           |
|              |     l_tax:decimal(12,2),                |
|              |     l_returnflag:string,                |
|              |     l_linestatus:string,                |
|              |     l_shipdate:string,                  |
|              |     l_commitdate:string,                |
|              |     l_receiptdate:string,               |
|              |     l_shipinstruct:string,              |
|              |     l_shipmode:string,                  |
|              |     l_comment:string                    |
|              |   >>                                    |
|              | >>                                      |
+--------------+-----------------------------------------+

DESCRIBE part;
+---------------+---------------+
| name          | type          |
+---------------+---------------+
| p_partkey     | bigint        |
| p_name        | string        |
| p_mfgr        | string        |
| p_brand       | string        |
| p_type        | string        |
| p_size        | int           |
| p_container   | string        |
| p_retailprice | decimal(12,2) |
| p_comment     | string        |
+---------------+---------------+

DESCRIBE region;
+-------------+-------------------------------------------------------------------------+
| name        | type                                                                    |
+-------------+-------------------------------------------------------------------------+
| r_regionkey | smallint                                                                |
| r_name      | string                                                                  |
| r_comment   | string                                                                  |
| r_nations   | array<struct<n_nationkey:smallint,n_name:string,n_comment:string>> |
+-------------+-------------------------------------------------------------------------+

DESCRIBE supplier;
+-------------+-----------------------------------------------+
| name        | type                                          |
+-------------+-----------------------------------------------+
| s_suppkey   | bigint                                        |
| s_name      | string                                        |
| s_address   | string                                        |
| s_nationkey | smallint                                      |
| s_phone     | string                                        |
| s_acctbal   | decimal(12,2)                                 |
| s_comment   | string                                        |
| s_partsupps | array<struct<ps_partkey:bigint,               |
|             | ps_availqty:int,ps_supplycost:decimal(12,2),  |
|             | ps_comment:string>>                           |
+-------------+-----------------------------------------------+
```

The volume of data used in the following examples is:

```
SELECT count(*) FROM customer;
+----------+
| count(*) |
+----------+
| 150000   |
+----------+

SELECT count(*) FROM part;
+----------+
| count(*) |
+----------+
| 200000   |
+----------+

SELECT count(*) FROM region;
+----------+
| count(*) |
+----------+
| 5        |
+----------+

SELECT count(*) FROM supplier;
+----------+
| count(*) |
+----------+
| 10000    |
+----------+
```

Constructing Parquet Files with Complex Columns Using Hive

The following examples demonstrate the Hive syntax to transform flat data (tables with all scalar columns) into Parquet tables where Impala can query the complex type columns. Each example shows the full sequence of steps, including switching back and forth between Impala and Hive. Although the source table can use any file format, the destination table must use the Parquet file format.

Create table with ARRAY in Impala, load data in Hive, query in Impala:

This example shows the cycle of creating the tables and querying the complex data in Impala, and using Hive (either the `hive` shell or `beeline`) for the data loading step. The data starts in flattened, denormalized form in a text table. Hive writes the corresponding Parquet data, including an ARRAY column. Then Impala can run analytic queries on the Parquet table, using join notation to unpack the ARRAY column.

```
/* Initial DDL and loading of flat, denormalized data happens in impala-shell */CREATE
 TABLE flat_array (country STRING, city STRING);INSERT INTO flat_array VALUES
    ('Canada', 'Toronto') , ('Canada', 'Vancouver') , ('Canada', "St. John\'s")
  , ('Canada', 'Saint John') , ('Canada', 'Montreal') , ('Canada', 'Halifax')
  , ('Canada', 'Winnipeg') , ('Canada', 'Calgary') , ('Canada', 'Saskatoon')
  , ('Canada', 'Ottawa') , ('Canada', 'Yellowknife') , ('France', 'Paris')
  , ('France', 'Nice') , ('France', 'Marseilles') , ('France', 'Cannes')
  , ('Greece', 'Athens') , ('Greece', 'Piraeus') , ('Greece', 'Hania')
  , ('Greece', 'Heraklion') , ('Greece', 'Rethymnon') , ('Greece', 'Fira');

CREATE TABLE complex_array (country STRING, city ARRAY <STRING>) STORED AS PARQUET;
```

```
/* Conversion to Parquet and complex and/or nested columns happens in Hive */

INSERT INTO complex_array SELECT country, collect_list(city) FROM flat_array GROUP BY
country;
Query ID = dev_20151108160808_84477ff2-82bd-4ba4-9a77-554fa7b8c0cb
Total jobs = 1
Launching Job 1 out of 1
```

```
...
```

```
/* Back to impala-shell again for analytic queries */

REFRESH complex_array;
SELECT country, city.item FROM complex_array, complex_array.city
+---------+-------------+
| country | item        |
+---------+-------------+
| Canada  | Toronto     |
| Canada  | Vancouver   |
| Canada  | St. John's  |
| Canada  | Saint John  |
| Canada  | Montreal    |
| Canada  | Halifax     |
| Canada  | Winnipeg    |
| Canada  | Calgary     |
| Canada  | Saskatoon   |
| Canada  | Ottawa      |
| Canada  | Yellowknife |
| France  | Paris       |
| France  | Nice        |
| France  | Marseilles  |
| France  | Cannes      |
| Greece  | Athens      |
| Greece  | Piraeus     |
| Greece  | Hania       |
| Greece  | Heraklion   |
| Greece  | Rethymnon   |
| Greece  | Fira        |
+---------+-------------+
```

Create table with STRUCT and ARRAY in Impala, load data in Hive, query in Impala:

This example shows the cycle of creating the tables and querying the complex data in Impala, and using Hive (either the `hive` shell or `beeline`) for the data loading step. The data starts in flattened, denormalized form in a text table. Hive writes the corresponding Parquet data, including a STRUCT column with an ARRAY field. Then Impala can run analytic queries on the Parquet table, using join notation to unpack the ARRAY field from the STRUCT column.

```
/* Initial DDL and loading of flat, denormalized data happens in impala-shell */

CREATE TABLE flat_struct_array (continent STRING, country STRING, city STRING);
INSERT INTO flat_struct_array VALUES
    ('North America', 'Canada', 'Toronto') , ('North America', 'Canada', 'Vancouver')
  , ('North America', 'Canada', "St. John\'s") , ('North America', 'Canada', 'Saint
John')
  , ('North America', 'Canada', 'Montreal') , ('North America', 'Canada', 'Halifax')
  , ('North America', 'Canada', 'Winnipeg') , ('North America', 'Canada', 'Calgary')
  , ('North America', 'Canada', 'Saskatoon') , ('North America', 'Canada', 'Ottawa')
  , ('North America', 'Canada', 'Yellowknife') , ('Europe', 'France', 'Paris')
  , ('Europe', 'France', 'Nice') , ('Europe', 'France', 'Marseilles')
  , ('Europe', 'France', 'Cannes') , ('Europe', 'Greece', 'Athens')
  , ('Europe', 'Greece', 'Piraeus') , ('Europe', 'Greece', 'Hania')
  , ('Europe', 'Greece', 'Heraklion') , ('Europe', 'Greece', 'Rethymnon')
  , ('Europe', 'Greece', 'Fira');

CREATE TABLE complex_struct_array (continent STRING, country STRUCT <name: STRING, city:
 ARRAY <STRING> >) STORED AS PARQUET;
```

```
/* Conversion to Parquet and complex and/or nested columns happens in Hive */

INSERT INTO complex_struct_array SELECT continent, named_struct('name', country, 'city',
 collect_list(city)) FROM flat_array_array GROUP BY continent, country;
Query ID = dev_20151108163535_11a4fa53-0003-4638-97e6-ef13cdb8e09e
Total jobs = 1
Launching Job 1 out of 1
```

```
...
```

```
/* Back to impala-shell again for analytic queries */

REFRESH complex_struct_array;
SELECT t1.continent, t1.country.name, t2.item
  FROM complex_struct_array t1, t1.country.city t2
+---------------+--------------+-------------+
| continent     | country.name | item        |
+---------------+--------------+-------------+
| Europe        | France       | Paris       |
| Europe        | France       | Nice        |
| Europe        | France       | Marseilles  |
| Europe        | France       | Cannes      |
| Europe        | Greece       | Athens      |
| Europe        | Greece       | Piraeus     |
| Europe        | Greece       | Hania       |
| Europe        | Greece       | Heraklion   |
| Europe        | Greece       | Rethymnon   |
| Europe        | Greece       | Fira        |
| North America | Canada       | Toronto     |
| North America | Canada       | Vancouver   |
| North America | Canada       | St. John's  |
| North America | Canada       | Saint John  |
| North America | Canada       | Montreal    |
| North America | Canada       | Halifax     |
| North America | Canada       | Winnipeg    |
| North America | Canada       | Calgary     |
| North America | Canada       | Saskatoon   |
| North America | Canada       | Ottawa      |
| North America | Canada       | Yellowknife |
+---------------+--------------+-------------+
```

Flattening Normalized Tables into a Single Table with Complex Types

One common use for complex types is to embed the contents of one table into another. The traditional technique of denormalizing results in a huge number of rows with some column values repeated over and over. With complex types, you can keep the same number of rows as in the original normalized table, and put all the associated data from the other table in a single new column.

In this flattening scenario, you might frequently use a column that is an ARRAY consisting of STRUCT elements, where each field within the STRUCT corresponds to a column name from the table that you are combining.

The following example shows a traditional normalized layout using two tables, and then an equivalent layout using complex types in a single table.

```
/* Traditional relational design */

-- This table just stores numbers, allowing us to look up details about the employee
-- and details about their vacation time using a three-table join query.
CREATE table employee_vacations
(
  employee_id BIGINT,
  vacation_id BIGINT
)
STORED AS PARQUET;

-- Each kind of information to track gets its own "fact table".
CREATE table vacation_details
(
  vacation_id BIGINT,
  vacation_start TIMESTAMP,
  duration INT
)
STORED AS PARQUET;

-- Any time we print a human-readable report, we join with this table to
-- display info about employee #1234.
```

```
CREATE TABLE employee_contact
(
  employee_id BIGINT,
  name STRING,
  address STRING,
  phone STRING,
  email STRING,
  address_type STRING /* 'home', 'work', 'remote', etc. */
)
STORED AS PARQUET;

/* Equivalent flattened schema using complex types */

-- For analytic queries using complex types, we can bundle the dimension table
-- and multiple fact tables into a single table.
CREATE TABLE employee_vacations_nested_types
(
-- We might still use the employee_id for other join queries.
-- The table needs at least one scalar column to serve as an identifier
-- for the complex type columns.
  employee_id BIGINT,

-- Columns of the VACATION_DETAILS table are folded into a STRUCT.
-- We drop the VACATION_ID column because Impala doesn't need
-- synthetic IDs to join a complex type column.
-- Each row from the VACATION_DETAILS table becomes an array element.
  vacation ARRAY < STRUCT <
    vacation_start: TIMESTAMP,
    duration: INT
  >>,

-- The ADDRESS_TYPE column, with a small number of predefined values that are distinct
-- for each employee, makes the EMPLOYEE_CONTACT table a good candidate to turn into a
 MAP,
-- with each row represented as a STRUCT. The string value from ADDRESS_TYPE becomes
the
-- "key" (the anonymous first field) of the MAP.
  contact MAP < STRING, STRUCT <
    address: STRING,
    phone: STRING,
    email: STRING
  >>
)
STORED AS PARQUET;
```

Interchanging Complex Type Tables and Data Files with Hive and Other Components

You can produce Parquet data files through several Hadoop components and APIs, as explained in http://www.cloudera.com/documentation/enterprise/latest/topics/cdh_ig_parquet.html.

If you have a Hive-created Parquet table that includes ARRAY, STRUCT, or MAP columns, Impala can query that same table in CDH 5.5 / Impala 2.3 and higher, subject to the usual restriction that all other columns are of data types supported by Impala, and also that the file type of the table must be Parquet.

If you have a Parquet data file produced outside of Impala, Impala can automatically deduce the appropriate table structure using the syntax `CREATE TABLE ... LIKE PARQUET 'hdfs_path_of_parquet_file'`. In CDH 5.5 / Impala 2.3 and higher, this feature works for Parquet files that include ARRAY, STRUCT, or MAP types.

```
/* In impala-shell, find the HDFS data directory of the original table.
DESCRIBE FORMATTED tpch_nested_parquet.customer;
...
| Location: | hdfs://localhost:20500/test-warehouse/tpch_nested_parquet.db/customer
| NULL |
...

# In the Unix shell, find the path of any Parquet data file in that HDFS directory.
$ hdfs dfs -ls hdfs://localhost:20500/test-warehouse/tpch_nested_parquet.db/customer
Found 4 items
-rwxr-xr-x   3 dev supergroup  171298918 2015-09-22 23:30
hdfs://localhost:20500/blah/tpch_nested_parquet.db/customer/000000_0
```

```
...

/* Back in impala-shell, use the HDFS path in a CREATE TABLE LIKE PARQUET statement. */
CREATE TABLE customer_ctlp
  LIKE PARQUET 'hdfs://localhost:20500/blah/tpch_nested_parquet.db/customer/000000_0'
  STORED AS PARQUET;

/* Confirm that old and new tables have the same column layout, including complex types.
 */
DESCRIBE tpch_nested_parquet.customer
+--------------+-----------------------------------+---------+
| name         | type                              | comment |
+--------------+-----------------------------------+---------+
| c_custkey    | bigint                            |         |
| c_name       | string                            |         |
| c_address    | string                            |         |
| c_nationkey  | smallint                          |         |
| c_phone      | string                            |         |
| c_acctbal    | decimal(12,2)                     |         |
| c_mktsegment | string                            |         |
| c_comment    | string                            |         |
| c_orders     | array<struct<                     |         |
|              |   o_orderkey:bigint,              |         |
|              |   o_orderstatus:string,           |         |
|              |   o_totalprice:decimal(12,2),     |         |
|              |   o_orderdate:string,             |         |
|              |   o_orderpriority:string,         |         |
|              |   o_clerk:string,                 |         |
|              |   o_shippriority:int,             |         |
|              |   o_comment:string,               |         |
|              |   o_lineitems:array<struct<       |         |
|              |     l_partkey:bigint,             |         |
|              |     l_suppkey:bigint,             |         |
|              |     l_linenumber:int,             |         |
|              |     l_quantity:decimal(12,2),     |         |
|              |     l_extendedprice:decimal(12,2), |         |
|              |     l_discount:decimal(12,2),     |         |
|              |     l_tax:decimal(12,2),          |         |
|              |     l_returnflag:string,          |         |
|              |     l_linestatus:string,          |         |
|              |     l_shipdate:string,            |         |
|              |     l_commitdate:string,          |         |
|              |     l_receiptdate:string,         |         |
|              |     l_shipinstruct:string,        |         |
|              |     l_shipmode:string,            |         |
|              |     l_comment:string              |         |
|              |   >>                              |         |
|              | >>                                |         |
+--------------+-----------------------------------+---------+

describe customer_ctlp;
+--------------+-----------------------------------+------------------------------+
| name         | type                              | comment                      |
+--------------+-----------------------------------+------------------------------+
| c_custkey    | bigint                            | Inferred from Parquet file. |
| c_name       | string                            | Inferred from Parquet file. |
| c_address    | string                            | Inferred from Parquet file. |
| c_nationkey  | int                               | Inferred from Parquet file. |
| c_phone      | string                            | Inferred from Parquet file. |
| c_acctbal    | decimal(12,2)                     | Inferred from Parquet file. |
| c_mktsegment | string                            | Inferred from Parquet file. |
| c_comment    | string                            | Inferred from Parquet file. |
| c_orders     | array<struct<                     | Inferred from Parquet file. |
|              |   o_orderkey:bigint,              |                              |
|              |   o_orderstatus:string,           |                              |
|              |   o_totalprice:decimal(12,2),     |                              |
|              |   o_orderdate:string,             |                              |
|              |   o_orderpriority:string,         |                              |
|              |   o_clerk:string,                 |                              |
|              |   o_shippriority:int,             |                              |
|              |   o_comment:string,               |                              |
|              |   o_lineitems:array<struct<       |                              |
|              |     l_partkey:bigint,             |                              |
```

```
|              |     l_suppkey:bigint,               |                               |
|              |     l_linenumber:int,               |                               |
|              |     l_quantity:decimal(12,2),       |                               |
|              |     l_extendedprice:decimal(12,2),  |                               |
|              |     l_discount:decimal(12,2),       |                               |
|              |     l_tax:decimal(12,2),            |                               |
|              |     l_returnflag:string,            |                               |
|              |     l_linestatus:string,            |                               |
|              |     l_shipdate:string,              |                               |
|              |     l_commitdate:string,            |                               |
|              |     l_receiptdate:string,           |                               |
|              |     l_shipinstruct:string,          |                               |
|              |     l_shipmode:string,              |                               |
|              |     l_comment:string                |                               |
|              |   >>                                |                               |
|              | >>                                  |                               |
+--------------+-------------------------------------+-------------------------------+
```

Literals

Each of the Impala data types has corresponding notation for literal values of that type. You specify literal values in SQL statements, such as in the SELECT list or WHERE clause of a query, or as an argument to a function call. See Data Types on page 101 for a complete list of types, ranges, and conversion rules.

Numeric Literals

To write literals for the integer types (TINYINT, SMALLINT, INT, and BIGINT), use a sequence of digits with optional leading zeros.

To write literals for the floating-point types (DECIMAL, FLOAT, and DOUBLE), use a sequence of digits with an optional decimal point (. character). To preserve accuracy during arithmetic expressions, Impala interprets floating-point literals as the DECIMAL type with the smallest appropriate precision and scale, until required by the context to convert the result to FLOAT or DOUBLE.

Integer values are promoted to floating-point when necessary, based on the context.

You can also use exponential notation by including an e character. For example, 1e6 is 1 times 10 to the power of 6 (1 million). A number in exponential notation is always interpreted as floating-point.

When Impala encounters a numeric literal, it considers the type to be the “smallest” that can accurately represent the value. The type is promoted to larger or more accurate types if necessary, based on subsequent parts of an expression.

For example, you can see by the types Impala defines for the following table columns how it interprets the corresponding numeric literals:

```
[localhost:21000] > create table ten as select 10 as x;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc ten;
+------+---------+---------+
| name | type    | comment |
+------+---------+---------+
| x    | tinyint |         |
+------+---------+---------+

[localhost:21000] > create table four_k as select 4096 as x;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc four_k;
```

```
+------+----------+---------+
| name | type     | comment |
+------+----------+---------+
| x    | smallint |         |
+------+----------+---------+

[localhost:21000] > create table one_point_five as select 1.5 as x;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc one_point_five;
+------+--------------+---------+
| name | type         | comment |
+------+--------------+---------+
| x    | decimal(2,1) |         |
+------+--------------+---------+

[localhost:21000] > create table one_point_three_three_three as select 1.333 as x;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc one_point_three_three_three;
+------+--------------+---------+
| name | type         | comment |
+------+--------------+---------+
| x    | decimal(4,3) |         |
+------+--------------+---------+
```

String Literals

String literals are quoted using either single or double quotation marks. You can use either kind of quotes for string literals, even both kinds for different literals within the same statement.

Quoted literals are considered to be of type STRING. To use quoted literals in contexts requiring a CHAR or VARCHAR value, CAST( ) the literal to a CHAR or VARCHAR of the appropriate length.

Escaping special characters:

To encode special characters within a string literal, precede them with the backslash (\) escape character:

- \t represents a tab.
- \n represents a newline or linefeed. This might cause extra line breaks in impala-shell output.
- \r represents a carriage return. This might cause unusual formatting (making it appear that some content is overwritten) in impala-shell output.
- \b represents a backspace. This might cause unusual formatting (making it appear that some content is overwritten) in impala-shell output.
- \0 represents an ASCII nul character (not the same as a SQL NULL). This might not be visible in impala-shell output.
- \Z represents a DOS end-of-file character. This might not be visible in impala-shell output.
- \% and _ can be used to escape wildcard characters within the string passed to the LIKE operator.
- \ followed by 3 octal digits represents the ASCII code of a single character; for example, \101 is ASCII 65, the character A.
- Use two consecutive backslashes (\\) to prevent the backslash from being interpreted as an escape character.
- Use the backslash to escape single or double quotation mark characters within a string literal, if the literal is enclosed by the same type of quotation mark.
- If the character following the \ does not represent the start of a recognized escape sequence, the character is passed through unchanged.

Quotes within quotes:

To include a single quotation character within a string value, enclose the literal with either single or double quotation marks, and optionally escape the single quote as a `\'` sequence. Earlier releases required escaping a single quote inside double quotes. Continue using escape sequences in this case if you also need to run your SQL code on older versions of Impala.

To include a double quotation character within a string value, enclose the literal with single quotation marks, no escaping is necessary in this case. Or, enclose the literal with double quotation marks and escape the double quote as a `\"` sequence.

```
[localhost:21000] > select "What\'s happening?" as single_within_double,
                  >        'I\'m not sure.' as single_within_single,
                  >        "Homer wrote \"The Iliad\"." as double_within_double,
                  >        'Homer also wrote "The Odyssey".' as double_within_single;
+--------------------+--------------------+------------------------+------------------------------+
| single_within_double | single_within_single | double_within_double     |
double_within_single            |
+--------------------+--------------------+------------------------+------------------------------+
| What's happening?    | I'm not sure.        | Homer wrote "The Iliad". | Homer also
wrote "The Odyssey". |
+--------------------+--------------------+------------------------+------------------------------+
```

Field terminator character in CREATE TABLE:

Note: The `CREATE TABLE` clauses `FIELDS TERMINATED BY`, `ESCAPED BY`, and `LINES TERMINATED BY` have special rules for the string literal used for their argument, because they all require a single character. You can use a regular character surrounded by single or double quotation marks, an octal sequence such as `'\054'` (representing a comma), or an integer in the range '-127'..'128' (with quotation marks but no backslash), which is interpreted as a single-byte ASCII character. Negative values are subtracted from 256; for example, `FIELDS TERMINATED BY '-2'` sets the field delimiter to ASCII code 254, the "Icelandic Thorn" character used as a delimiter by some data formats.

impala-shell considerations:

When dealing with output that includes non-ASCII or non-printable characters such as linefeeds and backspaces, use the `impala-shell` options to save to a file, turn off pretty printing, or both rather than relying on how the output appears visually. See impala-shell Configuration Options on page 714 for a list of `impala-shell` options.

Boolean Literals

For `BOOLEAN` values, the literals are `TRUE` and `FALSE`, with no quotation marks and case-insensitive.

Examples:

```
select true;
select * from t1 where assertion = false;
select case bool_col when true then 'yes' when false 'no' else 'null' end from t1;
```

Timestamp Literals

Impala automatically converts `STRING` literals of the correct format into `TIMESTAMP` values. Timestamp values are accepted in the format `'yyyy-MM-dd HH:mm:ss.SSSSSS'`, and can consist of just the date, or just the time, with or without the fractional second portion. For example, you can specify `TIMESTAMP` values such as `'1966-07-30'`, `'08:30:00'`, or `'1985-09-25 17:45:30.005'`.

Leading zeroes are not required in the numbers representing the date component, such as month and date, or the time component, such as hour, minute, and second. For example, Impala accepts both `'2018-1-1 01:02:03'` and `'2018-01-01 1:2:3'` as valid.

In `STRING` to `TIMESTAMP` conversions, leading and trailing white spaces, such as a space, a tab, a newline, or a carriage return, are ignored. For example, Impala treats the following as equivalent: '1999-12-01 01:02:03 ', ' 1999-12-01 01:02:03', '1999-12-01 01:02:03\r\n\t'.

When you convert or cast a STRING literal to TIMESTAMP, you can use the following separators between the date part and the time part:

- One or more space characters

  Example: CAST('2001-01-09 01:05:01' AS TIMESTAMP)

- The character "T"

  Example: CAST('2001-01-09T01:05:01' AS TIMESTAMP)

You can also use INTERVAL expressions to add or subtract from timestamp literal values, such as CAST('1966-07-30' AS TIMESTAMP) + INTERVAL 5 YEARS + INTERVAL 3 DAYS. See TIMESTAMP Data Type on page 130 for details.

Depending on your data pipeline, you might receive date and time data as text, in notation that does not exactly match the format for Impala TIMESTAMP literals. See Impala Date and Time Functions on page 424 for functions that can convert between a variety of string literals (including different field order, separators, and timezone notation) and equivalent TIMESTAMP or numeric values.

NULL

The notion of NULL values is familiar from all kinds of database systems, but each SQL dialect can have its own behavior and restrictions on NULL values. For Big Data processing, the precise semantics of NULL values are significant: any misunderstanding could lead to inaccurate results or misformatted data, that could be time-consuming to correct for large data sets.

- NULL is a different value than an empty string. The empty string is represented by a string literal with nothing inside, "" or ''.
- In a delimited text file, the NULL value is represented by the special token \N.
- When Impala inserts data into a partitioned table, and the value of one of the partitioning columns is NULL or the empty string, the data is placed in a special partition that holds only these two kinds of values. When these values are returned in a query, the result is NULL whether the value was originally NULL or an empty string. This behavior is compatible with the way Hive treats NULL values in partitioned tables. Hive does not allow empty strings as partition keys, and it returns a string value such as __HIVE_DEFAULT_PARTITION__ instead of NULL when such values are returned from a query. For example:

```
create table t1 (i int) partitioned by (x int, y string);
-- Select an INT column from another table, with all rows going into a special HDFS
subdirectory
-- named __HIVE_DEFAULT_PARTITION__. Depending on whether one or both of the partitioning
 keys
-- are null, this special directory name occurs at different levels of the physical data
 directory
-- for the table.
insert into t1 partition(x=NULL, y=NULL) select c1 from some_other_table;
insert into t1 partition(x, y=NULL) select c1, c2 from some_other_table;
insert into t1 partition(x=NULL, y) select c1, c3  from some_other_table;
```

- There is no NOT NULL clause when defining a column to prevent NULL values in that column.
- There is no DEFAULT clause to specify a non-NULL default value.
- If an INSERT operation mentions some columns but not others, the unmentioned columns contain NULL for all inserted rows.
- In Impala 1.2.1 and higher, all NULL values come at the end of the result set for ORDER BY ... ASC queries, and at the beginning of the result set for ORDER BY ... DESC queries. In effect, NULL is considered greater than all other values for sorting purposes. The original Impala behavior always put NULL values at the end, even for ORDER BY ... DESC queries. The new behavior in Impala 1.2.1 makes Impala more compatible with other popular database systems. In Impala 1.2.1 and higher, you can override or specify the sorting behavior for NULL by adding the clause NULLS FIRST or NULLS LAST at the end of the ORDER BY clause.



Note: Because the NULLS FIRST and NULLS LAST keywords are not currently available in Hive queries, any views you create using those keywords will not be available through Hive.

- In all other contexts besides sorting with ORDER BY, comparing a NULL to anything else returns NULL, making the comparison meaningless. For example, 10 > NULL produces NULL, 10 < NULL also produces NULL, 5 BETWEEN 1 AND NULL produces NULL, and so on.

Several built-in functions serve as shorthand for evaluating expressions and returning NULL, 0, or some other substitution value depending on the expression result: ifnull(), isnull(), nvl(), nullif(), nullifzero(), and zeroifnull(). See Impala Conditional Functions on page 457 for details.

Kudu considerations:

Columns in Kudu tables have an attribute that specifies whether or not they can contain NULL values. A column with a NULL attribute can contain nulls. A column with a NOT NULL attribute cannot contain any nulls, and an INSERT, UPDATE, or UPSERT statement will skip any row that attempts to store a null in a column designated as NOT NULL. Kudu tables default to the NULL setting for each column, except columns that are part of the primary key.

In addition to columns with the NOT NULL attribute, Kudu tables also have restrictions on NULL values in columns that are part of the primary key for a table. No column that is part of the primary key in a Kudu table can contain any NULL values.

SQL Operators

SQL operators are a class of comparison functions that are widely used within the WHERE clauses of SELECT statements.

Arithmetic Operators

The arithmetic operators use expressions with a left-hand argument, the operator, and then (in most cases) a right-hand argument.

Syntax:

```
left_hand_arg binary_operator right_hand_arg
unary_operator single_arg
```

- + and -: Can be used either as unary or binary operators.
  - With unary notation, such as +5, -2.5, or -*col_name*, they multiply their single numeric argument by +1 or -1. Therefore, unary + returns its argument unchanged, while unary - flips the sign of its argument. Although you can double up these operators in expressions such as ++5 (always positive) or -+2 or +-2 (both always negative), you cannot double the unary minus operator because -- is interpreted as the start of a comment. (You can use a double unary minus operator if you separate the - characters, for example with a space or parentheses.)
  - With binary notation, such as 2+2, 5-2.5, or *col1* + *col2*, they add or subtract respectively the right-hand argument to (or from) the left-hand argument. Both arguments must be of numeric types.

- * and /: Multiplication and division respectively. Both arguments must be of numeric types.

  When multiplying, the shorter argument is promoted if necessary (such as SMALLINT to INT or BIGINT, or FLOAT to DOUBLE), and then the result is promoted again to the next larger type. Thus, multiplying a TINYINT and an INT produces a BIGINT result. Multiplying a FLOAT and a FLOAT produces a DOUBLE result. Multiplying a FLOAT and a DOUBLE or a DOUBLE and a DOUBLE produces a DECIMAL(38,17), because DECIMAL values can represent much larger and more precise values than DOUBLE.

  When dividing, Impala always treats the arguments and result as DOUBLE values to avoid losing precision. If you need to insert the results of a division operation into a FLOAT column, use the CAST() function to convert the result to the correct type.

- DIV: Integer division. Arguments are not promoted to a floating-point type, and any fractional result is discarded. For example, 13 DIV 7 returns 1, 14 DIV 7 returns 2, and 15 DIV 7 returns 2. This operator is the same as the QUOTIENT( ) function.
- %: Modulo operator. Returns the remainder of the left-hand argument divided by the right-hand argument. Both arguments must be of one of the integer types.
- &, |, ~, and ^: Bitwise operators that return the logical AND, logical OR, NOT, or logical XOR (exclusive OR) of their argument values. Both arguments must be of one of the integer types. If the arguments are of different type, the argument with the smaller type is implicitly extended to match the argument with the longer type.

You can chain a sequence of arithmetic expressions, optionally grouping them with parentheses.

The arithmetic operators generally do not have equivalent calling conventions using functional notation. For example, prior to CDH 5.4 / Impala 2.2, there is no MOD( ) function equivalent to the % modulo operator. Conversely, there are some arithmetic functions that do not have a corresponding operator. For example, for exponentiation you use the POW( ) function, but there is no ** exponentiation operator. See Impala Mathematical Functions on page 397 for the arithmetic functions you can use.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM( ) and AVG( ) are computed using the numeric R_NATIONKEY field, and the general-purpose MAX( ) and MIN( ) values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
```

```
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+---------------+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+------+-----------+----------------+---------------+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5             |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5             |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5             |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5             |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5             |
+-------------+-------+-----+------+-----------+----------------+---------------+
```

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in an arithmetic expression, such as multiplying by 10:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey * 10
  from region, region.r_nations as nation
where nation.item.n_nationkey < 5;
+-------------+-------------+-----------------------------+
| r_name      | item.n_name | nation.item.n_nationkey * 10 |
+-------------+-------------+-----------------------------+
| AMERICA     | CANADA      | 30                          |
| AMERICA     | BRAZIL      | 20                          |
| AMERICA     | ARGENTINA   | 10                          |
```

```
| MIDDLE EAST | EGYPT       | 40                            |
| AFRICA      | ALGERIA     | 0                             |
+-------------+-------------+-------------------------------+
```

BETWEEN Operator

In a WHERE clause, compares an expression to both a lower and upper bound. The comparison is successful is the expression is greater than or equal to the lower bound, and less than or equal to the upper bound. If the bound values are switched, so the lower bound is greater than the upper bound, does not match any values.

Syntax:

```
expression BETWEEN lower_bound AND upper_bound
```

Data types: Typically used with numeric data types. Works with any data type, although not very practical for BOOLEAN values. (BETWEEN false AND true will match all BOOLEAN values.) Use CAST( ) if necessary to ensure the lower and upper bound values are compatible types. Call string or date/time functions if necessary to extract or transform the relevant portion to compare, especially if the value can be transformed into a number.

Usage notes:

Be careful when using short string operands. A longer string that starts with the upper bound value will not be included, because it is considered greater than the upper bound. For example, BETWEEN 'A' and 'M' would not match the string value 'Midway'. Use functions such as upper( ), lower( ), substr( ), trim( ), and so on if necessary to ensure the comparison works as expected.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

```
-- Retrieve data for January through June, inclusive.
select c1 from t1 where month between 1 and 6;

-- Retrieve data for names beginning with 'A' through 'M' inclusive.
-- Only test the first letter to ensure all the values starting with 'M' are matched.
-- Do a case-insensitive comparison to match names with various capitalization
conventions.
select last_name from customers where upper(substr(last_name,1,1)) between 'A' and 'M';

-- Retrieve data for only the first week of each month.
select count(distinct visitor_id)) from web_traffic where dayofmonth(when_viewed) between
 1 and 7;
```

The following example shows how to do a BETWEEN comparison using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in a comparison operator:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
```

```
|             | >>                        |         |
+-------------+---------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
from region, region.r_nations as nation
where nation.item.n_nationkey between 3 and 5
+-------------+-------------+------------------+
| r_name      | item.n_name | item.n_nationkey |
+-------------+-------------+------------------+
| AMERICA     | CANADA      | 3                |
| MIDDLE EAST | EGYPT       | 4                |
| AFRICA      | ETHIOPIA    | 5                |
+-------------+-------------+------------------+
```

Comparison Operators

Impala supports the familiar comparison operators for checking equality and sort order for the column data types:

Syntax:

```
left_hand_expression comparison_operator right_hand_expression
```

- =, !=, <>: apply to all scalar types.
- <, <=, >, >=: apply to all scalar types; for BOOLEAN, TRUE is considered greater than FALSE.

Alternatives:

The IN and BETWEEN operators provide shorthand notation for expressing combinations of equality, less than, and greater than comparisons with a single operator.

Because comparing any value to NULL produces NULL rather than TRUE or FALSE, use the IS NULL and IS NOT NULL operators to check if a value is NULL or not.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used with a comparison operator such as <:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+--------------------------+---------+
| name        | type                     | comment |
+-------------+--------------------------+---------+
| r_regionkey | smallint                 |         |
| r_name      | string                   |         |
| r_comment   | string                   |         |
| r_nations   | array<struct<            |         |
|             |   n_nationkey:smallint,  |         |
|             |   n_name:string,         |         |
|             |   n_comment:string       |         |
|             | >>                       |         |
+-------------+--------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
from region, region.r_nations as nation
```

```
where nation.item.n_nationkey < 5
+-------------+-------------+------------------+
| r_name      | item.n_name | item.n_nationkey |
+-------------+-------------+------------------+
| AMERICA     | CANADA      | 3                |
| AMERICA     | BRAZIL      | 2                |
| AMERICA     | ARGENTINA   | 1                |
| MIDDLE EAST | EGYPT       | 4                |
| AFRICA      | ALGERIA     | 0                |
+-------------+-------------+------------------+
```

EXISTS Operator

The EXISTS operator tests whether a subquery returns any results. You typically use it to find values from one table that have corresponding values in another table.

The converse, NOT EXISTS, helps to find all the values from one table that do not have any corresponding values in another table.

Syntax:

```
EXISTS (subquery)
NOT EXISTS (subquery)
```

Usage notes:

The subquery can refer to a different table than the outer query block, or the same table. For example, you might use EXISTS or NOT EXISTS to check the existence of parent/child relationships between two columns of the same table.

You can also use operators and function calls within the subquery to test for other kinds of relationships other than strict equality. For example, you might use a call to COUNT( ) in the subquery to check whether the number of matching values is higher or lower than some limit. You might call a UDF in the subquery to check whether values in one table matches a hashed representation of those same values in a different table.

NULL considerations:

If the subquery returns any value at all (even NULL), EXISTS returns TRUE and NOT EXISTS returns false.

The following example shows how even when the subquery returns only NULL values, EXISTS still returns TRUE and thus matches all the rows from the table in the outer query block.

```
[localhost:21000] > create table all_nulls (x int);
[localhost:21000] > insert into all_nulls values (null), (null), (null);
[localhost:21000] > select y from t2 where exists (select x from all_nulls);
+---+
| y |
+---+
| 2 |
| 4 |
| 6 |
+---+
```

However, if the table in the subquery is empty and so the subquery returns an empty result set, EXISTS returns FALSE:

```
[localhost:21000] > create table empty (x int);
[localhost:21000] > select y from t2 where exists (select x from empty);
[localhost:21000] >
```

Added in: CDH 5.2.0 / Impala 2.0.0

Restrictions:

Correlated subqueries used in EXISTS and IN operators cannot include a LIMIT clause.

Prior to CDH 5.8 / Impala 2.6, the NOT EXISTS operator required a correlated subquery. In CDH 5.8 / Impala 2.6 and higher, NOT EXISTS works with uncorrelated queries also.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

The following examples refer to these simple tables containing small sets of integers or strings:

```
[localhost:21000] > create table t1 (x int);
[localhost:21000] > insert into t1 values (1), (2), (3), (4), (5), (6);

[localhost:21000] > create table t2 (y int);
[localhost:21000] > insert into t2 values (2), (4), (6);

[localhost:21000] > create table t3 (z int);
[localhost:21000] > insert into t3 values (1), (3), (5);

[localhost:21000] > create table month_names (m string);
[localhost:21000] > insert into month_names values
                  > ('January'), ('February'), ('March'),
                  > ('April'), ('May'), ('June'), ('July'),
                  > ('August'), ('September'), ('October'),
                  > ('November'), ('December');
```

The following example shows a correlated subquery that finds all the values in one table that exist in another table. For each value X from T1, the query checks if the Y column of T2 contains an identical value, and the EXISTS operator returns TRUE or FALSE as appropriate in each case.

```
localhost:21000] > select x from t1 where exists (select y from t2 where t1.x = y);
+---+
| x |
+---+
| 2 |
| 4 |
| 6 |
+---+
```

An uncorrelated query is less interesting in this case. Because the subquery always returns TRUE, all rows from T1 are returned. If the table contents where changed so that the subquery did not match any rows, none of the rows from T1 would be returned.

```
[localhost:21000] > select x from t1 where exists (select y from t2 where y > 5);
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
+---+
```

The following example shows how an uncorrelated subquery can test for the existence of some condition within a table. By using LIMIT 1 or an aggregate function, the query returns a single result or no result based on whether the subquery matches any rows. Here, we know that T1 and T2 contain some even numbers, but T3 does not.

```
[localhost:21000] > select "contains an even number" from t1 where exists (select x from
 t1 where x % 2 = 0) limit 1;
+--------------------------+
| 'contains an even number' |
+--------------------------+
| contains an even number   |
+--------------------------+
```

```
[localhost:21000] > select "contains an even number" as assertion from t1 where exists
 (select x from t1 where x % 2 = 0) limit 1;
+------------------------+
| assertion              |
+------------------------+
| contains an even number |
+------------------------+
[localhost:21000] > select "contains an even number" as assertion from t2 where exists
 (select x from t2 where y % 2 = 0) limit 1;
ERROR: AnalysisException: couldn't resolve column reference: 'x'
[localhost:21000] > select "contains an even number" as assertion from t2 where exists
 (select y from t2 where y % 2 = 0) limit 1;
+------------------------+
| assertion              |
+------------------------+
| contains an even number |
+------------------------+
[localhost:21000] > select "contains an even number" as assertion from t3 where exists
 (select z from t3 where z % 2 = 0) limit 1;
[localhost:21000] >
```

The following example finds numbers in one table that are 1 greater than numbers from another table. The `EXISTS` notation is simpler than an equivalent `CROSS JOIN` between the tables. (The example then also illustrates how the same test could be performed using an `IN` operator.)

```
[localhost:21000] > select x from t1 where exists (select y from t2 where x = y + 1);
+---+
| x |
+---+
| 3 |
| 5 |
+---+
[localhost:21000] > select x from t1 where x in (select y + 1 from t2);
+---+
| x |
+---+
| 3 |
| 5 |
+---+
```

The following example finds values from one table that do not exist in another table.

```
[localhost:21000] > select x from t1 where not exists (select y from t2 where x = y);
+---+
| x |
+---+
| 1 |
| 3 |
| 5 |
+---+
```

The following example uses the `NOT EXISTS` operator to find all the leaf nodes in tree-structured data. This simplified "tree of life" has multiple levels (class, order, family, and so on), with each item pointing upward through a `PARENT` pointer. The example runs an outer query and a subquery on the same table, returning only those items whose `ID` value is *not* referenced by the `PARENT` of any other item.

```
[localhost:21000] > create table tree (id int, parent int, name string);
[localhost:21000] > insert overwrite tree values
                  > (0, null, "animals"),
                  > (1, 0, "placentals"),
                  > (2, 0, "marsupials"),
                  > (3, 1, "bats"),
                  > (4, 1, "cats"),
                  > (5, 2, "kangaroos"),
                  > (6, 4, "lions"),
                  > (7, 4, "tigers"),
                  > (8, 5, "red kangaroo"),
                  > (9, 2, "wallabies");
```

```
[localhost:21000] > select name as "leaf node" from tree one
                  > where not exists (select parent from tree two where one.id =
two.parent);
+--------------+
| leaf node    |
+--------------+
| bats         |
| lions        |
| tigers       |
| red kangaroo |
| wallabies    |
+--------------+
```

Related information:

Subqueries in Impala SELECT Statements on page 312

ILIKE Operator

A case-insensitive comparison operator for STRING data, with basic wildcard capability using _ to match a single character and % to match multiple characters. The argument expression must match the entire string value. Typically, it is more efficient to put any % wildcard match at the end of the string.

This operator, available in CDH 5.7 / Impala 2.5 and higher, is the equivalent of the LIKE operator, but with case-insensitive comparisons.

Syntax:

```
string_expression ILIKE wildcard_expression
string_expression NOT ILIKE wildcard_expression
```

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

In the following examples, strings that are the same except for differences in uppercase and lowercase match successfully with ILIKE, but do not match with LIKE:

```
select 'fooBar' ilike 'FOOBAR';
+-------------------------+
| 'foobar' ilike 'foobar' |
+-------------------------+
| true                    |
+-------------------------+

select 'fooBar' like 'FOOBAR';
+------------------------+
| 'foobar' like 'foobar' |
+------------------------+
| false                  |
+------------------------+

select 'FOOBAR' ilike 'f%';
+---------------------+
| 'foobar' ilike 'f%' |
+---------------------+
| true                |
+---------------------+

select 'FOOBAR' like 'f%';
+--------------------+
| 'foobar' like 'f%' |
+--------------------+
```

```
| false              |
+--------------------+

select 'ABCXYZ' not ilike 'ab_xyz';
+----------------------------+
| not 'abcxyz' ilike 'ab_xyz' |
+----------------------------+
| false                      |
+----------------------------+

select 'ABCXYZ' not like 'ab_xyz';
+---------------------------+
| not 'abcxyz' like 'ab_xyz' |
+---------------------------+
| true                      |
+---------------------------+
```

Related information:

For case-sensitive comparisons, see LIKE Operator on page 186. For a more general kind of search operator using regular expressions, see REGEXP Operator on page 189 or its case-insensitive counterpart IREGEXP Operator on page 182.

IN Operator

The `IN` operator compares an argument value to a set of values, and returns `TRUE` if the argument matches any value in the set. The `NOT IN` operator reverses the comparison, and checks if the argument value is not part of a set of values.

Syntax:

```
expression IN (expression [, expression])
expression IN (subquery)

expression NOT IN (expression [, expression])
expression NOT IN (subquery)
```

The left-hand expression and the set of comparison values must be of compatible types.

The left-hand expression must consist only of a single value, not a tuple. Although the left-hand expression is typically a column name, it could also be some other value. For example, the `WHERE` clauses `WHERE id IN (5)` and `WHERE 5 IN (id)` produce the same results.

The set of values to check against can be specified as constants, function calls, column names, or other expressions in the query text. The maximum number of expressions in the `IN` list is 9999. (The maximum number of elements of a single expression is 10,000 items, and the `IN` operator itself counts as one.)

In Impala 2.0 and higher, the set of values can also be generated by a subquery. `IN` can evaluate an unlimited number of results using a subquery.

Usage notes:

Any expression using the `IN` operator could be rewritten as a series of equality tests connected with `OR`, but the `IN` syntax is often clearer, more concise, and easier for Impala to optimize. For example, with partitioned tables, queries frequently use `IN` clauses to filter data by comparing the partition key columns to specific values.

NULL considerations:

If there really is a matching non-null value, `IN` returns `TRUE`:

```
[localhost:21000] > select 1 in (1,null,2,3);
+----------------------+
| 1 in (1, null, 2, 3) |
+----------------------+
| true                 |
+----------------------+
[localhost:21000] > select 1 not in (1,null,2,3);
+--------------------------+
```

```
| 1 not in (1, null, 2, 3) |
+--------------------------+
| false                    |
+--------------------------+
```

If the searched value is not found in the comparison values, and the comparison values include `NULL`, the result is `NULL`:

```
[localhost:21000] > select 5 in (1,null,2,3);
+----------------------+
| 5 in (1, null, 2, 3) |
+----------------------+
| NULL                 |
+----------------------+
[localhost:21000] > select 5 not in (1,null,2,3);
+--------------------------+
| 5 not in (1, null, 2, 3) |
+--------------------------+
| NULL                     |
+--------------------------+
[localhost:21000] > select 1 in (null);
+-------------+
| 1 in (null) |
+-------------+
| NULL        |
+-------------+
[localhost:21000] > select 1 not in (null);
+-----------------+
| 1 not in (null) |
+-----------------+
| NULL            |
+-----------------+
```

If the left-hand argument is `NULL`, `IN` always returns `NULL`. This rule applies even if the comparison values include `NULL`.

```
[localhost:21000] > select null in (1,2,3);
+-------------------+
| null in (1, 2, 3) |
+-------------------+
| NULL              |
+-------------------+
[localhost:21000] > select null not in (1,2,3);
+-----------------------+
| null not in (1, 2, 3) |
+-----------------------+
| NULL                  |
+-----------------------+
[localhost:21000] > select null in (null);
+----------------+
| null in (null) |
+----------------+
| NULL           |
+----------------+
[localhost:21000] > select null not in (null);
+--------------------+
| null not in (null) |
+--------------------+
| NULL               |
+--------------------+
```

Added in: Available in earlier Impala releases, but new capabilities were added in CDH 5.2.0 / Impala 2.0.0

Complex type considerations:

You cannot refer to a column with a complex data type (`ARRAY`, `STRUCT`, or `MAP`) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a `STRUCT`, the items of an `ARRAY`, or the key or value portion of a `MAP`) as part of a join query that refers to the scalar value using the appropriate dot notation or `ITEM`, `KEY`, or `VALUE` pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in an arithmetic expression, such as multiplying by 10:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
from region, region.r_nations as nation
where nation.item.n_nationkey in (1,3,5)
+---------+-------------+------------------+
| r_name  | item.n_name | item.n_nationkey |
+---------+-------------+------------------+
| AMERICA | CANADA      | 3                |
| AMERICA | ARGENTINA   | 1                |
| AFRICA  | ETHIOPIA    | 5                |
+---------+-------------+------------------+
```

Restrictions:

Correlated subqueries used in EXISTS and IN operators cannot include a LIMIT clause.

Examples:

```
-- Using IN is concise and self-documenting.
SELECT * FROM t1 WHERE c1 IN (1,2,10);
-- Equivalent to series of = comparisons ORed together.
SELECT * FROM t1 WHERE c1 = 1 OR c1 = 2 OR c1 = 10;

SELECT c1 AS "starts with vowel" FROM t2 WHERE upper(substr(c1,1,1)) IN
('A','E','I','O','U');

SELECT COUNT(DISTINCT(visitor_id)) FROM web_traffic WHERE month IN
('January','June','July');
```

Related information:

Subqueries in Impala SELECT Statements on page 312

IREGEXP Operator

Tests whether a value matches a regular expression, using case-insensitive string comparisons. Uses the POSIX regular expression syntax where ^ and $ match the beginning and end of the string, . represents any single character, * represents a sequence of zero or more items, + represents a sequence of one or more items, ? produces a non-greedy match, and so on.

This operator, available in CDH 5.7 / Impala 2.5 and higher, is the equivalent of the REGEXP operator, but with case-insensitive comparisons.

Syntax:

```
string_expression IREGEXP regular_expression
```

Usage notes:

The | symbol is the alternation operator, typically used within ( ) to match different sequences. The ( ) groups do not allow backreferences. To retrieve the part of a value matched within a ( ) section, use the `regexp_extract()` built-in function. (Currently, there is not any case-insensitive equivalent for the `regexp_extract()` function.)

In Impala 1.3.1 and higher, the REGEXP and RLIKE operators now match a regular expression string that occurs anywhere inside the target string, the same as if the regular expression was enclosed on each side by `.*`. See REGEXP Operator on page 189 for examples. Previously, these operators only succeeded when the regular expression matched the entire target string. This change improves compatibility with the regular expression support for popular database systems. There is no change to the behavior of the `regexp_extract()` and `regexp_replace()` built-in functions.

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including `.*?` for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

The following examples demonstrate the syntax for the IREGEXP operator.

```
select 'abcABCaabbcc' iregexp '^[a-c]+$';
+---------------------------------+
| 'abcabcaabbcc' iregexp '[a-c]+' |
+---------------------------------+
| true                            |
+---------------------------------+
```

Related information:

REGEXP Operator on page 189

IS DISTINCT FROM Operator

The `IS DISTINCT FROM` operator, and its converse the `IS NOT DISTINCT FROM` operator, test whether or not values are identical. `IS NOT DISTINCT FROM` is similar to the = operator, and `IS DISTINCT FROM` is similar to the `!=` operator, except that NULL values are treated as identical. Therefore, `IS NOT DISTINCT FROM` returns `true` rather than NULL, and `IS DISTINCT FROM` returns `false` rather than NULL, when comparing two NULL values. If one of the values being compared is NULL and the other is not, `IS DISTINCT FROM` returns `true` and `IS NOT DISTINCT FROM` returns `false`, again instead of returning NULL in both cases.

Syntax:

```
expression1 IS DISTINCT FROM expression2

expression1 IS NOT DISTINCT FROM expression2
expression1 <=> expression2
```

The operator <=> is an alias for `IS NOT DISTINCT FROM`. It is typically used as a NULL-safe equality operator in join queries. That is, A `<=>` B is true if A equals B or if both A and B are NULL.

Usage notes:

This operator provides concise notation for comparing two values and always producing a `true` or `false` result, without treating `NULL` as a special case. Otherwise, to unambiguously distinguish between two values requires a compound expression involving `IS [NOT] NULL` tests of both operands in addition to the = or `!=` operator.

The <=> operator, used like an equality operator in a join query, is more efficient than the equivalent clause: `A = B OR (A IS NULL AND B IS NULL)`. The <=> operator can use a hash join, while the `OR` expression cannot.

Examples:

The following examples show how `IS DISTINCT FROM` gives output similar to the `!=` operator, and `IS NOT DISTINCT FROM` gives output similar to the = operator. The exception is when the expression involves a `NULL` value on one side or both sides, where `!=` and = return `NULL` but the `IS [NOT] DISTINCT FROM` operators still return `true` or `false`.

```
select 1 is distinct from 0, 1 != 0;
+----------------------+--------+
| 1 is distinct from 0 | 1 != 0 |
+----------------------+--------+
| true                 | true   |
+----------------------+--------+

select 1 is distinct from 1, 1 != 1;
+----------------------+--------+
| 1 is distinct from 1 | 1 != 1 |
+----------------------+--------+
| false                | false  |
+----------------------+--------+

select 1 is distinct from null, 1 != null;
+-------------------------+-----------+
| 1 is distinct from null | 1 != null |
+-------------------------+-----------+
| true                    | NULL      |
+-------------------------+-----------+

select null is distinct from null, null != null;
+---------------------------+--------------+
| null is distinct from null | null != null |
+---------------------------+--------------+
| false                     | NULL         |
+---------------------------+--------------+

select 1 is not distinct from 0, 1 = 0;
+--------------------------+-------+
| 1 is not distinct from 0 | 1 = 0 |
+--------------------------+-------+
| false                    | false |
+--------------------------+-------+

select 1 is not distinct from 1, 1 = 1;
+--------------------------+-------+
| 1 is not distinct from 1 | 1 = 1 |
+--------------------------+-------+
| true                     | true  |
+--------------------------+-------+

select 1 is not distinct from null, 1 = null;
+---------------------------+----------+
| 1 is not distinct from null | 1 = null |
+---------------------------+----------+
| false                     | NULL     |
+---------------------------+----------+

select null is not distinct from null, null = null;
+-------------------------------+-------------+
| null is not distinct from null | null = null |
+-------------------------------+-------------+
| true                          | NULL        |
+-------------------------------+-------------+
```

The following example shows how IS DISTINCT FROM considers CHAR values to be the same (not distinct from each other) if they only differ in the number of trailing spaces. Therefore, sometimes the result of an IS [NOT] DISTINCT FROM operator differs depending on whether the values are STRING/VARCHAR or CHAR.

```
select
  'x' is distinct from 'x ' as string_with_trailing_spaces,
  cast('x' as char(5)) is distinct from cast('x ' as char(5)) as
char_with_trailing_spaces;
+-----------------------------+--------------------------+
| string_with_trailing_spaces | char_with_trailing_spaces |
+-----------------------------+--------------------------+
| true                        | false                    |
+-----------------------------+--------------------------+
```

IS NULL Operator

The IS NULL operator, and its converse the IS NOT NULL operator, test whether a specified value is NULL. Because using NULL with any of the other comparison operators such as = or != also returns NULL rather than TRUE or FALSE, you use a special-purpose comparison operator to check for this special condition.

In CDH 5.14 / Impala 2.11 and higher, you can use the operators IS UNKNOWN and IS NOT UNKNOWN as synonyms for IS NULL and IS NOT NULL, respectively.

Syntax:

```
expression IS NULL
expression IS NOT NULL

expression IS UNKNOWN
expression IS NOT UNKNOWN
```

Usage notes:

In many cases, NULL values indicate some incorrect or incomplete processing during data ingestion or conversion. You might check whether any values in a column are NULL, and if so take some followup action to fill them in.

With sparse data, often represented in “wide” tables, it is common for most values to be NULL with only an occasional non-NULL value. In those cases, you can use the IS NOT NULL operator to identify the rows containing any data at all for a particular column, regardless of the actual value.

With a well-designed database schema, effective use of NULL values and IS NULL and IS NOT NULL operators can save having to design custom logic around special values such as 0, -1, 'N/A', empty string, and so on. NULL lets you distinguish between a value that is known to be 0, false, or empty, and a truly unknown value.

Complex type considerations:

The IS [NOT] UNKNOWN operator, as with the IS [NOT] NULL operator, is not applicable to complex type columns (STRUCT, ARRAY, or MAP). Using a complex type column with this operator causes a query error.

Examples:

```
-- If this value is non-zero, something is wrong.
select count(*) from employees where employee_id is null;

-- With data from disparate sources, some fields might be blank.
-- Not necessarily an error condition.
select count(*) from census where household_income is null;

-- Sometimes we expect fields to be null, and followup action
-- is needed when they are not.
select count(*) from web_traffic where weird_http_code is not null;
```

IS TRUE Operator

This variation of the IS operator tests for truth or falsity, with right-hand arguments [NOT] TRUE, [NOT] FALSE, and [NOT] UNKNOWN.

Syntax:

```
expression IS TRUE
expression IS NOT TRUE

expression IS FALSE
expression IS NOT FALSE
```

Usage notes:

This IS TRUE and IS FALSE forms are similar to doing equality comparisons with the Boolean values TRUE and FALSE, except that IS TRUE and IS FALSE always return either TRUE or FALSE, even if the left-hand side expression returns NULL

These operators let you simplify Boolean comparisons that must also check for NULL, for example X != 10 AND X IS NOT NULL is equivalent to (X != 10) IS TRUE.

In CDH 5.14 / Impala 2.11 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE(), ISNOTTRUE(), ISFALSE(), and ISNOTFALSE().

Complex type considerations:

The IS [NOT] TRUE and IS [NOT] FALSE operators are not applicable to complex type columns (STRUCT, ARRAY, or MAP). Using a complex type column with these operators causes a query error.

Added in: CDH 5.14.0 / Impala 2.11.0

Examples:

```
select assertion, b, b is true, b is false, b is unknown
  from boolean_test;
+-------------+-------+-----------+------------+-----------+
| assertion   | b     | istrue(b) | isfalse(b) | b is null |
+-------------+-------+-----------+------------+-----------+
| 2 + 2 = 4   | true  | true      | false      | false     |
| 2 + 2 = 5   | false | false     | true       | false     |
| 1 = null    | NULL  | false     | false      | true      |
| null = null | NULL  | false     | false      | true      |
+-------------+-------+-----------+------------+-----------+
```

LIKE Operator

A comparison operator for STRING data, with basic wildcard capability using the underscore (_) to match a single character and the percent sign (%) to match multiple characters. The argument expression must match the entire string value. Typically, it is more efficient to put any % wildcard match at the end of the string.

Syntax:

```
string_expression LIKE wildcard_expression
string_expression NOT LIKE wildcard_expression
```

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

```
select distinct c_last_name from customer where c_last_name like 'Mc%' or c_last_name
like 'Mac%';
select count(c_last_name) from customer where c_last_name like 'M%';
select c_email_address from customer where c_email_address like '%.edu';

-- We can find 4-letter names beginning with 'M' by calling functions...
select distinct c_last_name from customer where length(c_last_name) = 4 and
substr(c_last_name,1,1) = 'M';
-- ...or in a more readable way by matching M followed by exactly 3 characters.
select distinct c_last_name from customer where c_last_name like 'M___';
```

For case-insensitive comparisons, see ILIKE Operator on page 179. For a more general kind of search operator using regular expressions, see REGEXP Operator on page 189 or its case-insensitive counterpart IREGEXP Operator on page 182.

Logical Operators

Logical operators return a BOOLEAN value, based on a binary or unary logical operation between arguments that are also Booleans. Typically, the argument expressions use comparison operators.

Syntax:

```
boolean_expression binary_logical_operator boolean_expression
unary_logical_operator boolean_expression
```

The Impala logical operators are:

- AND: A binary operator that returns `true` if its left-hand and right-hand arguments both evaluate to `true`, `NULL` if either argument is `NULL`, and `false` otherwise.
- OR: A binary operator that returns `true` if either of its left-hand and right-hand arguments evaluate to `true`, `NULL` if one argument is `NULL` and the other is either `NULL` or `false`, and `false` otherwise.
- NOT: A unary operator that flips the state of a Boolean expression from `true` to `false`, or `false` to `true`. If the argument expression is `NULL`, the result remains `NULL`. (When `NOT` is used this way as a unary logical operator, it works differently than the `IS NOT NULL` comparison operator, which returns `true` when applied to a `NULL`.)

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in an arithmetic expression, such as multiplying by 10:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- When we refer to the scalar value using dot notation,
```

```
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
  from region, region.r_nations as nation
where
  nation.item.n_nationkey between 3 and 5
  or nation.item.n_nationkey < 15;
+-------------+----------------+------------------+
| r_name      | item.n_name    | item.n_nationkey |
+-------------+----------------+------------------+
| EUROPE      | UNITED KINGDOM | 23               |
| EUROPE      | RUSSIA         | 22               |
| EUROPE      | ROMANIA        | 19               |
| ASIA        | VIETNAM        | 21               |
| ASIA        | CHINA          | 18               |
| AMERICA     | UNITED STATES  | 24               |
| AMERICA     | PERU           | 17               |
| AMERICA     | CANADA         | 3                |
| MIDDLE EAST | SAUDI ARABIA   | 20               |
| MIDDLE EAST | EGYPT          | 4                |
| AFRICA      | MOZAMBIQUE     | 16               |
| AFRICA      | ETHIOPIA       | 5                |
+-------------+----------------+------------------+
```

Examples:

These examples demonstrate the AND operator:

```
[localhost:21000] > select true and true;
+---------------+
| true and true |
+---------------+
| true          |
+---------------+
[localhost:21000] > select true and false;
+----------------+
| true and false |
+----------------+
| false          |
+----------------+
[localhost:21000] > select false and false;
+-----------------+
| false and false |
+-----------------+
| false           |
+-----------------+
[localhost:21000] > select true and null;
+---------------+
| true and null |
+---------------+
| NULL          |
+---------------+
[localhost:21000] > select (10 > 2) and (6 != 9);
+-----------------------+
| (10 > 2) and (6 != 9) |
+-----------------------+
| true                  |
+-----------------------+
```

These examples demonstrate the OR operator:

```
[localhost:21000] > select true or true;
+--------------+
| true or true |
+--------------+
| true         |
+--------------+
[localhost:21000] > select true or false;
+---------------+
| true or false |
+---------------+
```

```
| true          |
+---------------+
[localhost:21000] > select false or false;
+----------------+
| false or false |
+----------------+
| false          |
+----------------+
[localhost:21000] > select true or null;
+--------------+
| true or null |
+--------------+
| true         |
+--------------+
[localhost:21000] > select null or true;
+--------------+
| null or true |
+--------------+
| true         |
+--------------+
[localhost:21000] > select false or null;
+---------------+
| false or null |
+---------------+
| NULL          |
+---------------+
[localhost:21000] > select (1 = 1) or ('hello' = 'world');
+--------------------------------+
| (1 = 1) or ('hello' = 'world') |
+--------------------------------+
| true                           |
+--------------------------------+
[localhost:21000] > select (2 + 2 != 4) or (-1 > 0);
+--------------------------+
| (2 + 2 != 4) or (-1 > 0) |
+--------------------------+
| false                    |
+--------------------------+
```

These examples demonstrate the NOT operator:

```
[localhost:21000] > select not true;
+----------+
| not true |
+----------+
| false    |
+----------+
[localhost:21000] > select not false;
+-----------+
| not false |
+-----------+
| true      |
+-----------+
[localhost:21000] > select not null;
+----------+
| not null |
+----------+
| NULL     |
+----------+
[localhost:21000] > select not (1=1);
+-------------+
| not (1 = 1) |
+-------------+
| false       |
+-------------+
```

REGEXP Operator

Tests whether a value matches a regular expression. Uses the POSIX regular expression syntax where ^ and $ match the beginning and end of the string, . represents any single character, * represents a sequence of zero or more items, + represents a sequence of one or more items, ? produces a non-greedy match, and so on.

Syntax:

```
string_expression REGEXP regular_expression
```

Usage notes:

The RLIKE operator is a synonym for REGEXP.

The | symbol is the alternation operator, typically used within ( ) to match different sequences. The ( ) groups do not allow backreferences. To retrieve the part of a value matched within a ( ) section, use the regexp_extract() built-in function.

In Impala 1.3.1 and higher, the REGEXP and RLIKE operators now match a regular expression string that occurs anywhere inside the target string, the same as if the regular expression was enclosed on each side by .*. See REGEXP Operator on page 189 for examples. Previously, these operators only succeeded when the regular expression matched the entire target string. This change improves compatibility with the regular expression support for popular database systems. There is no change to the behavior of the regexp_extract() and regexp_replace() built-in functions.

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

The following examples demonstrate the identical syntax for the REGEXP and RLIKE operators.

```
-- Find all customers whose first name starts with 'J', followed by 0 or more of any
character.
select c_first_name, c_last_name from customer where c_first_name regexp '^J.*';
select c_first_name, c_last_name from customer where c_first_name rlike '^J.*';

-- Find 'Macdonald', where the first 'a' is optional and the 'D' can be upper- or
lowercase.
-- The ^...$ are required, to match the start and end of the value.
select c_first_name, c_last_name from customer where c_last_name regexp '^Ma?c[Dd]onald$';
select c_first_name, c_last_name from customer where c_last_name rlike '^Ma?c[Dd]onald$';

-- Match multiple character sequences, either 'Mac' or 'Mc'.
select c_first_name, c_last_name from customer where c_last_name regexp
'^(Mac|Mc)donald$';
select c_first_name, c_last_name from customer where c_last_name rlike '^(Mac|Mc)donald$';

-- Find names starting with 'S', then one or more vowels, then 'r', then any other
characters.
-- Matches 'Searcy', 'Sorenson', 'Sauer'.
select c_first_name, c_last_name from customer where c_last_name regexp '^S[aeiou]+r.*$';
select c_first_name, c_last_name from customer where c_last_name rlike '^S[aeiou]+r.*$';

-- Find names that end with 2 or more vowels: letters from the set a,e,i,o,u.
select c_first_name, c_last_name from customer where c_last_name regexp '.*[aeiou]{2,}$';
select c_first_name, c_last_name from customer where c_last_name rlike '.*[aeiou]{2,}$';

-- You can use letter ranges in the [] blocks, for example to find names starting with
 A, B, or C.
select c_first_name, c_last_name from customer where c_last_name regexp '^[A-C].*';
select c_first_name, c_last_name from customer where c_last_name rlike '^[A-C].*';
```

```
-- If you are not sure about case, leading/trailing spaces, and so on, you can process
 the
-- column using string functions first.
select c_first_name, c_last_name from customer where lower(trim(c_last_name)) regexp
'^de.*';
select c_first_name, c_last_name from customer where lower(trim(c_last_name)) rlike
'^de.*';
```

Related information:

For regular expression matching with case-insensitive comparisons, see IREGEXP Operator on page 182.

RLIKE Operator

Synonym for the REGEXP operator. See REGEXP Operator on page 189 for details.

Examples:

The following examples demonstrate the identical syntax for the REGEXP and RLIKE operators.

```
-- Find all customers whose first name starts with 'J', followed by 0 or more of any
character.
select c_first_name, c_last_name from customer where c_first_name regexp '^J.*';
select c_first_name, c_last_name from customer where c_first_name rlike '^J.*';

-- Find 'Macdonald', where the first 'a' is optional and the 'D' can be upper- or
lowercase.
-- The ^...$ are required, to match the start and end of the value.
select c_first_name, c_last_name from customer where c_last_name regexp '^Ma?c[Dd]onald$';
select c_first_name, c_last_name from customer where c_last_name rlike '^Ma?c[Dd]onald$';

-- Match multiple character sequences, either 'Mac' or 'Mc'.
select c_first_name, c_last_name from customer where c_last_name regexp
'^(Mac|Mc)donald$';
select c_first_name, c_last_name from customer where c_last_name rlike '^(Mac|Mc)donald$';

-- Find names starting with 'S', then one or more vowels, then 'r', then any other
characters.
-- Matches 'Searcy', 'Sorenson', 'Sauer'.
select c_first_name, c_last_name from customer where c_last_name regexp '^S[aeiou]+r.*$';
select c_first_name, c_last_name from customer where c_last_name rlike '^S[aeiou]+r.*$';

-- Find names that end with 2 or more vowels: letters from the set a,e,i,o,u.
select c_first_name, c_last_name from customer where c_last_name regexp '.*[aeiou]{2,}$';
select c_first_name, c_last_name from customer where c_last_name rlike '.*[aeiou]{2,}$';

-- You can use letter ranges in the [] blocks, for example to find names starting with
 A, B, or C.
select c_first_name, c_last_name from customer where c_last_name regexp '^[A-C].*';
select c_first_name, c_last_name from customer where c_last_name rlike '^[A-C].*';

-- If you are not sure about case, leading/trailing spaces, and so on, you can process
 the
-- column using string functions first.
select c_first_name, c_last_name from customer where lower(trim(c_last_name)) regexp
'^de.*';
select c_first_name, c_last_name from customer where lower(trim(c_last_name)) rlike
'^de.*';
```

Impala Schema Objects and Object Names

With Impala, you work with schema objects that are familiar to database users: primarily databases, tables, views, and functions. The SQL syntax to work with these objects is explained in Impala SQL Statements on page 202. This section explains the conceptual knowledge you need to work with these objects and the various ways to specify their names.

Within a table, partitions can also be considered a kind of object. Partitioning is an important subject for Impala, with its own documentation section covering use cases and performance considerations. See Partitioning for Impala Tables on page 625 for details.

Impala does not have a counterpart of the “tablespace” notion from some database systems. By default, all the data files for a database, table, or partition are located within nested folders within the HDFS file system. You can also specify a particular HDFS location for a given Impala table or partition. The raw data for these objects is represented as a collection of data files, providing the flexibility to load data by simply moving files into the expected HDFS location.

Information about the schema objects is held in the metastore database. This database is shared between Impala and Hive, allowing each to create, drop, and query each other's databases, tables, and so on. When Impala makes a change to schema objects through a CREATE, ALTER, DROP, INSERT, or LOAD DATA statement, it broadcasts those changes to all nodes in the cluster through the catalog service. When you make such changes through Hive or directly through manipulating HDFS files, you use the REFRESH or INVALIDATE METADATA statements on the Impala side to recognize the newly loaded data, new tables, and so on.

Overview of Impala Aliases

When you write the names of tables, columns, or column expressions in a query, you can assign an alias at the same time. Then you can specify the alias rather than the original name when making other references to the table or column in the same statement. You typically specify aliases that are shorter, easier to remember, or both than the original names. The aliases are printed in the query header, making them useful for self-documenting output.

To set up an alias, add the AS *alias* clause immediately after any table, column, or expression name in the SELECT list or FROM list of a query. The AS keyword is optional; you can also specify the alias immediately after the original name.

```
-- Make the column headers of the result set easier to understand.
SELECT c1 AS name, c2 AS address, c3 AS phone FROM table_with_terse_columns;
SELECT SUM(ss_xyz_dollars_net) AS total_sales FROM table_with_cryptic_columns;
-- The alias can be a quoted string for extra readability.
SELECT c1 AS "Employee ID", c2 AS "Date of hire" FROM t1;
-- The AS keyword is optional.
SELECT c1 "Employee ID", c2 "Date of hire" FROM t1;

-- The table aliases assigned in the FROM clause can be used both earlier
-- in the query (the SELECT list) and later (the WHERE clause).
SELECT one.name, two.address, three.phone
  FROM census one, building_directory two, phonebook three
WHERE one.id = two.id and two.id = three.id;

-- The aliases c1 and c2 let the query handle columns with the same names from 2 joined
 tables.
-- The aliases t1 and t2 let the query abbreviate references to long or cryptically
named tables.
SELECT t1.column_n AS c1, t2.column_n AS c2 FROM long_name_table AS t1,
very_long_name_table2 AS t2
  WHERE c1 = c2;
SELECT t1.column_n c1, t2.column_n c2 FROM table1 t1, table2 t2
  WHERE c1 = c2;
```

You can specify column aliases with or without the AS keyword, and with no quotation marks, single quotation marks, or double quotation marks. Some kind of quotation marks are required if the column alias contains any spaces or other problematic characters. The alias text is displayed in the impala-shell output as all-lowercase. For example:

```
[localhost:21000] > select c1 First_Column from t;
[localhost:21000] > select c1 as First_Column from t;
+--------------+
| first_column |
+--------------+
...

[localhost:21000] > select c1 'First Column' from t;
[localhost:21000] > select c1 as 'First Column' from t;
+--------------+
| first column |
+--------------+
...

[localhost:21000] > select c1 "First Column" from t;
[localhost:21000] > select c1 as "First Column" from t;
```

```
+--------------+
| first column |
+--------------+
...
```

From Impala 3.0, the alias substitution logic in the GROUP BY, HAVING, and ORDER BY clauses has become more consistent with standard SQL behavior, as follows. Aliases are now only legal at the top level, and not in subexpressions. The following statements are allowed:

```
SELECT int_col / 2 AS x
FROM t
GROUP BY x;

SELECT int_col / 2 AS x
FROM t
ORDER BY x;

SELECT NOT bool_col AS nb
FROM t
GROUP BY nb
HAVING nb;
```

And the following statements are NOT allowed:

```
SELECT int_col / 2 AS x
FROM t
GROUP BY x / 2;

SELECT int_col / 2 AS x
FROM t
ORDER BY -x;

SELECT int_col / 2 AS x
FROM t
GROUP BY x
HAVING x > 3;
```

To use an alias name that matches one of the Impala reserved keywords (listed in Impala Reserved Words on page 745), surround the identifier with either single or double quotation marks, or `` characters (backticks).

Aliases follow the same rules as identifiers when it comes to case insensitivity. Aliases can be longer than identifiers (up to the maximum length of a Java string) and can include additional characters such as spaces and dashes when they are quoted using backtick characters.

Complex type considerations:

Queries involving the complex types (ARRAY, STRUCT, and MAP), typically make extensive use of table aliases. These queries involve join clauses where the complex type column is treated as a joined table. To construct two-part or three-part qualified names for the complex column elements in the FROM list, sometimes it is syntactically required to construct a table alias for the complex column where it is referenced in the join clause. See Complex Types (CDH 5.5 or higher only) on page 139 for details and examples.

Alternatives:

Another way to define different names for the same tables or columns is to create views. See Overview of Impala Views on page 199 for details.

Overview of Impala Databases

In Impala, a database is a logical container for a group of tables. Each database defines a separate namespace. Within a database, you can refer to the tables inside it using their unqualified names. Different databases can contain tables with identical names.

Creating a database is a lightweight operation. There are minimal database-specific properties to configure, such as LOCATION and COMMENT.

You can change the owner of a database with the ALTER DATABASE statement.

Typically, you create a separate database for each project or application, to avoid naming conflicts between tables and to make clear which tables are related to each other. The USE statement lets you switch between databases. Unqualified references to tables, views, and functions refer to objects within the current database. You can also refer to objects in other databases by using qualified names of the form *dbname*.*object_name*.

Each database is physically represented by a directory in HDFS. When you do not specify a LOCATION attribute, the directory is located in the Impala data directory with the associated tables managed by Impala. When you do specify a LOCATION attribute, any read and write operations for tables in that database are relative to the specified HDFS directory.

There is a special database, named default, where you begin when you connect to Impala. Tables created in default are physically located one level higher in HDFS than all the user-created databases.

Impala includes another predefined database, _impala_builtins, that serves as the location for the built-in functions. To see the built-in functions, use a statement like the following:

```
show functions in _impala_builtins;
show functions in _impala_builtins like '*substring*';
```

Related statements:

CREATE DATABASE Statement on page 226, DROP DATABASE Statement on page 262, USE Statement on page 385, SHOW DATABASES on page 368

Overview of Impala Functions

Functions let you apply arithmetic, string, or other computations and transformations to Impala data. You typically use them in SELECT lists and WHERE clauses to filter and format query results so that the result set is exactly what you want, with no further processing needed on the application side.

Scalar functions return a single result for each input row. See Impala Built-In Functions on page 391.

```
[localhost:21000] > select name, population from country where continent = 'North America'
 order by population desc limit 4;
[localhost:21000] > select upper(name), population from country where continent = 'North
 America' order by population desc limit 4;
+-------------+------------+
| upper(name) | population |
+-------------+------------+
| USA         | 320000000  |
| MEXICO      | 122000000  |
| CANADA      | 25000000   |
| GUATEMALA   | 16000000   |
+-------------+------------+
```

Aggregate functions combine the results from multiple rows: either a single result for the entire table, or a separate result for each group of rows. Aggregate functions are frequently used in combination with GROUP BY and HAVING clauses in the SELECT statement. See Impala Aggregate Functions on page 479.

```
[localhost:21000] > select continent, sum(population) as howmany from country group by
 continent order by howmany desc;
+---------------+------------+
| continent     | howmany    |
+---------------+------------+
| Asia          | 4298723000 |
| Africa        | 1110635000 |
| Europe        | 742452000  |
| North America | 565265000  |
| South America | 406740000  |
| Oceania       | 38304000   |
+---------------+------------+
```

User-defined functions (UDFs) let you code your own logic. They can be either scalar or aggregate functions. UDFs let you implement important business or scientific logic using high-performance code for Impala to automatically parallelize. You can also use UDFs to implement convenience functions to simplify reporting or porting SQL from other database systems. See User-Defined Functions (UDFs) on page 525.

```
[localhost:21000] > select rot13('Hello world!') as 'Weak obfuscation';
+------------------+
| weak obfuscation |
+------------------+
| Uryyb jbeyq!     |
+------------------+
[localhost:21000] > select likelihood_of_new_subatomic_particle(sensor1, sensor2, sensor3)
 as probability
                  > from experimental_results group by experiment;
```

Each function is associated with a specific database. For example, if you issue a `USE somedb` statement followed by `CREATE FUNCTION somefunc`, the new function is created in the `somedb` database, and you could refer to it through the fully qualified name `somedb.somefunc`. You could then issue another `USE` statement and create a function with the same name in a different database.

Impala built-in functions are associated with a special database named `_impala_builtins`, which lets you refer to them from any database without qualifying the name.

```
[localhost:21000] > show databases;
+-------------------------+
| name                    |
+-------------------------+
| _impala_builtins        |
| analytic_functions      |
| avro_testing            |
| data_file_size          |
...
[localhost:21000] > show functions in _impala_builtins like '*subs*';
+-------------+----------------------------------+
| return type | signature                        |
+-------------+----------------------------------+
| STRING      | substr(STRING, BIGINT)           |
| STRING      | substr(STRING, BIGINT, BIGINT)   |
| STRING      | substring(STRING, BIGINT)        |
| STRING      | substring(STRING, BIGINT, BIGINT) |
+-------------+----------------------------------+
```

Related statements: CREATE FUNCTION Statement on page 228, DROP FUNCTION Statement on page 263

Overview of Impala Identifiers

Identifiers are the names of databases, tables, or columns that you specify in a SQL statement. The rules for identifiers govern what names you can give to things you create, the notation for referring to names containing unusual characters, and other aspects such as case sensitivity.

- The minimum length of an identifier is 1 character.
- The maximum length of an identifier is currently 128 characters, enforced by the metastore database.
- An identifier must start with an alphanumeric or underscore character. The remainder can contain any combination of alphanumeric characters and underscores. Quoting the identifier with backticks has no effect on the allowed characters in the name.
- An identifier can contain only ASCII characters.
- To use an identifier name that matches one of the Impala reserved keywords (listed in Impala Reserved Words on page 745), surround the identifier with `` characters (backticks). Quote the reserved word even if it is part of a fully qualified name. The following example shows how a reserved word can be used as a column name if it is

quoted with backticks in the CREATE TABLE statement, and how the column name must also be quoted with backticks in a query:

```
[localhost:21000] > create table reserved (`data` string);

[localhost:21000] > select data from reserved;
ERROR: AnalysisException: Syntax error in line 1:
select data from reserved
       ^
Encountered: DATA
Expected: ALL, CASE, CAST, DISTINCT, EXISTS, FALSE, IF, INTERVAL, NOT, NULL,
STRAIGHT_JOIN, TRUE, IDENTIFIER
CAUSED BY: Exception: Syntax error

[localhost:21000] > select reserved.data from reserved;
ERROR: AnalysisException: Syntax error in line 1:
select reserved.data from reserved
                ^
Encountered: DATA
Expected: IDENTIFIER
CAUSED BY: Exception: Syntax error

[localhost:21000] > select reserved.`data` from reserved;

[localhost:21000] >
```

Important: Because the list of reserved words grows over time as new SQL syntax is added, consider adopting coding conventions (especially for any automated scripts or in packaged applications) to always quote all identifiers with backticks. Quoting all identifiers protects your SQL from compatibility issues if new reserved words are added in later releases.

- Impala identifiers are always case-insensitive. That is, tables named `t1` and `T1` always refer to the same table, regardless of quote characters. Internally, Impala always folds all specified table and column names to lowercase. This is why the column headers in query output are always displayed in lowercase.

See Overview of Impala Aliases on page 192 for how to define shorter or easier-to-remember aliases if the original names are long or cryptic identifiers. Aliases follow the same rules as identifiers when it comes to case insensitivity. Aliases can be longer than identifiers (up to the maximum length of a Java string) and can include additional characters such as spaces and dashes when they are quoted using backtick characters.

Another way to define different names for the same tables or columns is to create views. See Overview of Impala Views on page 199 for details.

Overview of Impala Tables

Tables are the primary containers for data in Impala. They have the familiar row and column layout similar to other database systems, plus some features such as partitioning often associated with higher-end data warehouse systems.

Logically, each table has a structure based on the definition of its columns, partitions, and other properties.

Physically, each table that uses HDFS storage is associated with a directory in HDFS. The table data consists of all the data files underneath that directory:

- Internal tables are managed by Impala, and use directories inside the designated Impala work area.
- External tables use arbitrary HDFS directories, where the data files are typically shared between different Hadoop components.
- Large-scale data is usually handled by partitioned tables, where the data files are divided among different HDFS subdirectories.

Impala tables can also represent data that is stored in HBase, or in the Amazon S3 filesystem (CDH 5.4 / Impala 2.2 or higher). See Using Impala to Query HBase Tables on page 684 and Using Impala with the Amazon S3 Filesystem on page 692.

Impala queries ignore files with extensions commonly used for temporary work files by Hadoop tools. Any files with extensions `.tmp` or `.copying` are not considered part of the Impala table. The suffix matching is case-insensitive, so for example Impala ignores both `.copying` and `.COPYING` suffixes.

Internal Tables

The default kind of table produced by the `CREATE TABLE` statement is known as an internal table. (Its counterpart is the external table, produced by the `CREATE EXTERNAL TABLE` syntax.)

- Impala creates a directory in HDFS to hold the data files.
- You can create data in internal tables by issuing `INSERT` or `LOAD DATA` statements.
- If you add or replace data using HDFS operations, issue the `REFRESH` command in `impala-shell` so that Impala recognizes the changes in data files, block locations, and so on.
- When you issue a `DROP TABLE` statement, Impala physically removes all the data files from the directory.
- To see whether a table is internal or external, and its associated HDFS location, issue the statement `DESCRIBE FORMATTED` *`table_name`*. The `Table Type` field displays `MANAGED_TABLE` for internal tables and `EXTERNAL_TABLE` for external tables. The `Location` field displays the path of the table directory as an HDFS URI.
- When you issue an `ALTER TABLE` statement to rename an internal table, all data files are moved into the new HDFS directory for the table. The files are moved even if they were formerly in a directory outside the Impala data directory, for example in an internal table with a `LOCATION` attribute pointing to an outside HDFS directory.

Examples:

You can switch a table from internal to external, or from external to internal, by using the `ALTER TABLE` statement:

```
-- Switch a table from internal to external.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='TRUE');

-- Switch a table from external to internal.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='FALSE');
```

If the Kudu service is integrated with the Hive Metastore, the above operations are not supported.

External Tables

The syntax `CREATE EXTERNAL TABLE` sets up an Impala table that points at existing data files, potentially in HDFS locations outside the normal Impala data directories.. This operation saves the expense of importing the data into a new table when you already have the data files in a known location in HDFS, in the desired file format.

- You can use Impala to query the data in this table.
- You can create data in external tables by issuing `INSERT` or `LOAD DATA` statements.
- If you add or replace data using HDFS operations, issue the `REFRESH` command in `impala-shell` so that Impala recognizes the changes in data files, block locations, and so on.
- When you issue a `DROP TABLE` statement in Impala, that removes the connection that Impala has with the associated data files, but does not physically remove the underlying data. You can continue to use the data files with other Hadoop components and HDFS operations.
- To see whether a table is internal or external, and its associated HDFS location, issue the statement `DESCRIBE FORMATTED` *`table_name`*. The `Table Type` field displays `MANAGED_TABLE` for internal tables and `EXTERNAL_TABLE` for external tables. The `Location` field displays the path of the table directory as an HDFS URI.
- When you issue an `ALTER TABLE` statement to rename an external table, all data files are left in their original locations.

- You can point multiple external tables at the same HDFS directory by using the same LOCATION attribute for each one. The tables could have different column definitions, as long as the number and types of columns are compatible with the schema evolution considerations for the underlying file type. For example, for text data files, one table might define a certain column as a STRING while another defines the same column as a BIGINT.

Examples:

You can switch a table from internal to external, or from external to internal, by using the ALTER TABLE statement:

```
-- Switch a table from internal to external.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='TRUE');

-- Switch a table from external to internal.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='FALSE');
```

If the Kudu service is integrated with the Hive Metastore, the above operations are not supported.

File Formats

Each table has an associated file format, which determines how Impala interprets the associated data files. See How Impala Works with Hadoop File Formats on page 634 for details.

You set the file format during the CREATE TABLE statement, or change it later using the ALTER TABLE statement. Partitioned tables can have a different file format for individual partitions, allowing you to change the file format used in your ETL process for new data without going back and reconverting all the existing data in the same table.

Any INSERT statements produce new data files with the current file format of the table. For existing data files, changing the file format of the table does not automatically do any data conversion. You must use TRUNCATE TABLE or INSERT OVERWRITE to remove any previous data files that use the old file format. Then you use the LOAD DATA statement, INSERT ... SELECT, or other mechanism to put data files of the correct format into the table.

The default file format, text, is the most flexible and easy to produce when you are just getting started with Impala. The Parquet file format offers the highest query performance and uses compression to reduce storage requirements; therefore, where practical, use Parquet for Impala tables with substantial amounts of data. Also, the complex types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher are currently only supported with the Parquet file type. Based on your existing ETL workflow, you might use other file formats such as Avro, possibly doing a final conversion step to Parquet to take advantage of its performance for analytic queries.

Kudu Tables

By default, tables stored in Apache Kudu are treated specially, because Kudu manages its data independently of HDFS files.

All metadata that Impala needs is stored in the HMS.

When Kudu is not integrated with the HMS, when you create a Kudu table through Impala, the table is assigned an internal Kudu table name of the form impala::*db_name*.*table_name*. You can see the Kudu-assigned name in the output of DESCRIBE FORMATTED, in the kudu.table_name field of the table properties.

For Impala-Kudu managed tables, ALTER TABLE ... RENAME renames both the Impala and the Kudu table.

For Impala-Kudu external tables, ALTER TABLE ... RENAME renames just the Impala table. To change the Kudu table that an Impala external table points to, use ALTER TABLE *impala_name* SET TBLPROPERTIES('kudu.table_name' = '*different_kudu_table_name*'). The underlying Kudu table must already exist.

In practice, external tables are typically used to access underlying Kudu tables that were created outside of Impala, that is, through the Kudu API.

The SHOW TABLE STATS output for a Kudu table shows Kudu-specific details about the layout of the table. Instead of information about the number and sizes of files, the information is divided by the Kudu tablets. For each tablet, the output includes the fields # Rows (although this number is not currently computed), Start Key, Stop Key, Leader

Replica, and # Replicas. The output of SHOW COLUMN STATS, illustrating the distribution of values within each column, is the same for Kudu tables as for HDFS-backed tables.

If the Kudu service is not integrated with the Hive Metastore, the distinction between internal and external tables has some special details for Kudu tables. Tables created entirely through Impala are internal tables. The table name as represented within Kudu includes notation such as an impala:: prefix and the Impala database name. External Kudu tables are those created by a non-Impala mechanism, such as a user application calling the Kudu APIs. For these tables, the CREATE EXTERNAL TABLE syntax lets you establish a mapping from Impala to the existing Kudu table:

```
CREATE EXTERNAL TABLE impala_name STORED AS KUDU
  TBLPROPERTIES('kudu.table_name' = 'original_kudu_name');
```

External Kudu tables differ in one important way from other external tables: adding or dropping a column or range partition changes the data in the underlying Kudu table, in contrast to an HDFS-backed external table where existing data files are left untouched.

Overview of Impala Views

Views are lightweight logical constructs that act as aliases for queries. You can specify a view name in a query (a SELECT statement or the SELECT portion of an INSERT statement) where you would usually specify a table name.

A view lets you:

- Issue complicated queries with compact and simple syntax:

```
-- Take a complicated reporting query, plug it into a CREATE VIEW statement...
create view v1 as select c1, c2, avg(c3) from t1 group by c3 order by c1 desc limit 10;
-- ... and now you can produce the report with 1 line of code.
select * from v1;
```

- Reduce maintenance, by avoiding the duplication of complicated queries across multiple applications in multiple languages:

```
create view v2 as select t1.c1, t1.c2, t2.c3 from t1 join t2 on (t1.id = t2.id);
-- This simple query is safer to embed in reporting applications than the longer query
 above.
-- The view definition can remain stable even if the structure of the underlying tables
 changes.
select c1, c2, c3 from v2;
```

- Build a new, more refined query on top of the original query by adding new clauses, select-list expressions, function calls, and so on:

```
create view average_price_by_category as select category, avg(price) as avg_price from
 products group by category;
create view expensive_categories as select category, avg_price from
average_price_by_category order by avg_price desc limit 10000;
create view top_10_expensive_categories as select category, avg_price from
expensive_categories limit 10;
```

  This technique lets you build up several more or less granular variations of the same query, and switch between them when appropriate.

- Set up aliases with intuitive names for tables, columns, result sets from joins, and so on:

```
-- The original tables might have cryptic names inherited from a legacy system.
create view action_items as select rrptsk as assignee, treq as due_date, dmisc as notes
 from vxy_t1_br;
-- You can leave original names for compatibility, build new applications using more
intuitive ones.
select assignee, due_date, notes from action_items;
```

- Swap tables with others that use different file formats, partitioning schemes, and so on without any downtime for data copying or conversion:

```
create table slow (x int, s string) stored as textfile;
create view report as select s from slow where x between 20 and 30;
-- Query is kind of slow due to inefficient table definition, but it works.
select * from report;

create table fast (s string) partitioned by (x int) stored as parquet;
-- ...Copy data from SLOW to FAST. Queries against REPORT view continue to work...

-- After changing the view definition, queries will be faster due to partitioning,
-- binary format, and compression in the new table.
alter view report as select s from fast where x between 20 and 30;
select * from report;
```

- Avoid coding lengthy subqueries and repeating the same subquery text in many other queries.
- Set up fine-grained security where a user can query some columns from a table but not other columns. Because CDH 5.5 / Impala 2.3 and higher support column-level authorization, this technique is no longer required. If you formerly implemented column-level security through views, see Hive SQL Syntax for Use with Sentry for details about the column-level authorization feature.

The SQL statements that configure views are CREATE VIEW Statement on page 248, ALTER VIEW Statement on page 218, and DROP VIEW Statement on page 270. You can specify view names when querying data (SELECT Statement on page 295) and copying data from one table to another (INSERT Statement on page 277). The WITH clause creates an inline view, that only exists for the duration of a single query.

```
[localhost:21000] > create view trivial as select * from customer;
[localhost:21000] > create view some_columns as select c_first_name, c_last_name, c_login
 from customer;
[localhost:21000] > select * from some_columns limit 5;
Query finished, fetching results ...
+--------------+-------------+---------+
| c_first_name | c_last_name | c_login |
+--------------+-------------+---------+
| Javier       | Lewis       |         |
| Amy          | Moses       |         |
| Latisha      | Hamilton    |         |
| Michael      | White       |         |
| Robert       | Moran       |         |
+--------------+-------------+---------+
[localhost:21000] > create view ordered_results as select * from some_columns order by
 c_last_name desc, c_first_name desc limit 1000;
[localhost:21000] > select * from ordered_results limit 5;
Query: select * from ordered_results limit 5
Query finished, fetching results ...
+--------------+-------------+---------+
| c_first_name | c_last_name | c_login |
+--------------+-------------+---------+
| Thomas       | Zuniga      |         |
| Sarah        | Zuniga      |         |
| Norma        | Zuniga      |         |
| Lloyd        | Zuniga      |         |
| Lisa         | Zuniga      |         |
+--------------+-------------+---------+
Returned 5 row(s) in 0.48s
```

The previous example uses descending order for ORDERED_RESULTS because in the sample TPCD-H data, there are some rows with empty strings for both C_FIRST_NAME and C_LAST_NAME, making the lowest-ordered names unuseful in a sample query.

```
create view visitors_by_day as select day, count(distinct visitors) as howmany from
web_traffic group by day;
create view top_10_days as select day, howmany from visitors_by_day order by howmany
limit 10;
select * from top_10_days;
```

Usage notes:

To see the definition of a view, issue a DESCRIBE FORMATTED statement, which shows the query from the original CREATE VIEW statement:

```
[localhost:21000] > create view v1 as select * from t1;
[localhost:21000] > describe formatted v1;
Query finished, fetching results ...
+------------------------------+------------------------------+------------+
| name                         | type                         | comment    |
+------------------------------+------------------------------+------------+
| # col_name                   | data_type                    | comment    |
|                              | NULL                         | NULL       |
| x                            | int                          | None       |
| y                            | int                          | None       |
| s                            | string                       | None       |
|                              | NULL                         | NULL       |
| # Detailed Table Information | NULL                         | NULL       |
| Database:                    | views                        | NULL       |
| Owner:                       | doc_demo                     | NULL       |
| CreateTime:                  | Mon Jul 08 15:56:27 EDT 2013 | NULL       |
| LastAccessTime:              | UNKNOWN                      | NULL       |
| Protect Mode:                | None                         | NULL       |
| Retention:                   | 0                            | NULL       |
| Table Type:                  | VIRTUAL_VIEW                 | NULL       |
| Table Parameters:            | NULL                         | NULL       |
|                              | transient_lastDdlTime        | 1373313387 |
|                              | NULL                         | NULL       |
| # Storage Information        | NULL                         | NULL       |
| SerDe Library:               | null                         | NULL       |
| InputFormat:                 | null                         | NULL       |
| OutputFormat:                | null                         | NULL       |
| Compressed:                  | No                           | NULL       |
| Num Buckets:                 | 0                            | NULL       |
| Bucket Columns:              | []                           | NULL       |
| Sort Columns:                | []                           | NULL       |
|                              | NULL                         | NULL       |
| # View Information           | NULL                         | NULL       |
| View Original Text:          | SELECT * FROM t1             | NULL       |
| View Expanded Text:          | SELECT * FROM t1             | NULL       |
+------------------------------+------------------------------+------------+
```

Prior to Impala 1.4.0, it was not possible to use the CREATE TABLE LIKE *view_name* syntax. In Impala 1.4.0 and higher, you can create a table with the same column definitions as a view using the CREATE TABLE LIKE technique. Although CREATE TABLE LIKE normally inherits the file format of the original table, a view has no underlying file format, so CREATE TABLE LIKE *view_name* produces a text table by default. To specify a different file format, include a STORED AS *file_format* clause at the end of the CREATE TABLE LIKE statement.

Complex type considerations:

For tables containing complex type columns (ARRAY, STRUCT, or MAP), you typically use join queries to refer to the complex values. You can use views to hide the join notation, making such tables seem like traditional denormalized tables, and making those tables queryable by business intelligence tools that do not have built-in support for those complex types. See Accessing Complex Type Data in Flattened Form Using Views on page 159 for details.

The STRAIGHT_JOIN hint affects the join order of table references in the query block containing the hint. It does not affect the join order of nested queries, such as views, inline views, or WHERE-clause subqueries. To use this hint for performance tuning of complex queries, apply the hint to all query blocks that need a fixed join order.

Restrictions:

- You cannot insert into an Impala view. (In some database systems, this operation is allowed and inserts rows into the base table.) You can use a view name on the right-hand side of an INSERT statement, in the SELECT part.
- If a view applies to a partitioned table, any partition pruning considers the clauses on both the original query and any additional WHERE predicates in the query that refers to the view. Prior to Impala 1.4, only the WHERE clauses on the original query from the CREATE VIEW statement were used for partition pruning.

- An ORDER BY clause without an additional LIMIT clause is ignored in any view definition. If you need to sort the entire result set from a view, use an ORDER BY clause in the SELECT statement that queries the view. You can still make a simple "top 10" report by combining the ORDER BY and LIMIT clauses in the same view definition:

```
[localhost:21000] > create table unsorted (x bigint);
[localhost:21000] > insert into unsorted values (1), (9), (3), (7), (5), (8), (4), (6),
 (2);
[localhost:21000] > create view sorted_view as select x from unsorted order by x;
[localhost:21000] > select x from sorted_view; -- ORDER BY clause in view has no effect.
+---+
| x |
+---+
| 1 |
| 9 |
| 3 |
| 7 |
| 5 |
| 8 |
| 4 |
| 6 |
| 2 |
+---+
[localhost:21000] > select x from sorted_view order by x; -- View query requires ORDER
 BY at outermost level.
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
+---+
[localhost:21000] > create view top_3_view as select x from unsorted order by x limit
3;
[localhost:21000] > select x from top_3_view; -- ORDER BY and LIMIT together in view
definition are preserved.
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

- The TABLESAMPLE clause of the SELECT statement does not apply to a table reference derived from a view, a subquery, or anything other than a real base table. This clause only works for tables backed by HDFS or HDFS-like data files, therefore it does not apply to Kudu or HBase tables.

Related statements: CREATE VIEW Statement on page 248, ALTER VIEW Statement on page 218, DROP VIEW Statement on page 270

Impala SQL Statements

The Impala SQL dialect supports a range of standard elements, plus some extensions for Big Data use cases related to data loading and data warehousing.



Note:

In the `impala-shell` interpreter, a semicolon at the end of each statement is required. Since the semicolon is not actually part of the SQL syntax, we do not include it in the syntax definition of each statement, but we do show it in examples intended to be run in `impala-shell`.

DDL Statements

DDL refers to "Data Definition Language", a subset of SQL statements that change the structure of the database schema in some way, typically by creating, deleting, or modifying schema objects such as databases, tables, and views. Most Impala DDL statements start with the keywords CREATE, DROP, or ALTER.

The Impala DDL statements are:

- ALTER TABLE Statement on page 205
- ALTER VIEW Statement on page 218
- COMPUTE STATS Statement on page 219
- CREATE DATABASE Statement on page 226
- CREATE FUNCTION Statement on page 228
- CREATE ROLE Statement (CDH 5.2 or higher only) on page 234
- CREATE TABLE Statement on page 234
- CREATE VIEW Statement on page 248
- DROP DATABASE Statement on page 262
- DROP FUNCTION Statement on page 263
- DROP ROLE Statement (CDH 5.2 or higher only) on page 265
- DROP TABLE Statement on page 268
- DROP VIEW Statement on page 270
- GRANT Statement (CDH 5.2 or higher only) on page 273
- REVOKE Statement (CDH 5.2 or higher only) on page 293

After Impala executes a DDL command, information about available tables, columns, views, partitions, and so on is automatically synchronized between all the Impala nodes in a cluster. (Prior to Impala 1.2, you had to issue a REFRESH or INVALIDATE METADATA statement manually on the other nodes to make them aware of the changes.)

If the timing of metadata updates is significant, for example if you use round-robin scheduling where each query could be issued through a different Impala node, you can enable the SYNC_DDL query option to make the DDL statement wait until all nodes have been notified about the metadata changes.

See Using Impala with the Amazon S3 Filesystem on page 692 for details about how Impala DDL statements interact with tables and partitions stored in the Amazon S3 filesystem.

Although the INSERT statement is officially classified as a DML (data manipulation language) statement, it also involves metadata changes that must be broadcast to all Impala nodes, and so is also affected by the SYNC_DDL query option.

Because the SYNC_DDL query option makes each DDL operation take longer than normal, you might only enable it before the last DDL operation in a sequence. For example, if you are running a script that issues multiple of DDL operations to set up an entire new schema, add several new partitions, and so on, you might minimize the performance overhead by enabling the query option only before the last CREATE, DROP, ALTER, or INSERT statement. The script only finishes when all the relevant metadata changes are recognized by all the Impala nodes, so you could connect to any node and issue queries through it.

The classification of DDL, DML, and other statements is not necessarily the same between Impala and Hive. Impala organizes these statements in a way intended to be familiar to people familiar with relational databases or data warehouse products. Statements that modify the metastore database, such as COMPUTE STATS, are classified as DDL. Statements that only query the metastore database, such as SHOW or DESCRIBE, are put into a separate category of utility statements.



Note: The query types shown in the Impala debug web user interface might not match exactly the categories listed here. For example, currently the USE statement is shown as DDL in the debug web UI. The query types shown in the debug web UI are subject to change, for improved consistency.

Related information:

The other major classifications of SQL statements are data manipulation language (see DML Statements on page 204) and queries (see SELECT Statement on page 295).

DML Statements

DML refers to "Data Manipulation Language", a subset of SQL statements that modify the data stored in tables. Because Impala focuses on query performance and leverages the append-only nature of HDFS storage, currently Impala only supports a small set of DML statements:

- DELETE Statement (CDH 5.10 or higher only) on page 249. Works for Kudu tables only.
- INSERT Statement on page 277.
- LOAD DATA Statement on page 288. Does not apply for HBase or Kudu tables.
- UPDATE Statement (CDH 5.10 or higher only) on page 383. Works for Kudu tables only.
- UPSERT Statement (CDH 5.10 or higher only) on page 384. Works for Kudu tables only.

`INSERT` in Impala is primarily optimized for inserting large volumes of data in a single statement, to make effective use of the multi-megabyte HDFS blocks. This is the way in Impala to create new data files. If you intend to insert one or a few rows at a time, such as using the `INSERT ... VALUES` syntax, that technique is much more efficient for Impala tables stored in HBase. See Using Impala to Query HBase Tables on page 684 for details.

`LOAD DATA` moves existing data files into the directory for an Impala table, making them immediately available for Impala queries. This is one way in Impala to work with data files produced by other Hadoop components. (`CREATE EXTERNAL TABLE` is the other alternative; with external tables, you can query existing data files, while the files remain in their original location.)

In CDH 5.10 / Impala 2.8 and higher, Impala does support the `UPDATE`, `DELETE`, and `UPSERT` statements for Kudu tables. For HDFS or S3 tables, to simulate the effects of an `UPDATE` or `DELETE` statement in other database systems, typically you use `INSERT` or `CREATE TABLE AS SELECT` to copy data from one table to another, filtering out or changing the appropriate rows during the copy operation.

You can also achieve a result similar to `UPDATE` by using Impala tables stored in HBase. When you insert a row into an HBase table, and the table already contains a row with the same value for the key column, the older row is hidden, effectively the same as a single-row `UPDATE`.

Impala can perform DML operations for tables or partitions stored in the Amazon S3 filesystem with CDH 5.8 / Impala 2.6 and higher. See Using Impala with the Amazon S3 Filesystem on page 692 for details.

Related information:

The other major classifications of SQL statements are data definition language (see DDL Statements on page 203) and queries (see SELECT Statement on page 295).

ALTER DATABASE Statement

The `ALTER DATABASE` statement changes the characteristics of a database.

Use the `SET OWNER` clause to transfer the ownership of the database from the current owner to another user or a role.

The database owner is originally set to the user who creates the database. When object ownership is enabled in Sentry, an owner of a database can have the `ALL` with `GRANT` or `ALL` without `GRANT` privilege. The term `OWNER` is used to differentiate between the `ALL` privilege that is explicitly granted via the `GRANT` statement and a privilege that is implicitly granted by the `CREATE DATABASE` statement.

Syntax:

```
ALTER DATABASE database_name SET OWNER USER user_name;
ALTER DATABASE database_name SET OWNER ROLE role_name;
```

Statement type: DDL

Cancellation: Cannot be cancelled.

Added in: CDH 6.1 / Impala 3.1

ALTER TABLE Statement

The ALTER TABLE statement changes the structure or properties of an existing Impala table.

In Impala, this is primarily a logical operation that updates the table metadata in the metastore database that Impala shares with Hive. Most ALTER TABLE operations do not actually rewrite, move, and so on the actual data files. (The RENAME TO clause is the one exception; it can cause HDFS files to be moved to different paths.) When you do an ALTER TABLE operation, you typically need to perform corresponding physical filesystem operations, such as rewriting the data files to include extra fields, or converting them to a different file format.

Syntax:

```
ALTER TABLE [old_db_name.]old_table_name RENAME TO [new_db_name.]new_table_name

ALTER TABLE name ADD [IF NOT EXISTS] COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])

ALTER TABLE name ADD COLUMN [IF NOT EXISTS] col_spec
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name col_spec

ALTER TABLE name SET OWNER USER user_name
ALTER TABLE name SET OWNER ROLE role_name


-- Kudu tables only.
ALTER TABLE name ALTER [COLUMN] column_name
  { SET kudu_storage_attr attr_value
    | DROP DEFAULT }

kudu_storage_attr ::= { DEFAULT | BLOCK_SIZE | ENCODING | COMPRESSION }

-- Non-Kudu tables only.
ALTER TABLE name ALTER [COLUMN] column_name
  SET COMMENT 'comment_text'

ALTER TABLE name ADD [IF NOT EXISTS] PARTITION (partition_spec)
  [location_spec]
  [cache_spec]
ALTER TABLE name ADD [IF NOT EXISTS] RANGE PARTITION kudu_partition_spec

ALTER TABLE name DROP [IF EXISTS] PARTITION (partition_spec)
  [PURGE]
ALTER TABLE name DROP [IF EXISTS] RANGE PARTITION kudu_partition_spec

ALTER TABLE name RECOVER PARTITIONS

ALTER TABLE name [PARTITION (partition_spec)]
  SET { FILEFORMAT file_format
  | ROW FORMAT row_format
  | LOCATION 'hdfs_path_of_directory'
  | TBLPROPERTIES (table_properties)
  | SERDEPROPERTIES (serde_properties) }

ALTER TABLE name colname
  ('statsKey'='val, ...)

statsKey ::= numDVs | numNulls | avgSize | maxSize

ALTER TABLE name [PARTITION (partition_spec)] SET { CACHED IN 'pool_name' [WITH
REPLICATION = integer] | UNCACHED }

new_name ::= [new_database.]new_table_name

col_spec ::= col_name type_name  COMMENT 'column-comment' [kudu_attributes]

kudu_attributes ::= { [NOT] NULL | ENCODING codec | COMPRESSION algorithm |
  DEFAULT constant | BLOCK_SIZE number }

partition_spec ::= simple_partition_spec | complex_partition_spec

simple_partition_spec ::= partition_col=constant_value
```

```
complex_partition_spec ::= comparison_expression_on_partition_col

kudu_partition_spec ::= constant range_operator VALUES range_operator constant | VALUE
 = constant

cache_spec ::= CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED

location_spec ::= LOCATION 'hdfs_path_of_directory'

table_properties ::= 'name'='value'[, 'name'='value' ...]

serde_properties ::= 'name'='value'[, 'name'='value' ...]

file_format ::= { PARQUET | TEXTFILE | RCFILE | SEQUENCEFILE | AVRO }

row_format ::= DELIMITED
  [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
  [LINES TERMINATED BY 'char']
```

Statement type: DDL

Complex type considerations:

In CDH 5.5 / Impala 2.3 and higher, the ALTER TABLE statement can change the metadata for tables containing complex types (ARRAY, STRUCT, and MAP). For example, you can use an ADD COLUMNS, DROP COLUMN, or CHANGE clause to modify the table layout for complex type columns. Although Impala queries only work for complex type columns in Parquet tables, the complex type support in the ALTER TABLE statement applies to all file formats. For example, you can use Impala to update metadata for a staging table in a non-Parquet file format where the data is populated by Hive. Or you can use ALTER TABLE SET FILEFORMAT to change the format of an existing table to Parquet so that Impala can query it. Remember that changing the file format for a table does not convert the data files within the table; you must prepare any Parquet data files containing complex types outside Impala, and bring them into the table using LOAD DATA or updating the table's LOCATION property. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types.

Usage notes:

Whenever you specify partitions in an ALTER TABLE statement, through the PARTITION (*partition_spec*) clause, you must include all the partitioning columns in the specification.

Most of the ALTER TABLE operations work the same for internal tables (managed by Impala) as for external tables (with data files located in arbitrary locations). The exception is renaming a table; for an external table, the underlying data directory is not renamed or moved.

To drop or alter multiple partitions:

In CDH 5.10 / Impala 2.8 and higher, the expression for the partition clause with a DROP or SET operation can include comparison operators such as <, IN, or BETWEEN, and Boolean operators such as AND and OR.

For example, you might drop a group of partitions corresponding to a particular date range after the data “ages out”:

```
alter table historical_data drop partition (year < 1995);
alter table historical_data drop partition (year = 1996 and month between 1 and 6);
```

For tables with multiple partition keys columns, you can specify multiple conditions separated by commas, and the operation only applies to the partitions that match all the conditions (similar to using an AND clause):

```
alter table historical_data drop partition (year < 1995, last_name like 'A%');
```

This technique can also be used to change the file format of groups of partitions, as part of an ETL pipeline that periodically consolidates and rewrites the underlying data files in a different file format:

```
alter table fast_growing_data partition (year = 2016, month in (10,11,12)) set fileformat
 parquet;
```



Note:

The extended syntax involving comparison operators and multiple partitions applies to the SET FILEFORMAT, SET TBLPROPERTIES, SET SERDEPROPERTIES, and SET [UN]CACHED clauses. You can also use this syntax with the PARTITION clause in the COMPUTE INCREMENTAL STATS statement, and with the PARTITION clause of the SHOW FILES statement. Some forms of ALTER TABLE still only apply to one partition at a time: the SET LOCATION and ADD PARTITION clauses. The PARTITION clauses in the LOAD DATA and INSERT statements also only apply to one partition at a time.

A DDL statement that applies to multiple partitions is considered successful (resulting in no changes) even if no partitions match the conditions. The results are the same as if the IF EXISTS clause was specified.

The performance and scalability of this technique is similar to issuing a sequence of single-partition ALTER TABLE statements in quick succession. To minimize bottlenecks due to communication with the metastore database, or causing other DDL operations on the same table to wait, test the effects of performing ALTER TABLE statements that affect large numbers of partitions.

Amazon S3 considerations:

You can specify an s3a:// prefix on the LOCATION attribute of a table or partition to make Impala query data from the Amazon S3 filesystem. In CDH 5.8 / Impala 2.6 and higher, Impala automatically handles creating or removing the associated folders when you issue ALTER TABLE statements with the ADD PARTITION or DROP PARTITION clauses.

In CDH 5.8 / Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to CDH 5.8 / Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed. See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

HDFS caching (CACHED IN clause):

If you specify the CACHED IN clause, any existing or future data files in the table directory or the partition subdirectories are designated to be loaded into memory with the HDFS caching mechanism. See Using HDFS Caching with Impala (CDH 5.3 or higher only) on page 593 for details about using the HDFS caching feature.

In CDH 5.4 / Impala 2.2 and higher, the optional WITH REPLICATION clause for CREATE TABLE and ALTER TABLE lets you specify a ***replication factor***, the number of hosts on which to cache the same data blocks. When Impala processes a cached data block, where the cache replication factor is greater than 1, Impala randomly selects a host that has a cached copy of that data block. This optimization avoids excessive CPU usage on a single host when the same cached data block is processed multiple times. Cloudera recommends specifying a value greater than or equal to the HDFS block replication factor.

If you connect to different Impala nodes within an impala-shell session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.

The following sections show examples of the use cases for various ALTER TABLE clauses.

To rename a table (RENAME TO clause):

The RENAME TO clause lets you change the name of an existing table, and optionally which database it is located in.

For internal tables, this operation physically renames the directory within HDFS that contains the data files; the original directory name no longer exists. By qualifying the table names with database names, you can use this technique to move an internal table (and its associated data directory) from one database to another. For example:

```
create database d1;
create database d2;
create database d3;
use d1;
create table mobile (x int);
use d2;
-- Move table from another database to the current one.
alter table d1.mobile rename to mobile;
use d1;
-- Move table from one database to another.
alter table d2.mobile rename to d3.mobile;
```

To change the owner of a table:

```
ALTER TABLE name SET OWNER USER user_name;
ALTER TABLE name SET OWNER ROLE role_name;
```

The table owner is originally set to the user who creates the table. When object ownership is enabled in Sentry, an owner of a table can have the ALL with GRANT or ALL without GRANT privilege. The term OWNER is used to differentiate between the ALL privilege that is explicitly granted via the GRANT statement and a privilege that is implicitly granted by the CREATE TABLE statement.

Use the ALTER TABLE SET OWNER to transfer the ownership from the current owner to another user or a role.

To change the physical location where Impala looks for data files associated with a table or partition:

```
ALTER TABLE table_name [PARTITION (partition_spec)] SET LOCATION 'hdfs_path_of_directory';
```

The path you specify is the full HDFS path where the data files reside, or will be created. Impala does not create any additional subdirectory named after the table. Impala does not move any data files to this new location or change any data files that might already exist in that directory.

To set the location for a single partition, include the PARTITION clause. Specify all the same partitioning columns for the table, with a constant value for each, to precisely identify the single partition affected by the statement:

```
create table p1 (s string) partitioned by (month int, day int);
-- Each ADD PARTITION clause creates a subdirectory in HDFS.
alter table p1 add partition (month=1, day=1);
alter table p1 add partition (month=1, day=2);
alter table p1 add partition (month=2, day=1);
alter table p1 add partition (month=2, day=2);
-- Redirect queries, INSERT, and LOAD DATA for one partition
-- to a specific different directory.
alter table p1 partition (month=1, day=1) set location '/usr/external_data/new_years_day';
```



Note: If you are creating a partition for the first time and specifying its location, for maximum efficiency, use a single ALTER TABLE statement including both the ADD PARTITION and LOCATION clauses, rather than separate statements with ADD PARTITION and SET LOCATION clauses.

To automatically detect new partition directories added through Hive or HDFS operations:

In CDH 5.5 / Impala 2.3 and higher, the RECOVER PARTITIONS clause scans a partitioned table to detect if any new partition directories were added outside of Impala, such as by Hive ALTER TABLE statements or by hdfs dfs or hadoop fs commands. The RECOVER PARTITIONS clause automatically recognizes any data files present in these new directories, the same as the REFRESH statement does.

For example, here is a sequence of examples showing how you might create a partitioned table in Impala, create new partitions through Hive, copy data files into the new partitions with the `hdfs` command, and have Impala recognize the new partitions and new data:

In Impala, create the table, and a single partition for demonstration purposes:

```
create database recover_partitions;
use recover_partitions;
create table t1 (s string) partitioned by (yy int, mm int);
insert into t1 partition (yy = 2016, mm = 1) values ('Partition exists');
show files in t1;
+---------------------------------------------------------------------------+------+-------------+
| Path                                                                      | Size | Partition
    |
+---------------------------------------------------------------------------+------+-------------+
| /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1/data.txt | 17B  |
yy=2016/mm=1 |
+---------------------------------------------------------------------------+------+-------------+
quit;
```

In Hive, create some new partitions. In a real use case, you might create the partitions and populate them with data as the final stages of an ETL pipeline.

```
hive> use recover_partitions;
OK
hive> alter table t1 add partition (yy = 2016, mm = 2);
OK
hive> alter table t1 add partition (yy = 2016, mm = 3);
OK
hive> quit;
```

For demonstration purposes, manually copy data (a single row) into these new partitions, using manual HDFS operations:

```
$ hdfs dfs -ls /user/hive/warehouse/recover_partitions.db/t1/yy=2016/
Found 3 items
drwxr-xr-x - impala   hive 0 2016-05-09 16:06
/user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1
drwxr-xr-x - jrussell hive 0 2016-05-09 16:14
/user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=2
drwxr-xr-x - jrussell hive 0 2016-05-09 16:13
/user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=3

$ hdfs dfs -cp /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1/data.txt \
  /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=2/data.txt
$ hdfs dfs -cp /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1/data.txt \
  /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=3/data.txt
```

```
hive> select * from t1;
OK
Partition exists  2016  1
Partition exists  2016  2
Partition exists  2016  3
hive> quit;
```

In Impala, initially the partitions and data are not visible. Running ALTER TABLE with the RECOVER PARTITIONS clause scans the table data directory to find any new partition directories, and the data files inside them:

```
select * from t1;
+------------------+------+----+
| s                | yy   | mm |
+------------------+------+----+
| Partition exists | 2016 | 1  |
+------------------+------+----+

alter table t1 recover partitions;
select * from t1;
+------------------+------+----+
| s                | yy   | mm |
+------------------+------+----+
| Partition exists | 2016 | 1  |
| Partition exists | 2016 | 3  |
| Partition exists | 2016 | 2  |
+------------------+------+----+
```

To change the key-value pairs of the TBLPROPERTIES and SERDEPROPERTIES fields:

```
ALTER TABLE table_name SET TBLPROPERTIES ('key1'='value1', 'key2'='value2'[, ...]);
ALTER TABLE table_name SET SERDEPROPERTIES ('key1'='value1', 'key2'='value2'[, ...]);
```

The TBLPROPERTIES clause is primarily a way to associate arbitrary user-specified data items with a particular table.

The SERDEPROPERTIES clause sets up metadata defining how tables are read or written, needed in some cases by Hive but not used extensively by Impala. You would use this clause primarily to change the delimiter in an existing text table or partition, by setting the 'serialization.format' and 'field.delim' property values to the new delimiter character. The SERDEPROPERTIES clause does not change the existing data in the table. The change only affects the future inserts into the table.

Use the DESCRIBE FORMATTED statement to see the current values of these properties for an existing table.

See CREATE TABLE Statement on page 234 for more details about these clauses.

To manually set or update table or column statistics:

Although for most tables the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement is all you need to keep table and column statistics up to date for a table, sometimes for a very large table or one that is updated frequently, the length of time to recompute all the statistics might make it impractical to run those statements as often as needed. As a workaround, you can use the ALTER TABLE statement to set table statistics at the level of the entire table or a single partition, or column statistics at the level of the entire table.

You can set the numrows value for table statistics by changing the TBLPROPERTIES setting for a table or partition. For example:

```
create table analysis_data stored as parquet as select * from raw_data;
Inserted 1000000000 rows in 181.98s
compute stats analysis_data;
insert into analysis_data select * from smaller_table_we_forgot_before;
Inserted 1000000 rows in 15.32s
-- Now there are 1001000000 rows. We can update this single data point in the stats.
alter table analysis_data set tblproperties('numRows'='1001000000',
'STATS_GENERATED_VIA_STATS_TASK'='true');
```

```
-- If the table originally contained 1 million rows, and we add another partition with
 30 thousand rows,
-- change the numRows property for the partition and the overall table.
alter table partitioned_data partition(year=2009, month=4) set tblproperties
('numRows'='30000', 'STATS_GENERATED_VIA_STATS_TASK'='true');
alter table partitioned_data set tblproperties ('numRows'='1030000',
'STATS_GENERATED_VIA_STATS_TASK'='true');
```

See Setting Table Statistics on page 584 for details.

In CDH 5.8 / Impala 2.6 and higher, you can use the SET COLUMN STATS clause to set a specific stats value for a particular column.

You specify a case-insensitive symbolic name for the kind of statistics: numDVs, numNulls, avgSize, maxSize. The key names and values are both quoted. This operation applies to an entire table, not a specific partition. For example:

```
create table t1 (x int, s string);
insert into t1 values (1, 'one'), (2, 'two'), (2, 'deux');
show column stats t1;
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| x      | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
alter table t1 set column stats x ('numDVs'='2','numNulls'='0');
alter table t1 set column stats s ('numdvs'='3','maxsize'='4');
show column stats t1;
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| x      | INT    | 2                | 0      | 4        | 4        |
| s      | STRING | 3                | -1     | 4        | -1       |
+--------+--------+------------------+--------+----------+----------+
```

To reorganize columns for a table:

You can add multiple columns at a time using the ALTER TABLE statement. If you specify the IF NOT EXISTS clause, Impala silently ignores the ADD request and does not return an error if a column with the same name exists in the table.

When you replace columns, all the original column definitions are discarded.

You might use these statements if you receive a new set of data files with different data types or columns in a different order. The data files are retained, so if the new columns are incompatible with the old ones, use INSERT OVERWRITE or LOAD DATA OVERWRITE to replace all the data before issuing any further queries.

For example, here is how you might add columns to an existing table. The first ALTER TABLE adds two new columns, and the second ALTER TABLE adds one new column. A single Impala query reads both the old and new data files, containing different numbers of columns. For any columns not present in a particular data file, all the column values are considered to be NULL.

```
create table t1 (x int);
insert into t1 values (1), (2);

alter table t1 add columns (s string, t timestamp);
insert into t1 values (3, 'three', now());

alter table t1 add columns (b boolean);
insert into t1 values (4, 'four', now(), true);

select * from t1 order by x;
+---+-------+-------------------------------+------+
| x | s     | t                             | b    |
+---+-------+-------------------------------+------+
| 1 | NULL  | NULL                          | NULL |
| 2 | NULL  | NULL                          | NULL |
| 3 | three | 2016-05-11 11:19:45.054457000 | NULL |
| 4 | four  | 2016-05-11 11:20:20.260733000 | true |
+---+-------+-------------------------------+------+
```

You might use the CHANGE clause to rename a single column, or to treat an existing column as a different type than before, such as to switch between treating a column as STRING and TIMESTAMP, or between INT and BIGINT. You

can only drop a single column at a time; to drop multiple columns, issue multiple ALTER TABLE statements, or define the new set of columns with a single ALTER TABLE ... REPLACE COLUMNS statement.

The following examples show some safe operations to drop or change columns. Dropping the final column in a table lets Impala ignore the data causing any disruption to existing data files. Changing the type of a column works if existing data values can be safely converted to the new type. The type conversion rules depend on the file format of the underlying table. For example, in a text table, the same value can be interpreted as a STRING or a numeric value, while in a binary format such as Parquet, the rules are stricter and type conversions only work between certain sizes of integers.

```
create table optional_columns (x int, y int, z int, a1 int, a2 int);
insert into optional_columns values (1,2,3,0,0), (2,3,4,100,100);

-- When the last column in the table is dropped, Impala ignores the
-- values that are no longer needed. (Dropping A1 but leaving A2
-- would cause problems, as we will see in a subsequent example.)
alter table optional_columns drop column a2;
alter table optional_columns drop column a1;

select * from optional_columns;
+---+---+---+
| x | y | z |
+---+---+---+
| 1 | 2 | 3 |
| 2 | 3 | 4 |
+---+---+---+
```

```
create table int_to_string (s string, x int);
insert into int_to_string values ('one', 1), ('two', 2);

-- What was an INT column will now be interpreted as STRING.
-- This technique works for text tables but not other file formats.
-- The second X represents the new name of the column, which we keep the same.
alter table int_to_string change x x string;

-- Once the type is changed, we can insert non-integer values into the X column
-- and treat that column as a string, for example by uppercasing or concatenating.
insert into int_to_string values ('three', 'trois');
select s, upper(x) from int_to_string;
+-------+----------+
| s     | upper(x) |
+-------+----------+
| one   | 1        |
| two   | 2        |
| three | TROIS    |
+-------+----------+
```

Remember that Impala does not actually do any conversion for the underlying data files as a result of ALTER TABLE statements. If you use ALTER TABLE to create a table layout that does not agree with the contents of the underlying files, you must replace the files yourself, such as using LOAD DATA to load a new set of data files, or INSERT OVERWRITE to copy from another table and replace the original data.

The following example shows what happens if you delete the middle column from a Parquet table containing three columns. The underlying data files still contain three columns of data. Because the columns are interpreted based on their positions in the data file instead of the specific column names, a SELECT * query now reads the first and second columns from the data file, potentially leading to unexpected results or conversion errors. For this reason, if you expect to someday drop a column, declare it as the last column in the table, where its data can be ignored by queries after the column is dropped. Or, re-run your ETL process and create new data files if you drop or change the type of a column in a way that causes problems with existing data files.

```
-- Parquet table showing how dropping a column can produce unexpected results.
create table p1 (s1 string, s2 string, s3 string) stored as parquet;
```

```
insert into p1 values ('one', 'un', 'uno'), ('two', 'deux', 'dos'),
  ('three', 'trois', 'tres');
select * from p1;
+-------+-------+------+
| s1    | s2    | s3   |
+-------+-------+------+
| one   | un    | uno  |
| two   | deux  | dos  |
| three | trois | tres |
+-------+-------+------+

alter table p1 drop column s2;
-- The S3 column contains unexpected results.
-- Because S2 and S3 have compatible types, the query reads
-- values from the dropped S2, because the existing data files
-- still contain those values as the second column.
select * from p1;
+-------+-------+
| s1    | s3    |
+-------+-------+
| one   | un    |
| two   | deux  |
| three | trois |
+-------+-------+
```

```
-- Parquet table showing how dropping a column can produce conversion errors.
create table p2 (s1 string, x int, s3 string) stored as parquet;

insert into p2 values ('one', 1, 'uno'), ('two', 2, 'dos'), ('three', 3, 'tres');
select * from p2;
+-------+---+------+
| s1    | x | s3   |
+-------+---+------+
| one   | 1 | uno  |
| two   | 2 | dos  |
| three | 3 | tres |
+-------+---+------+

alter table p2 drop column x;
select * from p2;
WARNINGS:
File 'hdfs_filename' has an incompatible Parquet schema for column 'add_columns.p2.s3'.
Column type: STRING, Parquet schema:
optional int32 x [i:1 d:1 r:0]

File 'hdfs_filename' has an incompatible Parquet schema for column 'add_columns.p2.s3'.
Column type: STRING, Parquet schema:
optional int32 x [i:1 d:1 r:0]
```

In CDH 5.8 / Impala 2.6 and higher, if an Avro table is created without column definitions in the `CREATE TABLE` statement, and columns are later added through `ALTER TABLE`, the resulting table is now queryable. Missing values from the newly added columns now default to `NULL`.

To change the file format that Impala expects data to be in, for a table or partition:

Use an `ALTER TABLE ... SET FILEFORMAT` clause. You can include an optional `PARTITION (`*`col1`*`=`*`val1`*`, `*`col2`*`=`*`val2`*`, ...` clause so that the file format is changed for a specific partition rather than the entire table.

Because this operation only changes the table metadata, you must do any conversion of existing data using regular Hadoop techniques outside of Impala. Any new data created by the Impala `INSERT` statement will be in the new format. You cannot specify the delimiter for Text files; the data files must be comma-delimited.

To set the file format for a single partition, include the `PARTITION` clause. Specify all the same partitioning columns for the table, with a constant value for each, to precisely identify the single partition affected by the statement:

```
create table p1 (s string) partitioned by (month int, day int);
-- Each ADD PARTITION clause creates a subdirectory in HDFS.
alter table p1 add partition (month=1, day=1);
```

```
alter table p1 add partition (month=1, day=2);
alter table p1 add partition (month=2, day=1);
alter table p1 add partition (month=2, day=2);
-- Queries and INSERT statements will read and write files
-- in this format for this specific partition.
alter table p1 partition (month=2, day=2) set fileformat parquet;
```

To change the row format with different delimiter characters:

Use the SET ROW FORMAT DELIMITED clause to ingest data files that use a different delimiter character or a different line end character. When specifying delimiter and line end characters with the FIELDS TERMINATED BY, ESCAPED BY, and LINES TERMINATED BY clauses, you can use the following:

- A regular ASCII character surrounded by single or double quotation marks.
- An octal sequence, such as '\054' representing a comma or '\0' for ASCII null (hex 00).
- Special characters, such as:
  - '\t' for tab
  - '\n' for newline or linefeed
  - '\r' for carriage return
- An integer in the range '-127'..'128' (with quotation marks but no backslash)

  Negative values are subtracted from 256. For example, FIELDS TERMINATED BY '-2' sets the field delimiter to ASCII code 254.

For more examples of text tables, see Using Text Data Files with Impala Tables on page 636.

For the ESCAPED BY clause, choose an escape character that is not used anywhere else in the file. The character following the escape character is taken literally as part of a field value.

Surrounding field values with quotation marks does not help Impala to parse fields with embedded delimiter characters as the quotation marks are considered to be part of the column value.

If you want to use \ as the escape character, specify the clause in impala-shell as ESCAPED BY '\\'.

To add or drop partitions for a table, the table must already be partitioned (that is, created with a PARTITIONED BY clause). The partition is a physical directory in HDFS, with a name that encodes a particular column value (the **partition key**). The Impala INSERT statement already creates the partition if necessary, so the ALTER TABLE ... ADD PARTITION is primarily useful for importing data by moving or copying existing data files into the HDFS directory corresponding to a partition. (You can use the LOAD DATA statement to move files into the partition directory, or ALTER TABLE ... PARTITION (...) SET LOCATION to point a partition at a directory that already contains data files.

The DROP PARTITION clause is used to remove the HDFS directory and associated data files for a particular set of partition key values; for example, if you always analyze the last 3 months worth of data, at the beginning of each month you might drop the oldest partition that is no longer needed. Removing partitions reduces the amount of metadata associated with the table and the complexity of calculating the optimal query plan, which can simplify and speed up queries on partitioned tables, particularly join queries. Here is an example showing the ADD PARTITION and DROP PARTITION clauses.

To avoid errors while adding or dropping partitions whose existence is not certain, add the optional IF [NOT] EXISTS clause between the ADD or DROP keyword and the PARTITION keyword. That is, the entire clause becomes ADD IF NOT EXISTS PARTITION or DROP IF EXISTS PARTITION. The following example shows how partitions can be created automatically through INSERT statements, or manually through ALTER TABLE statements. The IF [NOT] EXISTS clauses let the ALTER TABLE statements succeed even if a new requested partition already exists, or a partition to be dropped does not exist.

Inserting 2 year values creates 2 partitions:

```
create table partition_t (s string) partitioned by (y int);
insert into partition_t (s,y) values ('two thousand',2000), ('nineteen ninety',1990);
```

```
show partitions partition_t;
+-------+-------+--------+------+-------------+------------------+--------+------------------+
| y     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incremental
 stats |
+-------+-------+--------+------+-------------+-------------------+--------+-------+
| 1990  | -1    | 1      | 16B  | NOT CACHED   | NOT CACHED        | TEXT   | false |
| 2000  | -1    | 1      | 13B  | NOT CACHED   | NOT CACHED        | TEXT   | false |
| Total | -1    | 2      | 29B  | 0B           |                   |        |       |
+-------+-------+--------+------+-------------+-------------------+--------+-------+
```

Without the `IF NOT EXISTS` clause, an attempt to add a new partition might fail:

```
alter table partition_t add partition (y=2000);
ERROR: AnalysisException: Partition spec already exists: (y=2000).
```

The `IF NOT EXISTS` clause makes the statement succeed whether or not there was already a partition with the specified key value:

```
alter table partition_t add if not exists partition (y=2000);
alter table partition_t add if not exists partition (y=2010);
show partitions partition_t;
+-------+-------+--------+------+-------------+------------------+--------+------------------+
| y     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incremental
 stats |
+-------+-------+--------+------+-------------+-------------------+--------+-------+
| 1990  | -1    | 1      | 16B  | NOT CACHED   | NOT CACHED        | TEXT   | false |
| 2000  | -1    | 1      | 13B  | NOT CACHED   | NOT CACHED        | TEXT   | false |
| 2010  | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT   | false |
| Total | -1    | 2      | 29B  | 0B           |                   |        |       |
+-------+-------+--------+------+-------------+-------------------+--------+-------+
```

Likewise, the `IF EXISTS` clause lets `DROP PARTITION` succeed whether or not the partition is already in the table:

```
alter table partition_t drop if exists partition (y=2000);
alter table partition_t drop if exists partition (y=1950);
show partitions partition_t;
+-------+-------+--------+------+-------------+------------------+--------+------------------+
| y     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incremental
 stats |
+-------+-------+--------+------+-------------+-------------------+--------+-------+
| 1990  | -1    | 1      | 16B  | NOT CACHED   | NOT CACHED        | TEXT   | false |
| 2010  | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT   | false |
| Total | -1    | 1      | 16B  | 0B           |                   |        |       |
+-------+-------+--------+------+-------------+-------------------+--------+-------+
```

The optional `PURGE` keyword, available in CDH 5.5 / Impala 2.3 and higher, is used with the `DROP PARTITION` clause to remove associated HDFS data files immediately rather than going through the HDFS trashcan mechanism. Use this keyword when dropping a partition if it is crucial to remove the data as quickly as possible to free up space, or if there is a problem with the trashcan, such as the trash cannot being configured or being in a different HDFS encryption zone than the data files.

```
-- Create an empty table and define the partitioning scheme.
create table part_t (x int) partitioned by (month int);
-- Create an empty partition into which you could copy data files from some other source.
alter table part_t add partition (month=1);
-- After changing the underlying data, issue a REFRESH statement to make the data visible
 in Impala.
refresh part_t;
-- Later, do the same for the next month.
alter table part_t add partition (month=2);

-- Now you no longer need the older data.
alter table part_t drop partition (month=1);
-- If the table was partitioned by month and year, you would issue a statement like:
```

```
-- alter table part_t drop partition (year=2003,month=1);
-- which would require 12 ALTER TABLE statements to remove a year's worth of data.

-- If the data files for subsequent months were in a different file format,
-- you could set a different file format for the new partition as you create it.
alter table part_t add partition (month=3) set fileformat=parquet;
```

The value specified for a partition key can be an arbitrary constant expression, without any references to columns. For example:

```
alter table time_data add partition (month=concat('Decem','ber'));
alter table sales_data add partition (zipcode = cast(9021 * 10 as string));
```



Note:

An alternative way to reorganize a table and its associated data files is to use CREATE TABLE to create a variation of the original table, then use INSERT to copy the transformed or reordered data to the new table. The advantage of ALTER TABLE is that it avoids making a duplicate copy of the data files, allowing you to reorganize huge volumes of data in a space-efficient way using familiar Hadoop techniques.

To switch a table between internal and external:

You can switch a table from internal to external, or from external to internal, by using the ALTER TABLE statement:

```
-- Switch a table from internal to external.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='TRUE');

-- Switch a table from external to internal.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='FALSE');
```

If the Kudu service is integrated with the Hive Metastore, the above operations are not supported.

Cancellation: Cannot be cancelled.

HDFS permissions:

Most ALTER TABLE clauses do not actually read or write any HDFS files, and so do not depend on specific HDFS permissions. For example, the SET FILEFORMAT clause does not actually check the file format existing data files or convert them to the new format, and the SET LOCATION clause does not require any special permissions on the new location. (Any permission-related failures would come later, when you actually query or insert into the table.)

In general, ALTER TABLE clauses that do touch HDFS files and directories require the same HDFS permissions as corresponding CREATE, INSERT, or SELECT statements. The permissions allow the user ID that the impalad daemon runs under, typically the impala user, to read or write files or directories, or (in the case of the execute bit) descend into a directory. The RENAME TO clause requires read, write, and execute permission in the source and destination database directories and in the table data directory, and read and write permission for the data files within the table. The ADD PARTITION and DROP PARTITION clauses require write and execute permissions for the associated partition directory.

Kudu considerations:

Because of the extra constraints and features of Kudu tables, such as the NOT NULL and DEFAULT attributes for columns, ALTER TABLE has specific requirements related to Kudu tables:

- In an ADD COLUMNS operation, you can specify the NULL, NOT NULL, and DEFAULT *default_value* column attributes.
- In CDH 5.12 / Impala 2.9 and higher, you can also specify the ENCODING, COMPRESSION, and BLOCK_SIZE attributes when adding a column.

- If you add a column with a `NOT NULL` attribute, it must also have a `DEFAULT` attribute, so the default value can be assigned to that column for all existing rows.
- The `DROP COLUMN` clause works the same for a Kudu table as for other kinds of tables.
- Although you can change the name of a column with the `CHANGE` clause, you cannot change the type of a column in a Kudu table.
- You cannot change the nullability of existing columns in a Kudu table.
- In CDH 5.13 / Impala 2.10, you can change the default value, encoding, compression, or block size of existing columns in a Kudu table by using the `SET` clause.
- You cannot use the `REPLACE COLUMNS` clause with a Kudu table.
- The `RENAME TO` clause for a Kudu table only affects the name stored in the metastore database that Impala uses to refer to the table. To change which underlying Kudu table is associated with an Impala table name, you must change the `TBLPROPERTIES` property of the table: `SET TBLPROPERTIES('kudu.table_name'='`*`kudu_tbl_name`*`)`. You can only change underlying Kudu tables for the external tables.

The following are some examples of using the `ADD COLUMNS` clause for a Kudu table:

```
CREATE TABLE t1 ( x INT, PRIMARY KEY (x) )
  PARTITION BY HASH (x) PARTITIONS 16
  STORED AS KUDU

ALTER TABLE t1 ADD COLUMNS (y STRING ENCODING prefix_encoding);
ALTER TABLE t1 ADD COLUMNS (z INT DEFAULT 10);
ALTER TABLE t1 ADD COLUMNS (a STRING NOT NULL DEFAULT '', t TIMESTAMP COMPRESSION
default_compression);
```

The following are some examples of modifying column defaults and storage attributes for a Kudu table:

```
create table kt (x bigint primary key, s string default 'yes', t timestamp)
  stored as kudu;

-- You can change the default value for a column, which affects any rows
-- inserted after this change is made.
alter table kt alter column s set default 'no';

-- You can remove the default value for a column, which affects any rows
-- inserted after this change is made. If the column is nullable, any
-- future inserts default to NULL for this column. If the column is marked
-- NOT NULL, any future inserts must specify a value for the column.
alter table kt alter column s drop default;

insert into kt values (1, 'foo', now());
-- Because of the DROP DEFAULT above, omitting S from the insert
-- gives it a value of NULL.
insert into kt (x, t) values (2, now());

select * from kt;
+---+------+-------------------------------+
| x | s    | t                             |
+---+------+-------------------------------+
| 2 | NULL | 2017-10-02 00:03:40.652156000 |
| 1 | foo  | 2017-10-02 00:03:04.346185000 |
+---+------+-------------------------------+

-- Other storage-related attributes can also be changed for columns.
-- These changes take effect for any newly inserted rows, or rows
-- rearranged due to compaction after deletes or updates.
alter table kt alter column s set encoding prefix_encoding;
-- The COLUMN keyword is optional in the syntax.
alter table kt alter x set block_size 2048;
alter table kt alter column t set compression zlib;
```

```
desc kt;
+-----+----------+--------+------------+---------+-------------+---------------+-------------------+-----------+
| name | type      | comment | primary_key | nullable | default_value | encoding
  | compression          | block_size |
+-----+----------+--------+------------+---------+-------------+---------------+-------------------+-----------+
| x    | bigint    |         | true        | false    |               | AUTO_ENCODING
  | DEFAULT_COMPRESSION | 2048       |
| s    | string    |         | false       | true     |               | PREFIX_ENCODING
 | DEFAULT_COMPRESSION | 0          |
| t    | timestamp |         | false       | true     |               | AUTO_ENCODING
  | ZLIB                | 0          |
+-----+----------+--------+------------+---------+-------------+---------------+-------------------+-----------+
```

Kudu tables all use an underlying partitioning mechanism. The partition syntax is different than for non-Kudu tables. You can use the `ALTER TABLE` statement to add and drop ***range partitions*** from a Kudu table. Any new range must not overlap with any existing ranges. Dropping a range removes all the associated rows from the table. See Partitioning for Kudu Tables on page 675 for details.

Related information:

Overview of Impala Tables on page 196, CREATE TABLE Statement on page 234, DROP TABLE Statement on page 268, Partitioning for Impala Tables on page 625, Internal Tables on page 197, External Tables on page 197

ALTER VIEW Statement

The `ALTER VIEW` statement changes the characteristics of a view.

Because a view is a logical construct, an alias for a query, with no physical data behind it, `ALTER VIEW` only involves changes to metadata in the metastore database, not any data files in HDFS.

To see the definition of the updated view, issue a `DESCRIBE FORMATTED` statement.

Syntax:

```
ALTER VIEW [database_name.]view_name
   [(column_name [COMMENT 'column_comment'][, ...])]
   AS select_statement;

ALTER VIEW [database_name.]view_name
   RENAME TO [database_name.]view_name;

ALTER VIEW [database_name.]view_name SET OWNER USER user_name;
ALTER VIEW [database_name.]view_name SET OWNER ROLE role_name;
```

- The AS clause associates the view with a different query.

  An optional list of column names can be specified with or without the column-level comments.

  For example:

```
ALTER VIEW v1 AS SELECT x, UPPER(s) s FROM t2;
ALTER VIEW v1 (c1, c2) AS SELECT x, UPPER(s) s FROM t2;
ALTER VIEW v7 (c1 COMMENT 'Comment for c1', c2) AS SELECT t1.c1, t1.c2 FROM t1;
```

- The `RENAME TO` clause changes the name of the view, moves the view to a different database, or both.

  For example:

```
ALTER VIEW db1.v1 RENAME TO db2.v2; -- Move the view to a different database with a new
 name.
ALTER VIEW db1.v1 RENAME TO db1.v2;  -- Rename the view in the same database.
ALTER VIEW db1.v1 RENAME TO db2.v1; -- Move the view to a difference database with the
 same view name.
```

- The SET OWNER clause transfers the ownership of the view from the current owner to another user or a role.

  The view owner is originally set to the user who creates the view. When object ownership is enabled in Sentry, an owner of a view can have the ALL with GRANT or ALL without GRANT privilege. The term OWNER is used to differentiate between the ALL privilege that is explicitly granted via the GRANT statement and a privilege that is implicitly granted by the CREATE VIEW statement.

Statement type: DDL

If you connect to different Impala nodes within an impala-shell session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts. See http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/sg_redaction.html for details.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Overview of Impala Views on page 199, CREATE VIEW Statement on page 248, DROP VIEW Statement on page 270

COMMENT Statement

The COMMENT statement adds, changes, or removes a comment about a database, a table, or a column.

You can alternatively use the CREATE and ALTER statements to add comments to the objects.

You can view the comment on a database, a table, or a column using the SHOW or DESCRIBE statement.

Syntax:

```
COMMENT ON DATABASE db_name IS {'comment' | NULL}
```

```
COMMENT ON TABLE [db_name.]table_name IS {'comment' | NULL}
```

```
COMMENT ON COLUMN [db_name.]table_name.column_name IS {'comment' | NULL}
```

Parameters:

- *db_name*: Specify the database name if not for the current database.
- NULL: If given for the comment, removes the existing comment.
- The *comment* string can be up to 256 characters long.

Privileges required:

To add a comment, the ALTER privilege on the object is required.

To view a comment, the SELECT, INSERT, or REFRESH on the object is required.

Usage notes:

Added in: CDH 6.1

COMPUTE STATS Statement

The COMPUTE STATS statement gathers information about volume and distribution of data in a table and all associated columns and partitions. The information is stored in the metastore database, and used by Impala to help optimize

queries. For example, if Impala can determine that a table is large or small, or has many or few distinct values it can organize and parallelize the work appropriately for a join query or insert operation. For details about the kinds of information gathered by this statement, see Table and Column Statistics on page 575.

Syntax:

```
COMPUTE STATS [db_name.]table_name  [ ( column_list ) ] [TABLESAMPLE SYSTEM(percentage)
  [REPEATABLE(seed)]]

column_list ::= column_name [ , column_name, ... ]

COMPUTE INCREMENTAL STATS [db_name.]table_name [PARTITION (partition_spec)]

partition_spec ::= partition_col=constant_value

partition_spec ::= simple_partition_spec | complex_partition_spec

simple_partition_spec ::= partition_col=constant_value

complex_partition_spec ::= comparison_expression_on_partition_col
```

The `PARTITION` clause is only allowed in combination with the `INCREMENTAL` clause. It is optional for `COMPUTE INCREMENTAL STATS`, and required for `DROP INCREMENTAL STATS`. Whenever you specify partitions through the `PARTITION (partition_spec)` clause in a `COMPUTE INCREMENTAL STATS` or `DROP INCREMENTAL STATS` statement, you must include all the partitioning columns in the specification, and specify constant values for all the partition key columns.

Usage notes:

Originally, Impala relied on users to run the Hive `ANALYZE TABLE` statement, but that method of gathering statistics proved unreliable and difficult to use. The Impala `COMPUTE STATS` statement was built to improve the reliability and user-friendliness of this operation. `COMPUTE STATS` does not require any setup steps or special configuration. You only run a single Impala `COMPUTE STATS` statement to gather both table and column statistics, rather than separate Hive `ANALYZE TABLE` statements for each kind of statistics.

For non-incremental `COMPUTE STATS` statement, the columns for which statistics are computed can be specified with an optional comma-separate list of columns.

If no column list is given, the `COMPUTE STATS` statement computes column-level statistics for all columns of the table. This adds potentially unneeded work for columns whose stats are not needed by queries. It can be especially costly for very wide tables and unneeded large string fields.

`COMPUTE STATS` returns an error when a specified column cannot be analyzed, such as when the column does not exist, the column is of an unsupported type for COMPUTE STATS, e.g. colums of complex types, or the column is a partitioning column.

If an empty column list is given, no column is analyzed by `COMPUTE STATS`.

In CDH 5.15 / Impala 2.12 and higher, an optional `TABLESAMPLE` clause immediately after a table reference specifies that the `COMPUTE STATS` operation only processes a specified percentage of the table data. For tables that are so large that a full `COMPUTE STATS` operation is impractical, you can use `COMPUTE STATS` with a `TABLESAMPLE` clause to extrapolate statistics from a sample of the table data. See Table and Column Statistics about the experimental stats extrapolation and sampling features.

The `COMPUTE INCREMENTAL STATS` variation is a shortcut for partitioned tables that works on a subset of partitions rather than the entire table. The incremental nature makes it suitable for large tables with many partitions, where a full `COMPUTE STATS` operation takes too long to be practical each time a partition is added or dropped. See Generating Table and Column Statistics on page 578 for full usage details.



Important:

For a particular table, use either COMPUTE STATS or COMPUTE INCREMENTAL STATS. The two kinds of stats do not interoperate with each other at the table level. Without dropping the stats, if you run COMPUTE INCREMENTAL STATS it will overwrite the full compute stats or if you run COMPUTE STATS it will drop all incremental stats for consistency.

When you run COMPUTE INCREMENTAL STATS on a table for the first time, the statistics are computed again from scratch regardless of whether the table already has statistics. Therefore, expect a one-time resource-intensive operation for scanning the entire table when running COMPUTE INCREMENTAL STATS for the first time on a given table.

In Impala 3.0 and lower, approximately 400 bytes of metadata per column per partition are needed for caching. Tables with a big number of partitions and many columns can add up to a significant memory overhead as the metadata must be cached on the catalogd host and on every impalad host that is eligible to be a coordinator. If this metadata for all tables exceeds 2 GB, you might experience service downtime. In Impala 3.1 and higher, the issue was alleviated with an improved handling of incremental stats.

COMPUTE INCREMENTAL STATS only applies to partitioned tables. If you use the INCREMENTAL clause for an unpartitioned table, Impala automatically uses the original COMPUTE STATS statement. Such tables display false under the Incremental stats column of the SHOW TABLE STATS output.



Note:

Because many of the most performance-critical and resource-intensive operations rely on table and column statistics to construct accurate and efficient plans, COMPUTE STATS is an important step at the end of your ETL process. Run COMPUTE STATS on all tables as your first step during performance tuning for slow queries, or troubleshooting for out-of-memory conditions:

- Accurate statistics help Impala construct an efficient query plan for join queries, improving performance and reducing memory usage.
- Accurate statistics help Impala distribute the work effectively for insert operations into Parquet tables, improving performance and reducing memory usage.
- Accurate statistics help Impala estimate the memory required for each query, which is important when you use resource management features, such as admission control and the YARN resource management framework. The statistics help Impala to achieve high concurrency, full utilization of available memory, and avoid contention with workloads from other Hadoop components.
- In CDH 5.10 / Impala 2.8 and higher, when you run the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement against a Parquet table, Impala automatically applies the query option setting MT_DOP=4 to increase the amount of intra-node parallelism during this CPU-intensive operation. See MT_DOP Query Option on page 345 for details about what this query option does and how to use it with CPU-intensive SELECT statements.

Computing stats for groups of partitions:

In CDH 5.10 / Impala 2.8 and higher, you can run COMPUTE INCREMENTAL STATS on multiple partitions, instead of the entire table or one partition at a time. You include comparison operators other than = in the PARTITION clause, and the COMPUTE INCREMENTAL STATS statement applies to all partitions that match the comparison expression.

For example, the INT_PARTITIONS table contains 4 partitions. The following COMPUTE INCREMENTAL STATS statements affect some but not all partitions, as indicated by the Updated *n* partition(s) messages. The partitions that are affected depend on values in the partition key column X that match the comparison expression in the PARTITION clause.

```
show partitions int_partitions;
```

```
+-------+-------+--------+------+--------------+-------------------+---------+...
| x     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format  |...
+-------+-------+--------+------+--------------+-------------------+---------+...
| 99    | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | PARQUET |...
| 120   | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT    |...
| 150   | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT    |...
| 200   | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT    |...
| Total | -1    | 0      | 0B   | 0B           |                   |         |...
+-------+-------+--------+------+--------------+-------------------+---------+...

compute incremental stats int_partitions partition (x < 100);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x in (100, 150, 200));
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 2 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x between 100 and 175);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 2 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x in (100, 150, 200) or x < 100);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 3 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x != 150);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 3 partition(s) and 1 column(s). |
+-----------------------------------------+
```

Complex type considerations:

Currently, the statistics created by the COMPUTE STATS statement do not include information about complex type columns. The column stats metrics for complex columns are always shown as -1. For queries involving complex type columns, Impala uses heuristics to estimate the data distribution within such columns.

HBase considerations:

COMPUTE STATS works for HBase tables also. The statistics gathered for HBase tables are somewhat different than for HDFS-backed tables, but that metadata is still used for optimization when HBase tables are involved in join queries.

Amazon S3 considerations:

COMPUTE STATS also works for tables where data resides in the Amazon Simple Storage Service (S3). See Using Impala with the Amazon S3 Filesystem on page 692 for details.

Performance considerations:

The statistics collected by COMPUTE STATS are used to optimize join queries INSERT operations into Parquet tables, and other resource-intensive kinds of SQL statements. See Table and Column Statistics on page 575 for details.

For large tables, the COMPUTE STATS statement itself might take a long time and you might need to tune its performance. The COMPUTE STATS statement does not work with the EXPLAIN statement, or the SUMMARY command in impala-shell. You can use the PROFILE statement in impala-shell to examine timing information for the

statement as a whole. If a basic COMPUTE STATS statement takes a long time for a partitioned table, consider switching to the COMPUTE INCREMENTAL STATS syntax so that only newly added partitions are analyzed each time.

Examples:

This example shows two tables, T1 and T2, with a small number distinct values linked by a parent-child relationship between T1.ID and T2.PARENT. T1 is tiny, while T2 has approximately 100K rows. Initially, the statistics includes physical measurements such as the number of files, the total size, and size measurements for fixed-length columns such as with the INT type. Unknown values are represented by -1. After running COMPUTE STATS for each table, much more information is available through the SHOW STATS statements. If you were running a join query involving both of these tables, you would need statistics for both tables to get the most effective optimization for the query.

```
[localhost:21000] > show table stats t1;
Query: show table stats t1
+-------+--------+------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+------+--------+
| -1    | 1      | 33B  | TEXT   |
+-------+--------+------+--------+
Returned 1 row(s) in 0.02s
[localhost:21000] > show table stats t2;
Query: show table stats t2
+-------+--------+----------+--------+
| #Rows | #Files | Size     | Format |
+-------+--------+----------+--------+
| -1    | 28     | 960.00KB | TEXT   |
+-------+--------+----------+--------+
Returned 1 row(s) in 0.01s
[localhost:21000] > show column stats t1;
Query: show column stats t1
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| id     | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 1.71s
[localhost:21000] > show column stats t2;
Query: show column stats t2
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| parent | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
[localhost:21000] > compute stats t1;
Query: compute stats t1
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 5.30s
[localhost:21000] > show table stats t1;
Query: show table stats t1
+-------+--------+------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+------+--------+
| 3     | 1      | 33B  | TEXT   |
+-------+--------+------+--------+
Returned 1 row(s) in 0.01s
[localhost:21000] > show column stats t1;
Query: show column stats t1
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| id     | INT    | 3                | -1     | 4        | 4        |
| s      | STRING | 3                | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.02s
```

```
[localhost:21000] > compute stats t2;
Query: compute stats t2
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 5.70s
[localhost:21000] > show table stats t2;
Query: show table stats t2
+-------+--------+----------+--------+
| #Rows | #Files | Size     | Format |
+-------+--------+----------+--------+
| 98304 | 1      | 960.00KB | TEXT   |
+-------+--------+----------+--------+
Returned 1 row(s) in 0.03s
[localhost:21000] > show column stats t2;
Query: show column stats t2
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| parent | INT    | 3                | -1     | 4        | 4        |
| s      | STRING | 6                | -1     | 14       | 9.3      |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
```

The following example shows how to use the `INCREMENTAL` clause, available in Impala 2.1.0 and higher. The `COMPUTE INCREMENTAL STATS` syntax lets you collect statistics for newly added or changed partitions, without rescanning the entire table.

```
-- Initially the table has no incremental stats, as indicated
-- 'false' under Incremental stats.
show table stats item_partitioned;
+------------+-------+--------+----------+--------------+---------+------------------
| i_category | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+------------+-------+--------+----------+--------------+---------+------------------
| Books      | -1    | 1      | 223.74KB | NOT CACHED   | PARQUET | false
| Children   | -1    | 1      | 230.05KB | NOT CACHED   | PARQUET | false
| Electronics| -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home       | -1    | 1      | 232.56KB | NOT CACHED   | PARQUET | false
| Jewelry    | -1    | 1      | 223.72KB | NOT CACHED   | PARQUET | false
| Men        | -1    | 1      | 231.25KB | NOT CACHED   | PARQUET | false
| Music      | -1    | 1      | 237.90KB | NOT CACHED   | PARQUET | false
| Shoes      | -1    | 1      | 234.90KB | NOT CACHED   | PARQUET | false
| Sports     | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women      | -1    | 1      | 226.27KB | NOT CACHED   | PARQUET | false
| Total      | -1    | 10     | 2.25MB   | 0B           |         |
+------------+-------+--------+----------+--------------+---------+------------------

-- After the first COMPUTE INCREMENTAL STATS,
-- all partitions have stats. The first
-- COMPUTE INCREMENTAL STATS scans the whole
-- table, discarding any previous stats from
-- a traditional COMPUTE STATS statement.
compute incremental stats item_partitioned;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 10 partition(s) and 21 column(s). |
+-----------------------------------------+
show table stats item_partitioned;
+------------+-------+--------+----------+--------------+---------+------------------
| i_category | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+------------+-------+--------+----------+--------------+---------+------------------
| Books      | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children   | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics| 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home       | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry    | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men        | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music      | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes      | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
```

```
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------

-- Add a new partition...
alter table item_partitioned add partition (i_category='Camping');
-- Add or replace files in HDFS outside of Impala,
-- rendering the stats for a partition obsolete.
!import_data_into_sports_partition.sh
refresh item_partitioned;
drop incremental stats item_partitioned partition (i_category='Sports');
-- Now some partitions have incremental stats
-- and some do not.
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Camping     | -1    | 1      | 408.02KB | NOT CACHED   | PARQUET | false
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 11     | 2.65MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------

-- After another COMPUTE INCREMENTAL STATS,
-- all partitions have incremental stats, and only the 2
-- partitions without incremental stats were scanned.
compute incremental stats item_partitioned;
+------------------------------------------+
| summary                                  |
+------------------------------------------+
| Updated 2 partition(s) and 21 column(s). |
+------------------------------------------+
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Camping     | 5328  | 1      | 408.02KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 11     | 2.65MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------
```

File format considerations:

The COMPUTE STATS statement works with tables created with any of the file formats supported by Impala. See How Impala Works with Hadoop File Formats on page 634 for details about working with the different file formats. The following considerations apply to COMPUTE STATS depending on the file format of the table.

The COMPUTE STATS statement works with text tables with no restrictions. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with Parquet tables. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with Avro tables without restriction in CDH 5.4 / Impala 2.2 and higher. In earlier releases, COMPUTE STATS worked only for Avro tables created through Hive, and required the CREATE TABLE statement to use SQL-style column names and types rather than an Avro-style schema specification.

The COMPUTE STATS statement works with RCFile tables with no restrictions. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with SequenceFile tables with no restrictions. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with partitioned tables, whether all the partitions use the same file format, or some partitions are defined through ALTER TABLE to use different file formats.

Statement type: DDL

Cancellation: Certain multi-stage statements (CREATE TABLE AS SELECT and COMPUTE STATS) can be cancelled during some stages, when running INSERT or SELECT operations internally. To cancel this statement, use Ctrl-C from the impala-shell interpreter, the **Cancel** button from the **Watch** page in Hue, **Actions > Cancel** from the **Queries** list in Cloudera Manager, or **Cancel** from the list of in-flight queries (for a particular node) on the **Queries** tab in the Impala web UI (port 25000).

Restrictions:



Note: Prior to Impala 1.4.0, COMPUTE STATS counted the number of NULL values in each column and recorded that figure in the metastore database. Because Impala does not currently use the NULL count during query planning, Impala 1.4.0 and higher speeds up the COMPUTE STATS statement by skipping this NULL counting.

Internal details:

Behind the scenes, the COMPUTE STATS statement executes two statements: one to count the rows of each partition in the table (or the entire table if unpartitioned) through the COUNT(*) function, and another to count the approximate number of distinct values in each column through the NDV() function. You might see these queries in your monitoring and diagnostic displays. The same factors that affect the performance, scalability, and execution of other queries (such as parallel execution, memory usage, admission control, and timeouts) also apply to the queries run by the COMPUTE STATS statement.

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have read permission for all affected files in the source directory: all files in the case of an unpartitioned table or a partitioned table in the case of COMPUTE STATS; or all the files in partitions without incremental stats in the case of COMPUTE INCREMENTAL STATS. It must also have read and execute permissions for all relevant directories holding the data files. (Essentially, COMPUTE STATS requires the same permissions as the underlying SELECT queries it runs against the table.)

Kudu considerations:

The COMPUTE STATS statement applies to Kudu tables. Impala does not compute the number of rows for each partition for Kudu tables. Therefore, you do not need to re-run the operation when you see -1 in the # Rows column of the output from SHOW TABLE STATS. That column always shows -1 for all Kudu tables.

Related information:

DROP STATS Statement on page 265, SHOW TABLE STATS Statement on page 373, SHOW COLUMN STATS Statement on page 375, Table and Column Statistics on page 575

CREATE DATABASE Statement

Creates a new database.

In Impala, a database is both:

- A logical construct for grouping together related tables, views, and functions within their own namespace. You might use a separate database for each application, set of related tables, or round of experimentation.

- A physical construct represented by a directory tree in HDFS. Tables (internal tables), partitions, and data files are all located under this directory. You can perform HDFS-level operations such as backing it up and measuring space usage, or remove it with a DROP DATABASE statement.

Syntax:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name[COMMENT 'database_comment']
  [LOCATION hdfs_path];
```

Statement type: DDL

Usage notes:

A database is physically represented as a directory in HDFS, with a filename extension .db, under the main Impala data directory. If the associated HDFS directory does not exist, it is created for you. All databases and their associated directories are top-level objects, with no physical or logical nesting.

After creating a database, to make it the current database within an impala-shell session, use the USE statement. You can refer to tables in the current database without prepending any qualifier to their names.

When you first connect to Impala through impala-shell, the database you start in (before issuing any CREATE DATABASE or USE statements) is named default.

Impala includes another predefined database, _impala_builtins, that serves as the location for the built-in functions. To see the built-in functions, use a statement like the following:

```
show functions in _impala_builtins;
show functions in _impala_builtins like '*substring*';
```

After creating a database, your impala-shell session or another impala-shell connected to the same node can immediately access that database. To access the database through the Impala daemon on a different node, issue the INVALIDATE METADATA statement first while connected to that other node.

Setting the LOCATION attribute for a new database is a way to work with sets of files in an HDFS directory structure outside the default Impala data directory, as opposed to setting the LOCATION attribute for each individual table.

If you connect to different Impala nodes within an impala-shell session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.

Hive considerations:

When you create a database in Impala, the database can also be used by Hive. When you create a database in Hive, issue an INVALIDATE METADATA statement in Impala to make Impala permanently aware of the new database.

The SHOW DATABASES statement lists all databases, or the databases whose name matches a wildcard pattern. In CDH 5.7 / Impala 2.5 and higher, the SHOW DATABASES output includes a second column that displays the associated comment, if any, for each database.

Amazon S3 considerations:

To specify that any tables created within a database reside on the Amazon S3 system, you can include an s3a:// prefix on the LOCATION attribute. In CDH 5.8 / Impala 2.6 and higher, Impala automatically creates any required folders as the databases, tables, and partitions are created, and removes them when they are dropped.

In CDH 5.8 / Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to CDH 5.8 / Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed. See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have write permission for the parent HDFS directory under which the database is located.

Examples:

```
create database first_db;
use first_db;
create table t1 (x int);

create database second_db;
use second_db;
-- Each database has its own namespace for tables.
-- You can reuse the same table names in each database.
create table t1 (s string);

create database temp;

-- You can either USE a database after creating it,
-- or qualify all references to the table name with the name of the database.
-- Here, tables T2 and T3 are both created in the TEMP database.

create table temp.t2 (x int, y int);
use database temp;
create table t3 (s string);

-- You cannot drop a database while it is selected by the USE statement.
drop database temp;
ERROR: AnalysisException: Cannot drop current default database: temp

-- The always-available database 'default' is a convenient one to USE
-- before dropping a database you created.
use default;

-- Before dropping a database, first drop all the tables inside it,
-- or in CDH 5.5 / Impala 2.3 and higher use the CASCADE clause.
drop database temp;
ERROR: ImpalaRuntimeException: Error making 'dropDatabase' RPC to Hive Metastore:
CAUSED BY: InvalidOperationException: Database temp is not empty
show tables in temp;
+------+
| name |
+------+
| t3   |
+------+

-- CDH 5.5 / Impala 2.3 and higher:
drop database temp cascade;

-- Earlier releases:
drop table temp.t3;
drop database temp;
```

Related information:

Overview of Impala Databases on page 193, DROP DATABASE Statement on page 262, USE Statement on page 385, SHOW DATABASES on page 368, Overview of Impala Tables on page 196

CREATE FUNCTION Statement

Creates a user-defined function (UDF), which you can use to implement custom logic during SELECT or INSERT operations.

Syntax:

The syntax is different depending on whether you create a scalar UDF, which is called once for each row and implemented by a single function, or a user-defined aggregate function (UDA), which is implemented by multiple functions that compute intermediate results across sets of rows.

In CDH 5.7 / Impala 2.5 and higher, the syntax is also different for creating or dropping scalar Java-based UDFs. The statements for Java UDFs use a new syntax, without any argument types or return type specified. Java-based UDFs

created using the new syntax persist across restarts of the Impala catalog server, and can be shared transparently between Impala and Hive.

To create a persistent scalar C++ UDF with `CREATE FUNCTION`:

```
CREATE FUNCTION [IF NOT EXISTS] [db_name.]function_name([arg_type[, arg_type...]])
  RETURNS return_type
  LOCATION 'hdfs_path_to_dot_so'
  SYMBOL='symbol_name'
```

To create a persistent Java UDF with `CREATE FUNCTION`:

```
CREATE FUNCTION [IF NOT EXISTS] [db_name.]function_name
  LOCATION 'hdfs_path_to_jar'
  SYMBOL='class_name'
```

To create a persistent UDA, which must be written in C++, issue a `CREATE AGGREGATE FUNCTION` statement:

```
CREATE [AGGREGATE] FUNCTION [IF NOT EXISTS] [db_name.]function_name([arg_type[,
arg_type...])
  RETURNS return_type
  [INTERMEDIATE type_spec]
  LOCATION 'hdfs_path'
  [INIT_FN='function]
  UPDATE_FN='function
  MERGE_FN='function
  [PREPARE_FN='function]
  [CLOSEFN='function]
  [SERIALIZE_FN='function]
  [FINALIZE_FN='function]
```

Statement type: DDL

Varargs notation:



Note:

Variable-length argument lists are supported for C++ UDFs, but currently not for Java UDFs.

If the underlying implementation of your function accepts a variable number of arguments:

- The variable arguments must go last in the argument list.
- The variable arguments must all be of the same type.
- You must include at least one instance of the variable arguments in every function call invoked from SQL.
- You designate the variable portion of the argument list in the `CREATE FUNCTION` statement by including `...` immediately after the type name of the first variable argument. For example, to create a function that accepts an `INT` argument, followed by a `BOOLEAN`, followed by one or more `STRING` arguments, your `CREATE FUNCTION` statement would look like:

```
CREATE FUNCTION func_name (INT, BOOLEAN, STRING ...)
  RETURNS type LOCATION 'path' SYMBOL='entry_point';
```

See Variable-Length Argument Lists on page 531 for how to code a C++ UDF to accept variable-length argument lists.

Scalar and aggregate functions:

The simplest kind of user-defined function returns a single scalar value each time it is called, typically once for each row in the result set. This general kind of function is what is usually meant by UDF. User-defined aggregate functions (UDAs) are a specialized kind of UDF that produce a single value based on the contents of multiple rows. You usually use UDAs in combination with a `GROUP BY` clause to condense a large result set into a smaller one, or even a single row summarizing column values across an entire table.

You create UDAs by using the CREATE AGGREGATE FUNCTION syntax. The clauses INIT_FN, UPDATE_FN, MERGE_FN, SERIALIZE_FN, FINALIZE_FN, and INTERMEDIATE only apply when you create a UDA rather than a scalar UDF.

The *_FN clauses specify functions to call at different phases of function processing.

- **Initialize:** The function you specify with the INIT_FN clause does any initial setup, such as initializing member variables in internal data structures. This function is often a stub for simple UDAs. You can omit this clause and a default (no-op) function will be used.
- **Update:** The function you specify with the UPDATE_FN clause is called once for each row in the original result set, that is, before any GROUP BY clause is applied. A separate instance of the function is called for each different value returned by the GROUP BY clause. The final argument passed to this function is a pointer, to which you write an updated value based on its original value and the value of the first argument.
- **Merge:** The function you specify with the MERGE_FN clause is called an arbitrary number of times, to combine intermediate values produced by different nodes or different threads as Impala reads and processes data files in parallel. The final argument passed to this function is a pointer, to which you write an updated value based on its original value and the value of the first argument.
- **Serialize:** The function you specify with the SERIALIZE_FN clause frees memory allocated to intermediate results. It is required if any memory was allocated by the Allocate function in the Init, Update, or Merge functions, or if the intermediate type contains any pointers. See the UDA code samples for details.
- **Finalize:** The function you specify with the FINALIZE_FN clause does any required teardown for resources acquired by your UDF, such as freeing memory, closing file handles if you explicitly opened any files, and so on. This function is often a stub for simple UDAs. You can omit this clause and a default (no-op) function will be used. It is required in UDAs where the final return type is different than the intermediate type. or if any memory was allocated by the Allocate function in the Init, Update, or Merge functions. See the UDA code samples for details.

If you use a consistent naming convention for each of the underlying functions, Impala can automatically determine the names based on the first such clause, so the others are optional.

For end-to-end examples of UDAs, see User-Defined Functions (UDFs) on page 525.

Complex type considerations:

Currently, Impala UDFs cannot accept arguments or return values of the Impala complex types (STRUCT, ARRAY, or MAP).

Usage notes:

- When authorization is enabled, the CREATE FUNCTION statement requires:
  - The CREATE privilege on the database.
  - The ALL privilege on two URIs where the URIs are:
    - The JAR file on the file system. For example:

```
GRANT ALL ON URI 'file:///path_to_my.jar' TO ROLE my_role;
```

    - The JAR on HDFS. For example:

```
GRANT ALL ON URI 'hdfs:///path/to/jar' TO ROLE my_role
```

- You can write Impala UDFs in either C++ or Java. C++ UDFs are new to Impala, and are the recommended format for high performance utilizing native code. Java-based UDFs are compatible between Impala and Hive, and are most suited to reusing existing Hive UDFs. (Impala can run Java-based Hive UDFs but not Hive UDAs.)
- CDH 5.7 / Impala 2.5 introduces UDF improvements to persistence for both C++ and Java UDFs, and better compatibility between Impala and Hive for Java UDFs. See User-Defined Functions (UDFs) on page 525 for details.
- The body of the UDF is represented by a .so or .jar file, which you store in HDFS and the CREATE FUNCTION statement distributes to each Impala node.

- Impala calls the underlying code during SQL statement evaluation, as many times as needed to process all the rows from the result set. All UDFs are assumed to be deterministic, that is, to always return the same result when passed the same argument values. Impala might or might not skip some invocations of a UDF if the result value is already known from a previous call. Therefore, do not rely on the UDF being called a specific number of times, and do not return different result values based on some external factor such as the current time, a random number function, or an external data source that could be updated while an Impala query is in progress.
- The names of the function arguments in the UDF are not significant, only their number, positions, and data types.
- You can overload the same function name by creating multiple versions of the function, each with a different argument signature. For security reasons, you cannot make a UDF with the same name as any built-in function.
- In the UDF code, you represent the function return result as a `struct`. This `struct` contains 2 fields. The first field is a `boolean` representing whether the value is `NULL` or not. (When this field is `true`, the return value is interpreted as `NULL`.) The second field is the same type as the specified function return type, and holds the return value when the function returns something other than `NULL`.
- In the UDF code, you represent the function arguments as an initial pointer to a UDF context structure, followed by references to zero or more `structs`, corresponding to each of the arguments. Each `struct` has the same 2 fields as with the return value, a `boolean` field representing whether the argument is `NULL`, and a field of the appropriate type holding any non-`NULL` argument value.
- For sample code and build instructions for UDFs, see the sample UDFs in the Impala github repo.
- Because the file representing the body of the UDF is stored in HDFS, it is automatically available to all the Impala nodes. You do not need to manually copy any UDF-related files between servers.
- Because Impala currently does not have any `ALTER FUNCTION` statement, if you need to rename a function, move it to a different database, or change its signature or other properties, issue a `DROP FUNCTION` statement for the original function followed by a `CREATE FUNCTION` with the desired properties.
- Because each UDF is associated with a particular database, either issue a `USE` statement before doing any `CREATE FUNCTION` statements, or specify the name of the function as *`db_name`*`.`*`function_name`*.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the `SYNC_DDL` query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.

Compatibility:

Impala can run UDFs that were created through Hive, as long as they refer to Impala-compatible data types (not composite or nested column types). Hive can run Java-based UDFs that were created through Impala, but not Impala UDFs written in C++.

The Hive `current_user()` function cannot be called from a Java UDF through Impala.

Persistence:

In CDH 5.7 / Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new `CREATE FUNCTION` syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old `CREATE FUNCTION` syntax do not persist across restarts because they are held in the memory of the `catalogd` daemon. Until you re-create such Java UDFs using the new `CREATE FUNCTION` syntax, you must reload those Java-based UDFs by running the original `CREATE FUNCTION` statements again each time you restart the `catalogd` daemon. Prior to CDH 5.7 / Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

For additional examples of all kinds of user-defined functions, see User-Defined Functions (UDFs) on page 525.

The following example shows how to take a Java jar file and make all the functions inside one of its classes into UDFs under a single (overloaded) function name in Impala. Each `CREATE FUNCTION` or `DROP FUNCTION` statement applies to all the overloaded Java functions with the same name. This example uses the signatureless syntax for `CREATE FUNCTION` and `DROP FUNCTION`, which is available in CDH 5.7 / Impala 2.5 and higher.

At the start, the jar file is in the local filesystem. Then it is copied into HDFS, so that it is available for Impala to reference through the CREATE FUNCTION statement and queries that refer to the Impala function name.

```
$ jar -tvf udf-examples-cdh570.jar
     0 Mon Feb 22 04:06:50 PST 2016 META-INF/
   122 Mon Feb 22 04:06:48 PST 2016 META-INF/MANIFEST.MF
     0 Mon Feb 22 04:06:46 PST 2016 com/
     0 Mon Feb 22 04:06:46 PST 2016 com/cloudera/
     0 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/
  2460 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/IncompatibleUdfTest.class
   541 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/TestUdfException.class
  3438 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/JavaUdfTest.class
  5872 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/TestUdf.class
...
$ hdfs dfs -put udf-examples-cdh570.jar /user/impala/udfs
$ hdfs dfs -ls /user/impala/udfs
Found 2 items
-rw-r--r--   3 jrussell supergroup        853 2015-10-09 14:05
/user/impala/udfs/hello_world.jar
-rw-r--r--   3 jrussell supergroup       7366 2016-06-08 14:25
/user/impala/udfs/udf-examples-cdh570.jar
```

In impala-shell, the CREATE FUNCTION refers to the HDFS path of the jar file and the fully qualified class name inside the jar. Each of the functions inside the class becomes an Impala function, each one overloaded under the specified Impala function name.

```
[localhost:21000] > create function testudf location
'/user/impala/udfs/udf-examples-cdh570.jar' symbol='com.cloudera.impala.TestUdf';
[localhost:21000] > show functions;
+-------------+---------------------------------------+-------------+---------------+
| return type | signature                             | binary type | is persistent |
+-------------+---------------------------------------+-------------+---------------+
| BIGINT      | testudf(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN, BOOLEAN)    | JAVA        | true          |
| DOUBLE      | testudf(DOUBLE)                       | JAVA        | true          |
| DOUBLE      | testudf(DOUBLE, DOUBLE)               | JAVA        | true          |
| DOUBLE      | testudf(DOUBLE, DOUBLE, DOUBLE)       | JAVA        | true          |
| FLOAT       | testudf(FLOAT)                        | JAVA        | true          |
| FLOAT       | testudf(FLOAT, FLOAT)                 | JAVA        | true          |
| FLOAT       | testudf(FLOAT, FLOAT, FLOAT)          | JAVA        | true          |
| INT         | testudf(INT)                          | JAVA        | true          |
| DOUBLE      | testudf(INT, DOUBLE)                  | JAVA        | true          |
| INT         | testudf(INT, INT)                     | JAVA        | true          |
| INT         | testudf(INT, INT, INT)                | JAVA        | true          |
| SMALLINT    | testudf(SMALLINT)                     | JAVA        | true          |
| SMALLINT    | testudf(SMALLINT, SMALLINT)           | JAVA        | true          |
| SMALLINT    | testudf(SMALLINT, SMALLINT, SMALLINT) | JAVA        | true          |
| STRING      | testudf(STRING)                       | JAVA        | true          |
| STRING      | testudf(STRING, STRING)               | JAVA        | true          |
| STRING      | testudf(STRING, STRING, STRING)       | JAVA        | true          |
| TINYINT     | testudf(TINYINT)                      | JAVA        | true          |
+-------------+---------------------------------------+-------------+---------------+
```

These are all simple functions that return their single arguments, or sum, concatenate, and so on their multiple arguments. Impala determines which overloaded function to use based on the number and types of the arguments.

```
insert into bigint_x values (1), (2), (4), (3);
select testudf(x) from bigint_x;
+-----------------+
| udfs.testudf(x) |
+-----------------+
| 1               |
| 2               |
| 4               |
| 3               |
```

```
+-----------------+

insert into int_x values (1), (2), (4), (3);
select testudf(x, x+1, x*x) from int_x;
+-------------------------------+
| udfs.testudf(x, x + 1, x * x) |
+-------------------------------+
| 4                             |
| 9                             |
| 25                            |
| 16                            |
+-------------------------------+

select testudf(x) from string_x;
+-----------------+
| udfs.testudf(x) |
+-----------------+
| one             |
| two             |
| four            |
| three           |
+-----------------+
select testudf(x,x) from string_x;
+--------------------+
| udfs.testudf(x, x) |
+--------------------+
| oneone             |
| twotwo             |
| fourfour           |
| threethree         |
+--------------------+
```

The previous example used the same Impala function name as the name of the class. This example shows how the Impala function name is independent of the underlying Java class or function names. A second `CREATE FUNCTION` statement results in a set of overloaded functions all named `my_func`, to go along with the overloaded functions all named `testudf`.

```
create function my_func location '/user/impala/udfs/udf-examples-cdh570.jar'
  symbol='com.cloudera.impala.TestUdf';

show functions;
+-------------+---------------------------------------+-------------+---------------+
| return type | signature                             | binary type | is persistent |
+-------------+---------------------------------------+-------------+---------------+
| BIGINT      | my_func(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | my_func(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | my_func(BOOLEAN, BOOLEAN)             | JAVA        | true          |
...
| BIGINT      | testudf(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true          |
...
```

The corresponding `DROP FUNCTION` statement with no signature drops all the overloaded functions with that name.

```
drop function my_func;
show functions;
+-------------+---------------------------------------+-------------+---------------+
| return type | signature                             | binary type | is persistent |
+-------------+---------------------------------------+-------------+---------------+
| BIGINT      | testudf(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true          |
...
```

The signatureless `CREATE FUNCTION` syntax for Java UDFs ensures that the functions shown in this example remain available after the Impala service (specifically, the Catalog Server) are restarted.

Related information:

User-Defined Functions (UDFs) on page 525 for more background information, usage instructions, and examples for Impala UDFs; DROP FUNCTION Statement on page 263

CREATE ROLE Statement (CDH 5.2 or higher only)

The CREATE ROLE statement creates a role to which privileges can be granted. Privileges can be granted to roles, which can then be assigned to users. A user that has been assigned a role will only be able to exercise the privileges of that role. Only users that have administrative privileges can create/drop roles. By default, the `hive`, `impala` and hue users have administrative privileges in Sentry.

Syntax:

```
CREATE ROLE role_name
```

Required privileges:

Only administrative users (those with ALL privileges on the server, defined in the Sentry policy file) can use this statement.

Compatibility:

Impala makes use of any roles and privileges specified by the GRANT and REVOKE statements in Hive, and Hive makes use of any roles and privileges specified by the GRANT and REVOKE statements in Impala. The Impala GRANT and REVOKE statements for privileges do not require the ROLE keyword to be repeated before each role name, unlike the equivalent Hive statements.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Enabling Sentry Authorization for Impala on page 87, GRANT Statement (CDH 5.2 or higher only) on page 273, REVOKE Statement (CDH 5.2 or higher only) on page 293, DROP ROLE Statement (CDH 5.2 or higher only) on page 265, SHOW Statement on page 363

CREATE TABLE Statement

Creates a new table and specifies its characteristics. While creating a table, you optionally specify aspects such as:

- Whether the table is internal or external.
- The columns and associated data types.
- The columns used for physically partitioning the data.
- The file format for data files.
- The HDFS directory where the data files are located.

Syntax:

The general syntax for creating a table and specifying its columns is as follows:

Explicit column definitions:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  (col_name data_type
    [COMMENT 'col_comment']
    [, ...]
  )
  [PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
  [SORT BY ([column [, column ...]])]
  [COMMENT 'table_comment']
  [ROW FORMAT row_format]
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
```

```
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
  [CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED]
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
```

CREATE TABLE AS SELECT:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] db_name.]table_name
    [PARTITIONED BY (col_name[, ...])]
    [SORT BY ([column [, column ...]])]
    [COMMENT 'table_comment']
+   [ROW FORMAT row_format]
    [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
+   [STORED AS ctas_file_format]
    [LOCATION 'hdfs_path']
+     [CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED]
    [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  AS
    select_statement
```



Note: If creating a *partitioned* table, the partition key columns must be listed last in the SELECT columns list, in the same order as in the PARTITIONED BY clause. Otherwise, you will receive an error about a column name mismatch.

```
primitive_type:
    TINYINT
  | SMALLINT
  | INT
  | BIGINT
  | BOOLEAN
  | FLOAT
  | DOUBLE
  | DECIMAL
  | STRING
  | CHAR
  | VARCHAR
  | TIMESTAMP

complex_type:
    struct_type
  | array_type
  | map_type

struct_type: STRUCT < name : primitive_or_complex_type [COMMENT 'comment_string'], ...
 >

array_type: ARRAY < primitive_or_complex_type >

map_type: MAP < primitive_type, primitive_or_complex_type >

row_format:
  DELIMITED [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
  [LINES TERMINATED BY 'char']

file_format:
    PARQUET
  | TEXTFILE
  | AVRO
  | SEQUENCEFILE
  | RCFILE

ctas_file_format:
    PARQUET
  | TEXTFILE
```

Column definitions inferred from data file:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  LIKE PARQUET 'hdfs_path_of_parquet_file'
  [PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
  [SORT BY ([column [, column ...]])]
  [COMMENT 'table_comment']
  [ROW FORMAT row_format]
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
  [CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED]
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
data_type:
    primitive_type
  | array_type
  | map_type
  | struct_type
```

Kudu tables:

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  (col_name data_type
    [kudu_column_attribute ...]
    [COMMENT 'col_comment']
    [, ...]
    [PRIMARY KEY (col_name[, ...])]
  )
  [PARTITION BY kudu_partition_clause]
  [COMMENT 'table_comment']
  STORED AS KUDU
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
```

Kudu column attributes:

```
  PRIMARY KEY
| [NOT] NULL
| ENCODING codec
| COMPRESSION algorithm
| DEFAULT constant
| BLOCK_SIZE number
```

kudu_partition_clause:

```
kudu_partition_clause ::= [ hash_clause [, ...]] [, range_clause ]

hash_clause ::=
  HASH [ (pk_col [, ...]) ]
    PARTITIONS n

range_clause ::=
  RANGE [ (pk_col [, ...]) ]
  (
    {
      PARTITION constant_expression range_comparison_operator VALUES
range_comparison_operator constant_expression
      | PARTITION VALUE = constant_expression_or_tuple
    }
   [, ...]
  )

range_comparison_operator ::= { < | <= }
```

External Kudu tables:

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] [db_name.]table_name
  [COMMENT 'table_comment']
```

```
  STORED AS KUDU
  [TBLPROPERTIES ('kudu.table_name'='internal_kudu_name')]
```

CREATE TABLE AS SELECT for Kudu tables:

```
CREATE TABLE [IF NOT EXISTS] db_name.]table_name
  [PRIMARY KEY (col_name[, ...])]
  [PARTITION BY kudu_partition_clause]
  [COMMENT 'table_comment']
  STORED AS KUDU
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
AS
  select_statement
```

Statement type: DDL

Column definitions:

Depending on the form of the CREATE TABLE statement, the column definitions are required or not allowed.

With the CREATE TABLE AS SELECT and CREATE TABLE LIKE syntax, you do not specify the columns at all; the column names and types are derived from the source table, query, or data file.

With the basic CREATE TABLE syntax, you must list one or more columns, its name, type, and optionally a comment, in addition to any columns used as partitioning keys. There is one exception where the column list is not required: when creating an Avro table with the STORED AS AVRO clause, you can omit the list of columns and specify the same metadata as part of the TBLPROPERTIES clause.

Complex type considerations:

The Impala complex types (STRUCT, ARRAY, or MAP) are available in CDH 5.5 / Impala 2.3 and higher. Because you can nest these types (for example, to make an array of maps or a struct with an array field), these types are also sometimes referred to as nested types. See Complex Types (CDH 5.5 or higher only) on page 139 for usage details.

Impala can create tables containing complex type columns, with any supported file format. Because currently Impala can only query complex type columns in Parquet tables, creating tables with complex type columns and other file formats such as text is of limited use. For example, you might create a text table including some columns with complex types with Impala, and use Hive as part of your to ingest the nested type data and copy it to an identical Parquet table. Or you might create a partitioned table containing complex type columns using one file format, and use ALTER TABLE to change the file format of individual partitions to Parquet; Impala can then query only the Parquet-format partitions in that table.

Partitioned tables can contain complex type columns. All the partition key columns must be scalar types.

Internal and external tables (EXTERNAL and LOCATION clauses):

By default, Impala creates an “internal” table, where Impala manages the underlying data files for the table, and physically deletes the data files when you drop the table. If you specify the EXTERNAL clause, Impala treats the table as an “external” table, where the data files are typically produced outside Impala and queried from their original locations in HDFS, and Impala leaves the data files in place when you drop the table. For details about internal and external tables, see Overview of Impala Tables on page 196.

Typically, for an external table you include a LOCATION clause to specify the path to the HDFS directory where Impala reads and writes files for the table. For example, if your data pipeline produces Parquet files in the HDFS directory /user/etl/destination, you might create an external table as follows:

```
CREATE EXTERNAL TABLE external_parquet (c1 INT, c2 STRING, c3 TIMESTAMP)
  STORED AS PARQUET LOCATION '/user/etl/destination';
```

Although the EXTERNAL and LOCATION clauses are often specified together, LOCATION is optional for external tables, and you can also specify LOCATION for internal tables. The difference is all about whether Impala “takes control” of the underlying data files and moves them when you rename the table, or deletes them when you drop the table. For more about internal and external tables and how they interact with the LOCATION attribute, see Overview of Impala Tables on page 196.

Partitioned tables (PARTITIONED BY clause):

The PARTITIONED BY clause divides the data files based on the values from one or more specified columns. Impala queries can use the partition metadata to minimize the amount of data that is read from disk or transmitted across the network, particularly during join queries. For details about partitioning, see Partitioning for Impala Tables on page 625.



Note:

All Kudu tables require partitioning, which involves different syntax than non-Kudu tables. See the PARTITION BY clause, rather than PARTITIONED BY, for Kudu tables.

In CDH 5.13 / Impala 2.10 and higher, the PARTITION BY clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the PARTITION BY clause is only appropriate for small lookup tables.

Prior to CDH 5.7 / Impala 2.5, you could use a partitioned table as the source and copy data from it, but could not specify any partitioning clauses for the new table. In CDH 5.7 / Impala 2.5 and higher, you can now use the PARTITIONED BY clause with a CREATE TABLE AS SELECT statement. See the examples under the following discussion of the CREATE TABLE AS SELECT syntax variation.

Sorted tables (SORT BY clause):

The optional SORT BY clause lets you specify zero or more columns that are sorted in the data files created by each Impala INSERT or CREATE TABLE AS SELECT operation. Creating data files that are sorted is most useful for Parquet tables, where the metadata stored inside each file includes the minimum and maximum values for each column in the file. (The statistics apply to each row group within the file; for simplicity, Impala writes a single row group in each file.) Grouping data values together in relatively narrow ranges within each data file makes it possible for Impala to quickly skip over data files that do not contain value ranges indicated in the WHERE clause of a query, and can improve the effectiveness of Parquet encoding and compression.

This clause is not applicable for Kudu tables or HBase tables. Although it works for other HDFS file formats besides Parquet, the more efficient layout is most evident with Parquet tables, because each Parquet data file includes statistics about the data values in that file.

The SORT BY columns cannot include any partition key columns for a partitioned table, because those column values are not represented in the underlying data files.

Because data files can arrive in Impala tables by mechanisms that do not respect the SORT BY clause, such as LOAD DATA or ETL tools that create HDFS files, Impala does not guarantee or rely on the data being sorted. The sorting aspect is only used to create a more efficient layout for Parquet files generated by Impala, which helps to optimize the processing of those Parquet files during Impala queries. During an INSERT or CREATE TABLE AS SELECT operation, the sorting occurs when the SORT BY clause applies to the destination table for the data, regardless of whether the source table has a SORT BY clause.

For example, when creating a table intended to contain census data, you might define sort columns such as last name and state. If a data file in this table contains a narrow range of last names, for example from Smith to Smythe, Impala can quickly detect that this data file contains no matches for a WHERE clause such as WHERE last_name = 'Jones' and avoid reading the entire file.

```
CREATE TABLE census_data (last_name STRING, first_name STRING, state STRING, address
STRING)
  SORT BY (last_name, state)
  STORED AS PARQUET;
```

Likewise, if an existing table contains data without any sort order, you can reorganize the data in a more efficient way by using INSERT or CREATE TABLE AS SELECT to copy that data into a new table with a SORT BY clause:

```
CREATE TABLE sorted_census_data
  SORT BY (last_name, state)
```

```
  STORED AS PARQUET
  AS SELECT last_name, first_name, state, address
    FROM unsorted_census_data;
```

The metadata for the SORT BY clause is stored in the TBLPROPERTIES fields for the table. Other SQL engines that can interoperate with Impala tables, such as Hive and Spark SQL, do not recognize this property when inserting into a table that has a SORT BY clause.

Kudu considerations:

Because Kudu tables do not support clauses related to HDFS and S3 data files and partitioning mechanisms, the syntax associated with the STORED AS KUDU clause is shown separately in the above syntax descriptions. Kudu tables have their own syntax for CREATE TABLE, CREATE EXTERNAL TABLE, and CREATE TABLE AS SELECT. Prior to CDH 5.13 / Impala 2.10, all internal Kudu tables require a PARTITION BY clause, different than the PARTITIONED BY clause for HDFS-backed tables.

Here are some examples of creating empty Kudu tables:

```
-- Single partition. Only for CDH 5.13 / Impala 2.10 and higher.
-- Only suitable for small lookup tables.
CREATE TABLE kudu_no_partition_by_clause
  (
    id bigint PRIMARY KEY, s STRING, b BOOLEAN
  )
  STORED AS KUDU;

-- Single-column primary key.
CREATE TABLE kudu_t1 (id BIGINT PRIMARY key, s STRING, b BOOLEAN)
  PARTITION BY HASH (id) PARTITIONS 20 STORED AS KUDU;

-- Multi-column primary key.
CREATE TABLE kudu_t2 (id BIGINT, s STRING, b BOOLEAN, PRIMARY KEY (id,s))
  PARTITION BY HASH (s) PARTITIONS 30 STORED AS KUDU;

-- Meaningful primary key column is good for range partitioning.
CREATE TABLE kudu_t3 (id BIGINT, year INT, s STRING,
    b BOOLEAN, PRIMARY KEY (id,year))
  PARTITION BY HASH (id) PARTITIONS 20,
  RANGE (year) (PARTITION 1980 <= VALUES < 1990,
    PARTITION 1990 <= VALUES < 2000,
    PARTITION VALUE = 2001,
    PARTITION 2001 < VALUES)
  STORED AS KUDU;
```

Here is an example of creating an external Kudu table:

```
-- Inherits column definitions from original table.
-- For tables created through Impala, the kudu.table_name property
-- comes from DESCRIBE FORMATTED output from the original table.
CREATE EXTERNAL TABLE external_t1 STORED AS KUDU
  TBLPROPERTIES ('kudu.table_name'='kudu_tbl_created_via_api');
```

Here is an example of CREATE TABLE AS SELECT syntax for a Kudu table:

```
-- The CTAS statement defines the primary key and partitioning scheme.
-- The rest of the column definitions are derived from the select list.
CREATE TABLE ctas_t1
  PRIMARY KEY (id) PARTITION BY HASH (id) PARTITIONS 10
  STORED AS KUDU
  AS SELECT id, s FROM kudu_t1;
```

The following CREATE TABLE clauses are not supported for Kudu tables:

- PARTITIONED BY (Kudu tables use the clause PARTITION BY instead)
- LOCATION
- ROWFORMAT
- CACHED IN | UNCACHED
- WITH SERDEPROPERTIES

For more on the PRIMARY KEY clause, see Primary Key Columns for Kudu Tables on page 671 and PRIMARY KEY Attribute on page 672.

For more on creating a Kudu table with a specific replication factor, see Kudu Replication Factor on page 671.

For more on the NULL and NOT NULL attributes, see NULL | NOT NULL Attribute on page 673.

For more on the ENCODING attribute, see ENCODING Attribute on page 674.

For more on the COMPRESSION attribute, see COMPRESSION Attribute on page 675.

For more on the DEFAULT attribute, see DEFAULT Attribute on page 673.

For more on the BLOCK_SIZE attribute, see BLOCK_SIZE Attribute on page 675.

Partitioning for Kudu tables (PARTITION BY clause)

For Kudu tables, you specify logical partitioning across one or more columns using the PARTITION BY clause. In contrast to partitioning for HDFS-based tables, multiple values for a partition key column can be located in the same partition. The optional HASH clause lets you divide one or a set of partition key columns into a specified number of buckets. You can use more than one HASH clause, specifying a distinct set of partition key columns for each. The optional RANGE clause further subdivides the partitions, based on a set of comparison operations for the partition key columns.

Here are some examples of the PARTITION BY HASH syntax:

```
-- Apply hash function to 1 primary key column.
create table hash_t1 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (x) partitions 10
  stored as kudu;

-- Apply hash function to a different primary key column.
create table hash_t2 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (y) partitions 10
  stored as kudu;

-- Apply hash function to both primary key columns.
-- In this case, the total number of partitions is 10.
create table hash_t3 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (x,y) partitions 10
  stored as kudu;

-- When the column list is omitted, apply hash function to all primary key columns.
create table hash_t4 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash partitions 10
  stored as kudu;

-- Hash the X values independently from the Y values.
-- In this case, the total number of partitions is 10 x 20.
create table hash_t5 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (x) partitions 10, hash (y) partitions 20
  stored as kudu;
```

Here are some examples of the PARTITION BY RANGE syntax:

```
-- Create partitions that cover every possible value of X.
-- Ranges that span multiple values use the keyword VALUES between
-- a pair of < and <= comparisons.
create table range_t1 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (x)
    (
```

```
      partition 0 <= values <= 49, partition 50 <= values <= 100,
      partition values < 0, partition 100 < values
    )
  stored as kudu;

-- Create partitions that cover some possible values of X.
-- Values outside the covered range(s) are rejected.
-- New range partitions can be added through ALTER TABLE.
create table range_t2 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100
    )
  stored as kudu;

-- A range can also specify a single specific value, using the keyword VALUE
-- with an = comparison.
create table range_t3 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (s)
    (
      partition value = 'Yes', partition value = 'No', partition value = 'Maybe'
    )
  stored as kudu;

-- Using multiple columns in the RANGE clause and tuples inside the partition spec
-- only works for partitions specified with the VALUE= syntax.
create table range_t4 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (x,s)
    (
      partition value = (0,'zero'), partition value = (1,'one'), partition value =
(2,'two')
    )
  stored as kudu;
```

Here are some examples combining both HASH and RANGE syntax for the PARTITION BY clause:

```
-- Values from each range partition are hashed into 10 associated buckets.
-- Total number of partitions in this case is 10 x 2.
create table combined_t1 (x bigint, s string, s2 string, primary key (x, s))
  partition by hash (x) partitions 10, range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100
    )
  stored as kudu;

-- The hash partitioning and range partitioning can apply to different columns.
-- But all the columns used in either partitioning scheme must be from the primary key.
create table combined_t2 (x bigint, s string, s2 string, primary key (x, s))
  partition by hash (s) partitions 10, range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100
    )
  stored as kudu;
```

For more usage details and examples of the Kudu partitioning syntax, see Using Impala to Query Kudu Tables on page 670.

Specifying file format (STORED AS and ROW FORMAT clauses):

The STORED AS clause identifies the format of the underlying data files. Currently, Impala can query more types of file formats than it can create or insert into. Use Hive to perform any create or data load operations that are not currently available in Impala. For example, Impala can create an Avro, SequenceFile, or RCFile table but cannot insert data into it. There are also Impala-specific procedures for using compression with each kind of file format. For details about working with data files of various formats, see How Impala Works with Hadoop File Formats on page 634.



Note: In Impala 1.4.0 and higher, Impala can create Avro tables, which formerly required doing the CREATE TABLE statement in Hive. See Using the Avro File Format with Impala Tables on page 659 for details and examples.

By default (when no STORED AS clause is specified), data files in Impala tables are created as text files with Ctrl-A (hex 01) characters as the delimiter. Specify the ROW FORMAT DELIMITED clause to produce or ingest data files that use a different delimiter character such as tab or |, or a different line end character such as carriage return or newline. When specifying delimiter and line end characters with the FIELDS TERMINATED BY and LINES TERMINATED BY clauses, use '\t' for tab, '\n' for newline or linefeed, '\r' for carriage return, and \0 for ASCII nul (hex 00). For more examples of text tables, see Using Text Data Files with Impala Tables on page 636.

The ESCAPED BY clause applies both to text files that you create through an INSERT statement to an Impala TEXTFILE table, and to existing data files that you put into an Impala table directory. (You can ingest existing data files either by creating the table with CREATE EXTERNAL TABLE ... LOCATION, the LOAD DATA statement, or through an HDFS operation such as hdfs dfs -put *file hdfs_path*.) Choose an escape character that is not used anywhere else in the file, and put it in front of each instance of the delimiter character that occurs within a field value. Surrounding field values with quotation marks does not help Impala to parse fields with embedded delimiter characters; the quotation marks are considered to be part of the column value. If you want to use \ as the escape character, specify the clause in impala-shell as ESCAPED BY '\\'.



Note: The CREATE TABLE clauses FIELDS TERMINATED BY, ESCAPED BY, and LINES TERMINATED BY have special rules for the string literal used for their argument, because they all require a single character. You can use a regular character surrounded by single or double quotation marks, an octal sequence such as '\054' (representing a comma), or an integer in the range '-127'..'128' (with quotation marks but no backslash), which is interpreted as a single-byte ASCII character. Negative values are subtracted from 256; for example, FIELDS TERMINATED BY '-2' sets the field delimiter to ASCII code 254, the "Icelandic Thorn" character used as a delimiter by some data formats.

Cloning tables (LIKE clause):

To create an empty table with the same columns, comments, and other attributes as another table, use the following variation. The CREATE TABLE ... LIKE form allows a restricted set of clauses, currently only the LOCATION, COMMENT, and STORED AS clauses.

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  LIKE { [db_name.]table_name | PARQUET 'hdfs_path_of_parquet_file' }
  [COMMENT 'table_comment']
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
```



Note:

To clone the structure of a table and transfer data into it in a single operation, use the CREATE TABLE AS SELECT syntax described in the next subsection.

When you clone the structure of an existing table using the CREATE TABLE ... LIKE syntax, the new table keeps the same file format as the original one, so you only need to specify the STORED AS clause if you want to use a different file format, or when specifying a view as the original table. (Creating a table "like" a view produces a text table by default.)

Although normally Impala cannot create an HBase table directly, Impala can clone the structure of an existing HBase table with the CREATE TABLE ... LIKE syntax, preserving the file format and metadata from the original table.

There are some exceptions to the ability to use CREATE TABLE ... LIKE with an Avro table. For example, you cannot use this technique for an Avro table that is specified with an Avro schema but no columns. When in doubt, check if a CREATE TABLE ... LIKE operation works in Hive; if not, it typically will not work in Impala either.

If the original table is partitioned, the new table inherits the same partition key columns. Because the new table is initially empty, it does not inherit the actual partitions that exist in the original one. To create partitions in the new table, insert data or issue ALTER TABLE ... ADD PARTITION statements.

Prior to Impala 1.4.0, it was not possible to use the CREATE TABLE LIKE *view_name* syntax. In Impala 1.4.0 and higher, you can create a table with the same column definitions as a view using the CREATE TABLE LIKE technique. Although CREATE TABLE LIKE normally inherits the file format of the original table, a view has no underlying file format, so CREATE TABLE LIKE *view_name* produces a text table by default. To specify a different file format, include a STORED AS *file_format* clause at the end of the CREATE TABLE LIKE statement.

Because CREATE TABLE ... LIKE only manipulates table metadata, not the physical data of the table, issue INSERT INTO TABLE statements afterward to copy any data from the original table into the new one, optionally converting the data to a new file format. (For some file formats, Impala can do a CREATE TABLE ... LIKE to create the table, but Impala cannot insert data in that file format; in these cases, you must load the data in Hive. See How Impala Works with Hadoop File Formats on page 634 for details.)

CREATE TABLE AS SELECT:

The CREATE TABLE AS SELECT syntax is a shorthand notation to create a table based on column definitions from another table, and copy data from the source table to the destination table without issuing any separate INSERT statement. This idiom is so popular that it has its own acronym, "CTAS".

The following examples show how to copy data from a source table T1 to a variety of destinations tables, applying various transformations to the table properties, table layout, or the data itself as part of the operation:

```
-- Sample table to be the source of CTAS operations.
CREATE TABLE t1 (x INT, y STRING);
INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (3, 'three');

-- Clone all the columns and data from one table to another.
CREATE TABLE clone_of_t1 AS SELECT * FROM t1;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+

-- Clone the columns and data, and convert the data to a different file format.
CREATE TABLE parquet_version_of_t1 STORED AS PARQUET AS SELECT * FROM t1;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+

-- Copy only some rows to the new table.
CREATE TABLE subset_of_t1 AS SELECT * FROM t1 WHERE x >= 2;
+-------------------+
| summary           |
+-------------------+
| Inserted 2 row(s) |
+-------------------+

-- Same idea as CREATE TABLE LIKE: clone table layout but do not copy any data.
CREATE TABLE empty_clone_of_t1 AS SELECT * FROM t1 WHERE 1=0;
+-------------------+
| summary           |
+-------------------+
| Inserted 0 row(s) |
+-------------------+

-- Reorder and rename columns and transform the data.
CREATE TABLE t5 AS SELECT upper(y) AS s, x+1 AS a, 'Entirely new column' AS n FROM t1;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+
```

```
SELECT * FROM t5;
+-------+---+--------------------+
| s     | a | n                  |
+-------+---+--------------------+
| ONE   | 2 | Entirely new column |
| TWO   | 3 | Entirely new column |
| THREE | 4 | Entirely new column |
+-------+---+--------------------+
```

See SELECT Statement on page 295 for details about query syntax for the SELECT portion of a CREATE TABLE AS SELECT statement.

The newly created table inherits the column names that you select from the original table, which you can override by specifying column aliases in the query. Any column or table comments from the original table are not carried over to the new table.



Note: When using the STORED AS clause with a CREATE TABLE AS SELECT statement, the destination table must be a file format that Impala can write to: currently, text or Parquet. You cannot specify an Avro, SequenceFile, or RCFile table as the destination table for a CTAS operation.

Prior to CDH 5.7 / Impala 2.5 you could use a partitioned table as the source and copy data from it, but could not specify any partitioning clauses for the new table. In CDH 5.7 / Impala 2.5 and higher, you can now use the PARTITIONED BY clause with a CREATE TABLE AS SELECT statement. The following example demonstrates how you can copy data from an unpartitioned table in a CREATE TABLE AS SELECT operation, creating a new partitioned table in the process. The main syntax consideration is the column order in the PARTITIONED BY clause and the select list: the partition key columns must be listed last in the select list, in the same order as in the PARTITIONED BY clause. Therefore, in this case, the column order in the destination table is different from the source table. You also only specify the column names in the PARTITIONED BY clause, not the data types or column comments.

```
create table partitions_no (year smallint, month tinyint, s string);
insert into partitions_no values (2016, 1, 'January 2016'),
  (2016, 2, 'February 2016'), (2016, 3, 'March 2016');

-- Prove that the source table is not partitioned.
show partitions partitions_no;
ERROR: AnalysisException: Table is not partitioned: ctas_partition_by.partitions_no

-- Create new table with partitions based on column values from source table.
create table partitions_yes partitioned by (year, month)
  as select s, year, month from partitions_no;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+

-- Prove that the destination table is partitioned.
show partitions partitions_yes;
+-------+-------+-------+--------+------+...
| year  | month | #Rows | #Files | Size |...
+-------+-------+-------+--------+------+...
| 2016  | 1     | -1    | 1      | 13B  |...
| 2016  | 2     | -1    | 1      | 14B  |...
| 2016  | 3     | -1    | 1      | 11B  |...
| Total |       | -1    | 3      | 38B  |...
+-------+-------+-------+--------+------+...
```

The most convenient layout for partitioned tables is with all the partition key columns at the end. The CTAS PARTITIONED BY syntax requires that column order in the select list, resulting in that same column order in the destination table.

```
describe partitions_no;
+-------+----------+---------+
| name  | type     | comment |
```

```
+-------+----------+---------+
| year  | smallint |         |
| month | tinyint  |         |
| s     | string   |         |
+-------+----------+---------+

-- The CTAS operation forced us to put the partition key columns last.
-- Having those columns last works better with idioms such as SELECT *
-- for partitioned tables.
describe partitions_yes;
+-------+----------+---------+
| name  | type     | comment |
+-------+----------+---------+
| s     | string   |         |
| year  | smallint |         |
| month | tinyint  |         |
+-------+----------+---------+
```

Attempting to use a select list with the partition key columns not at the end results in an error due to a column name mismatch:

```
-- We expect this CTAS to fail because non-key column S
-- comes after key columns YEAR and MONTH in the select list.
create table partitions_maybe partitioned by (year, month)
  as select year, month, s from partitions_no;
ERROR: AnalysisException: Partition column name mismatch: year != month
```

As part of a CTAS operation, you can convert the data to any file format that Impala can write (currently, TEXTFILE and PARQUET). You cannot specify the lower-level properties of a text table, such as the delimiter.

Sorting considerations: Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored and the results are not necessarily sorted. An INSERT ... SELECT operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

CREATE TABLE LIKE PARQUET:

The variation CREATE TABLE ... LIKE PARQUET '*hdfs_path_of_parquet_file*' lets you skip the column definitions of the CREATE TABLE statement. The column names and data types are automatically configured based on the organization of the specified Parquet data file, which must already reside in HDFS. You can use a data file located outside the Impala database directories, or a file from an existing Impala Parquet table; either way, Impala only uses the column definitions from the file and does not use the HDFS location for the LOCATION attribute of the new table. (Although you can also specify the enclosing directory with the LOCATION attribute, to both use the same schema as the data file and point the Impala table at the associated directory for querying.)

The following considerations apply when you use the CREATE TABLE LIKE PARQUET technique:

- Any column comments from the original table are not preserved in the new table. Each column in the new table has a comment stating the low-level Parquet field type used to deduce the appropriate SQL column type.
- If you use a data file from a partitioned Impala table, any partition key columns from the original table are left out of the new table, because they are represented in HDFS directory names rather than stored in the data file. To preserve the partition information, repeat the same PARTITION clause as in the original CREATE TABLE statement.
- The file format of the new table defaults to text, as with other kinds of CREATE TABLE statements. To make the new table also use Parquet format, include the clause STORED AS PARQUET in the CREATE TABLE LIKE PARQUET statement.
- If the Parquet data file comes from an existing Impala table, currently, any TINYINT or SMALLINT columns are turned into INT columns in the new table. Internally, Parquet stores such values as 32-bit integers.
- When the destination table uses the Parquet file format, the CREATE TABLE AS SELECT and INSERT ... SELECT statements always create at least one data file, even if the SELECT part of the statement does not match any rows. You can use such an empty Parquet data file as a template for subsequent CREATE TABLE LIKE PARQUET statements.

For more details about creating Parquet tables, and examples of the CREATE TABLE LIKE PARQUET syntax, see Using the Parquet File Format with Impala Tables on page 643.

Visibility and Metadata (TBLPROPERTIES and WITH SERDEPROPERTIES clauses):

You can associate arbitrary items of metadata with a table by specifying the TBLPROPERTIES clause. This clause takes a comma-separated list of key-value pairs and stores those items in the metastore database. You can also change the table properties later with an ALTER TABLE statement. You can observe the table properties for different delimiter and escape characters using the DESCRIBE FORMATTED command, and change those settings for an existing table with ALTER TABLE ... SET TBLPROPERTIES.

You can also associate SerDes properties with the table by specifying key-value pairs through the WITH SERDEPROPERTIES clause. This metadata is not used by Impala, which has its own built-in serializer and deserializer for the file formats it supports. Particular property values might be needed for Hive compatibility with certain variations of file formats, particularly Avro.

Some DDL operations that interact with other Hadoop components require specifying particular values in the SERDEPROPERTIES or TBLPROPERTIES fields, such as creating an Avro table or an HBase table. (You typically create HBase tables in Hive, because they require additional clauses not currently available in Impala.)

To see the column definitions and column comments for an existing table, for example before issuing a CREATE TABLE ... LIKE or a CREATE TABLE ... AS SELECT statement, issue the statement DESCRIBE *table_name*. To see even more detail, such as the location of data files and the values for clauses such as ROW FORMAT and STORED AS, issue the statement DESCRIBE FORMATTED *table_name*. DESCRIBE FORMATTED is also needed to see any overall table comment (as opposed to individual column comments).

After creating a table, your impala-shell session or another impala-shell connected to the same node can immediately query that table. There might be a brief interval (one statestore heartbeat) before the table can be queried through a different Impala node. To make the CREATE TABLE statement return only when the table is recognized by all Impala nodes in the cluster, enable the SYNC_DDL query option.

HDFS caching (CACHED IN clause):

If you specify the CACHED IN clause, any existing or future data files in the table directory or the partition subdirectories are designated to be loaded into memory with the HDFS caching mechanism. See Using HDFS Caching with Impala (CDH 5.3 or higher only) on page 593 for details about using the HDFS caching feature.

In CDH 5.4 / Impala 2.2 and higher, the optional WITH REPLICATION clause for CREATE TABLE and ALTER TABLE lets you specify a ***replication factor***, the number of hosts on which to cache the same data blocks. When Impala processes a cached data block, where the cache replication factor is greater than 1, Impala randomly selects a host that has a cached copy of that data block. This optimization avoids excessive CPU usage on a single host when the same cached data block is processed multiple times. Cloudera recommends specifying a value greater than or equal to the HDFS block replication factor.

Column order:

If you intend to use the table to hold data files produced by some external source, specify the columns in the same order as they appear in the data files.

If you intend to insert or copy data into the table through Impala, or if you have control over the way externally produced data files are arranged, use your judgment to specify columns in the most convenient order:

- If certain columns are often NULL, specify those columns last. You might produce data files that omit these trailing columns entirely. Impala automatically fills in the NULL values if so.
- If an unpartitioned table will be used as the source for an INSERT ... SELECT operation into a partitioned table, specify last in the unpartitioned table any columns that correspond to partition key columns in the partitioned table, and in the same order as the partition key columns are declared in the partitioned table. This technique lets you use INSERT ... SELECT * when copying data to the partitioned table, rather than specifying each column name individually.
- If you specify columns in an order that you later discover is suboptimal, you can sometimes work around the problem without recreating the table. You can create a view that selects columns from the original table in a

permuted order, then do a SELECT * from the view. When inserting data into a table, you can specify a permuted order for the inserted columns to match the order in the destination table.

Hive considerations:

Impala queries can make use of metadata about the table and columns, such as the number of rows in a table or the number of different values in a column. Prior to Impala 1.2.2, to create this metadata, you issued the ANALYZE TABLE statement in Hive to gather this information, after creating the table and loading representative data into it. In Impala 1.2.2 and higher, the COMPUTE STATS statement produces these statistics within Impala, without needing to use Hive at all.

HBase considerations:



Note:

The Impala CREATE TABLE statement cannot create an HBase table, because it currently does not support the STORED BY clause needed for HBase tables. Create such tables in Hive, then query them through Impala. For information on using Impala with HBase tables, see Using Impala to Query HBase Tables on page 684.

Amazon S3 considerations:

To create a table where the data resides in the Amazon Simple Storage Service (S3), specify a s3a:// prefix LOCATION attribute pointing to the data files in S3.

In CDH 5.8 / Impala 2.6 and higher, you can use this special LOCATION syntax as part of a CREATE TABLE AS SELECT statement.

In CDH 5.8 / Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to CDH 5.8 / Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed. See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

Sorting considerations: Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored and the results are not necessarily sorted. An INSERT ... SELECT operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

HDFS considerations:

The CREATE TABLE statement for an internal table creates a directory in HDFS. The CREATE EXTERNAL TABLE statement associates the table with an existing HDFS directory, and does not create any new directory in HDFS. To locate the HDFS data directory for a table, issue a DESCRIBE FORMATTED *table* statement. To examine the contents of that HDFS directory, use an OS command such as hdfs dfs -ls hdfs://*path*, either from the OS command line or through the shell or ! commands in impala-shell.

The CREATE TABLE AS SELECT syntax creates data files under the table data directory to hold any data copied by the INSERT portion of the statement. (Even if no data is copied, Impala might create one or more empty data files.)

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have both execute and write permission for the database directory where the table is being created.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts. See http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/sg_redaction.html for details.

Cancellation: Certain multi-stage statements (`CREATE TABLE AS SELECT` and `COMPUTE STATS`) can be cancelled during some stages, when running `INSERT` or `SELECT` operations internally. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the **Cancel** button from the **Watch** page in Hue, **Actions > Cancel** from the **Queries** list in Cloudera Manager, or **Cancel** from the list of in-flight queries (for a particular node) on the **Queries** tab in the Impala web UI (port 25000).

Related information:

Overview of Impala Tables on page 196, ALTER TABLE Statement on page 205, DROP TABLE Statement on page 268, Partitioning for Impala Tables on page 625, Internal Tables on page 197, External Tables on page 197, COMPUTE STATS Statement on page 219, SYNC_DDL Query Option on page 361, SHOW TABLES Statement on page 369, SHOW CREATE TABLE Statement on page 370, DESCRIBE Statement on page 251

CREATE VIEW Statement

The `CREATE VIEW` statement lets you create a shorthand abbreviation for a more complicated query. The base query can involve joins, expressions, reordered columns, column aliases, and other SQL features that can make a query hard to understand or maintain.

Because a view is purely a logical construct (an alias for a query) with no physical data behind it, `ALTER VIEW` only involves changes to metadata in the metastore database, not any data files in HDFS.

Syntax:

```
CREATE VIEW [IF NOT EXISTS] view_name
    [(column_name [COMMENT 'column_comment'][, ...])]
    [COMMENT 'view_comment']
  AS select_statement
```

Statement type: DDL

Usage notes:

The `CREATE VIEW` statement can be useful in scenarios such as the following:

- To turn even the most lengthy and complicated SQL query into a one-liner. You can issue simple queries against the view from applications, scripts, or interactive queries in `impala-shell`. For example:

```
select * from view_name;
select * from view_name order by c1 desc limit 10;
```

  The more complicated and hard-to-read the original query, the more benefit there is to simplifying the query using a view.
- To hide the underlying table and column names, to minimize maintenance problems if those names change. In that case, you re-create the view using the new names, and all queries that use the view rather than the underlying tables keep running with no changes.
- To experiment with optimization techniques and make the optimized queries available to all applications. For example, if you find a combination of `WHERE` conditions, join order, join hints, and so on that works the best for a class of queries, you can establish a view that incorporates the best-performing techniques. Applications can then make relatively simple queries against the view, without repeating the complicated and optimized logic over and over. If you later find a better way to optimize the original query, when you re-create the view, all the applications immediately take advantage of the optimized base query.
- To simplify a whole class of related queries, especially complicated queries involving joins between multiple tables, complicated expressions in the column list, and other SQL syntax that makes the query difficult to understand and debug. For example, you might create a view that joins several tables, filters using several `WHERE` conditions, and selects several columns from the result set. Applications might issue queries against this view that only vary in their `LIMIT`, `ORDER BY`, and similar simple clauses.

For queries that require repeating complicated clauses over and over again, for example in the select list, `ORDER BY`, and `GROUP BY` clauses, you can use the `WITH` clause as an alternative to creating a view.

You can optionally specify the table-level and the column-level comments as in the `CREATE TABLE` statement.

Complex type considerations:

For tables containing complex type columns (ARRAY, STRUCT, or MAP), you typically use join queries to refer to the complex values. You can use views to hide the join notation, making such tables seem like traditional denormalized tables, and making those tables queryable by business intelligence tools that do not have built-in support for those complex types. See Accessing Complex Type Data in Flattened Form Using Views on page 159 for details.

Because you cannot directly issue SELECT *col_name* against a column of complex type, you cannot use a view or a WITH clause to "rename" a column by selecting it with a column alias.

If you connect to different Impala nodes within an impala-shell session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts. See http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/sg_redaction.html for details.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

```
-- Create a view that is exactly the same as the underlying table.
CREATE VIEW v1 AS SELECT * FROM t1;

-- Create a view that includes only certain columns from the underlying table.
CREATE VIEW v2 AS SELECT c1, c3, c7 FROM t1;

-- Create a view that filters the values from the underlying table.
CREATE VIEW v3 AS SELECT DISTINCT c1, c3, c7 FROM t1 WHERE c1 IS NOT NULL AND c5 > 0;

-- Create a view that that reorders and renames columns from the underlying table.
CREATE VIEW v4 AS SELECT c4 AS last_name, c6 AS address, c2 AS birth_date FROM t1;

-- Create a view that runs functions to convert or transform certain columns.
CREATE VIEW v5 AS SELECT c1, CAST(c3 AS STRING) c3, CONCAT(c4,c5) c5, TRIM(c6) c6,
"Constant" c8 FROM t1;

-- Create a view that hides the complexity of a view query.
CREATE VIEW v6 AS SELECT t1.c1, t2.c2 FROM t1 JOIN t2 ON t1.id = t2.id;

-- Create a view with a column comment and a table comment.
CREATE VIEW v7 (c1 COMMENT 'Comment for c1', c2) COMMENT 'Comment for v7' AS SELECT
t1.c1, t1.c2 FROM t1;
```

Related information:

Overview of Impala Views on page 199, ALTER VIEW Statement on page 218, DROP VIEW Statement on page 270

DELETE Statement (CDH 5.10 or higher only)

Deletes an arbitrary number of rows from a Kudu table. This statement only works for Impala tables that use the Kudu storage engine.

Syntax:

```
DELETE [FROM] [database_name.]table_name [ WHERE where_conditions ]

DELETE table_ref FROM [joined_table_refs] [ WHERE where_conditions ]
```

The first form evaluates rows from one table against an optional WHERE clause, and deletes all the rows that match the WHERE conditions, or all rows if WHERE is omitted.

The second form evaluates one or more join clauses, and deletes all matching rows from one of the tables. The join clauses can include non-Kudu tables, but the table from which the rows are deleted must be a Kudu table. The FROM keyword is required in this case, to separate the name of the table whose rows are being deleted from the table names of the join clauses.

Usage notes:

The conditions in the WHERE clause are the same ones allowed for the SELECT statement. See SELECT Statement on page 295 for details.

The conditions in the WHERE clause can refer to any combination of primary key columns or other columns. Referring to primary key columns in the WHERE clause is more efficient than referring to non-primary key columns.

If the WHERE clause is omitted, all rows are removed from the table.

Because Kudu currently does not enforce strong consistency during concurrent DML operations, be aware that the results after this statement finishes might be different than you intuitively expect:

- If some rows cannot be deleted because their some primary key columns are not found, due to their being deleted by a concurrent DELETE operation, the statement succeeds but returns a warning.
- A DELETE statement might also overlap with INSERT, UPDATE, or UPSERT statements running concurrently on the same table. After the statement finishes, there might be more or fewer rows than expected in the table because it is undefined whether the DELETE applies to rows that are inserted or updated while the DELETE is in progress.

The number of affected rows is reported in an `impala-shell` message and in the query profile.

Statement type: DML



Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following examples show how to delete rows from a specified table, either all rows or rows that match a WHERE clause:

```
-- Deletes all rows. The FROM keyword is optional.
DELETE FROM kudu_table;
DELETE kudu_table;

-- Deletes 0, 1, or more rows.
-- (If c1 is a single-column primary key, the statement could only
-- delete 0 or 1 rows.)
DELETE FROM kudu_table WHERE c1 = 100;

-- Deletes all rows that match all the WHERE conditions.
DELETE FROM kudu_table WHERE
  (c1 > c2 OR c3 IN ('hello','world')) AND c4 IS NOT NULL;
DELETE FROM t1 WHERE
  (c1 IN (1,2,3) AND c2 > c3) OR c4 IS NOT NULL;
DELETE FROM time_series WHERE
  year = 2016 AND month IN (11,12) AND day > 15;

-- WHERE condition with a subquery.
DELETE FROM t1 WHERE
  c5 IN (SELECT DISTINCT other_col FROM other_table);
```

```
-- Does not delete any rows, because the WHERE condition is always false.
DELETE FROM kudu_table WHERE 1 = 0;
```

The following examples show how to delete rows that are part of the result set from a join:

```
-- Remove _all_ rows from t1 that have a matching X value in t2.
DELETE t1 FROM t1 JOIN t2 ON t1.x = t2.x;

-- Remove _some_ rows from t1 that have a matching X value in t2.
DELETE t1 FROM t1 JOIN t2 ON t1.x = t2.x
  WHERE t1.y = FALSE and t2.z > 100;

-- Delete from a Kudu table based on a join with a non-Kudu table.
DELETE t1 FROM kudu_table t1 JOIN non_kudu_table t2 ON t1.x = t2.x;

-- The tables can be joined in any order as long as the Kudu table
-- is specified as the deletion target.
DELETE t2 FROM non_kudu_table t1 JOIN kudu_table t2 ON t1.x = t2.x;
```

Related information:

Using Impala to Query Kudu Tables on page 670, INSERT Statement on page 277, UPDATE Statement (CDH 5.10 or higher only) on page 383, UPSERT Statement (CDH 5.10 or higher only) on page 384

DESCRIBE Statement

The DESCRIBE statement displays metadata about a table, such as the column names and their data types. In CDH 5.5 / Impala 2.3 and higher, you can specify the name of a complex type column, which takes the form of a dotted path. The path might include multiple components in the case of a nested type definition. In CDH 5.7 / Impala 2.5 and higher, the DESCRIBE DATABASE form can display information about a database.

Syntax:

```
DESCRIBE [DATABASE] [FORMATTED|EXTENDED] object_name

object_name ::=
    [db_name.]table_name[.complex_col_name ...]
  | db_name
```

You can use the abbreviation DESC for the DESCRIBE statement.

The DESCRIBE FORMATTED variation displays additional information, in a format familiar to users of Apache Hive. The extra information includes low-level details such as whether the table is internal or external, when it was created, the file format, the location of the data in HDFS, whether the object is a table or a view, and (for views) the text of the query from the view definition.



Note: The Compressed field is not a reliable indicator of whether the table contains compressed data. It typically always shows No, because the compression settings only apply during the session that loads data and are not stored persistently with the table metadata.

Describing databases:

By default, the DESCRIBE output for a database includes the location and the comment, which can be set by the LOCATION and COMMENT clauses on the CREATE DATABASE statement.

The additional information displayed by the FORMATTED or EXTENDED keyword includes the HDFS user ID that is considered the owner of the database, and any optional database properties. The properties could be specified by the WITH DBPROPERTIES clause if the database is created using a Hive CREATE DATABASE statement. Impala currently does not set or do any special processing based on those properties.

The following examples show the variations in syntax and output for describing databases. This feature is available in CDH 5.7 / Impala 2.5 and higher.

```
describe database default;
+---------+----------------------+-----------------------+
| name    | location             | comment               |
+---------+----------------------+-----------------------+
| default | /user/hive/warehouse | Default Hive database |
+---------+----------------------+-----------------------+

describe database formatted default;
+---------+----------------------+-----------------------+
| name    | location             | comment               |
+---------+----------------------+-----------------------+
| default | /user/hive/warehouse | Default Hive database |
| Owner:  |                      |                       |
|         | public               | ROLE                  |
+---------+----------------------+-----------------------+

describe database extended default;
+---------+----------------------+-----------------------+
| name    | location             | comment               |
+---------+----------------------+-----------------------+
| default | /user/hive/warehouse | Default Hive database |
| Owner:  |                      |                       |
|         | public               | ROLE                  |
+---------+----------------------+-----------------------+
```

Describing tables:

If the DATABASE keyword is omitted, the default for the DESCRIBE statement is to refer to a table.

If you have the SELECT privilege on a subset of the table columns and no other relevant table/database/server-level privileges, DESCRIBE returns the data from the columns you have access to.

If you have the SELECT privilege on a subset of the table columns and no other relevant table/database/server-level privileges, DESCRIBE FORMATTED/EXTENDED does not return the LOCATION field. The LOCATION data is shown if you have any privilege on the table, the containing database or the server.

```
-- By default, the table is assumed to be in the current database.
describe my_table;
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| s    | string |         |
+------+--------+---------+

-- Use a fully qualified table name to specify a table in any database.
describe my_database.my_table;
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| s    | string |         |
+------+--------+---------+

-- The formatted or extended output includes additional useful information.
-- The LOCATION field is especially useful to know for DDL statements and HDFS commands
-- during ETL jobs. (The LOCATION includes a full hdfs:// URL, omitted here for
readability.)
describe formatted my_table;
+------------------------------+----------------------------------------------+----------------------+
| name                         | type                                         | comment
              |
+------------------------------+----------------------------------------------+----------------------+
| # col_name                   | data_type                                    | comment
              |
|                              | NULL                                          | NULL
```

```
                |
| x                           | int                                                | NULL
                |
| s                           | string                                             | NULL
                |
|                             | NULL                                               | NULL
                |
| # Detailed Table Information | NULL                                               | NULL
                |
| Database:                   | my_database                                        | NULL
                |
| Owner:                      | jrussell                                           | NULL
                |
| CreateTime:                 | Fri Mar 18 15:58:00 PDT 2016                       | NULL
                |
| LastAccessTime:             | UNKNOWN                                            | NULL
                |
| Protect Mode:               | None                                               | NULL
                |
| Retention:                  | 0                                                  | NULL
                |
| Location:                   | /user/hive/warehouse/my_database.db/my_table       | NULL
                |
| Table Type:                 | MANAGED_TABLE                                      | NULL
                |
| Table Parameters:           | NULL                                               | NULL
                |
|                           | transient_lastDdlTime                          | 1458341880
          |
|                             | NULL                                               | NULL
                |
| # Storage Information       | NULL                                               | NULL
                |
| SerDe Library:              | org. ... .LazySimpleSerDe                          | NULL
                |
| InputFormat:                | org.apache.hadoop.mapred.TextInputFormat           | NULL
                |
| OutputFormat:               | org. ... .HiveIgnoreKeyTextOutputFormat            | NULL
                |
| Compressed:                 | No                                                 | NULL
                |
| Num Buckets:                | 0                                                  | NULL
                |
| Bucket Columns:             | []                                                 | NULL
                |
| Sort Columns:               | []                                                 | NULL
                |
+-----------------------------+----------------------------------------------+----------------------+
```

Complex type considerations:

Because the column definitions for complex types can become long, particularly when such types are nested, the DESCRIBE statement uses special formatting for complex type columns to make the output readable.

For the ARRAY, STRUCT, and MAP types available in CDH 5.5 / Impala 2.3 and higher, the DESCRIBE output is formatted to avoid excessively long lines for multiple fields within a STRUCT, or a nested sequence of complex types.

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

For example, here is the DESCRIBE output for a table containing a single top-level column of each complex type:

```
create table t1 (x int, a array<int>, s struct<f1: string, f2: bigint>, m map<string,int>)
 stored as parquet;
```

```
describe t1;
+------+-----------------+---------+
| name | type            | comment |
+------+-----------------+---------+
| x    | int             |         |
| a    | array<int>      |         |
| s    | struct<         |         |
|      |   f1:string,    |         |
|      |   f2:bigint     |         |
|      | >               |         |
| m    | map<string,int> |         |
+------+-----------------+---------+
```

Here are examples showing how to “drill down” into the layouts of complex types, including using multi-part names to examine the definitions of nested types. The < > delimiters identify the columns with complex types; these are the columns where you can descend another level to see the parts that make up the complex type. This technique helps you to understand the multi-part names you use as table references in queries involving complex types, and the corresponding column names you refer to in the SELECT list. These tables are from the “nested TPC-H” schema, shown in detail in Sample Schema and Data for Experimenting with Impala Complex Types on page 160.

The REGION table contains an ARRAY of STRUCT elements:

- The first DESCRIBE specifies the table name, to display the definition of each top-level column.
- The second DESCRIBE specifies the name of a complex column, REGION.R_NATIONS, showing that when you include the name of an ARRAY column in a FROM clause, that table reference acts like a two-column table with columns ITEM and POS.
- The final DESCRIBE specifies the fully qualified name of the ITEM field, to display the layout of its underlying STRUCT type in table format, with the fields mapped to column names.

```
-- #1: The overall layout of the entire table.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- #2: The ARRAY column within the table.
describe region.r_nations;
+------+-------------------------+---------+
| name | type                    | comment |
+------+-------------------------+---------+
| item | struct<                 |         |
|      |   n_nationkey:smallint, |         |
|      |   n_name:string,        |         |
|      |   n_comment:string      |         |
|      | >                       |         |
| pos  | bigint                  |         |
+------+-------------------------+---------+

-- #3: The STRUCT that makes up each ARRAY element.
--     The fields of the STRUCT act like columns of a table.
describe region.r_nations.item;
+-------------+----------+---------+
| name        | type     | comment |
+-------------+----------+---------+
| n_nationkey | smallint |         |
| n_name      | string   |         |
```

```
| n_comment   | string   |         |
+-------------+----------+---------+
```

The CUSTOMER table contains an ARRAY of STRUCT elements, where one field in the STRUCT is another ARRAY of STRUCT elements:

- Again, the initial DESCRIBE specifies only the table name.
- The second DESCRIBE specifies the qualified name of the complex column, CUSTOMER.C_ORDERS, showing how an ARRAY is represented as a two-column table with columns ITEM and POS.
- The third DESCRIBE specifies the qualified name of the ITEM of the ARRAY column, to see the structure of the nested ARRAY. Again, it has has two parts, ITEM and POS. Because the ARRAY contains a STRUCT, the layout of the STRUCT is shown.
- The fourth and fifth DESCRIBE statements drill down into a STRUCT field that is itself a complex type, an ARRAY of STRUCT. The ITEM portion of the qualified name is only required when the ARRAY elements are anonymous. The fields of the STRUCT give names to any other complex types nested inside the STRUCT. Therefore, the DESCRIBE parameters CUSTOMER.C_ORDERS.ITEM.O_LINEITEMS and CUSTOMER.C_ORDERS.O_LINEITEMS are equivalent. (For brevity, leave out the ITEM portion of a qualified name when it is not required.)
- The final DESCRIBE shows the layout of the deeply nested STRUCT type. Because there are no more complex types nested inside this STRUCT, this is as far as you can drill down into the layout for this table.

```
-- #1: The overall layout of the entire table.
describe customer;
+--------------+-------------------------------------+
| name         | type                                |
+--------------+-------------------------------------+
| c_custkey    | bigint                              |
... more scalar columns ...
| c_orders     | array<struct<                       |
|              |   o_orderkey:bigint,                |
|              |   o_orderstatus:string,             |
|              |   o_totalprice:decimal(12,2),       |
|              |   o_orderdate:string,               |
|              |   o_orderpriority:string,           |
|              |   o_clerk:string,                   |
|              |   o_shippriority:int,               |
|              |   o_comment:string,                 |
|              |   o_lineitems:array<struct<         |
|              |     l_partkey:bigint,               |
|              |     l_suppkey:bigint,               |
|              |     l_linenumber:int,               |
|              |     l_quantity:decimal(12,2),       |
|              |     l_extendedprice:decimal(12,2),  |
|              |     l_discount:decimal(12,2),       |
|              |     l_tax:decimal(12,2),            |
|              |     l_returnflag:string,            |
|              |     l_linestatus:string,            |
|              |     l_shipdate:string,              |
|              |     l_commitdate:string,            |
|              |     l_receiptdate:string,           |
|              |     l_shipinstruct:string,          |
|              |     l_shipmode:string,              |
|              |     l_comment:string                |
|              |   >>                                |
|              | >>                                  |
+--------------+-------------------------------------+

-- #2: The ARRAY column within the table.
describe customer.c_orders;
+------+-------------------------------------+
| name | type                                |
+------+-------------------------------------+
| item | struct<                             |
|      |   o_orderkey:bigint,                |
```

```
|      |   o_orderstatus:string,           |
... more struct fields ...
|      |   o_lineitems:array<struct<       |
|      |     l_partkey:bigint,             |
|      |     l_suppkey:bigint,             |
... more nested struct fields ...
|      |     l_comment:string              |
|      |   >>                              |
|      | >                                 |
| pos  | bigint                            |
+------+-----------------------------------+

-- #3: The STRUCT that makes up each ARRAY element.
--     The fields of the STRUCT act like columns of a table.
describe customer.c_orders.item;
+-----------------+----------------------------------+
| name            | type                             |
+-----------------+----------------------------------+
| o_orderkey      | bigint                           |
| o_orderstatus   | string                           |
| o_totalprice    | decimal(12,2)                    |
| o_orderdate     | string                           |
| o_orderpriority | string                           |
| o_clerk         | string                           |
| o_shippriority  | int                              |
| o_comment       | string                           |
| o_lineitems     | array<struct<                    |
|                 |   l_partkey:bigint,              |
|                 |   l_suppkey:bigint,              |
... more struct fields ...
|                 |   l_comment:string               |
|                 | >>                               |
+-----------------+----------------------------------+

-- #4: The ARRAY nested inside the STRUCT elements of the first ARRAY.
describe customer.c_orders.item.o_lineitems;
+------+----------------------------------+
| name | type                             |
+------+----------------------------------+
| item | struct<                          |
|      |   l_partkey:bigint,              |
|      |   l_suppkey:bigint,              |
... more struct fields ...
|      |   l_comment:string               |
|      | >                                |
| pos  | bigint                           |
+------+----------------------------------+

-- #5: Shorter form of the previous DESCRIBE. Omits the .ITEM portion of the name
--     because O_LINEITEMS and other field names provide a way to refer to things
--     inside the ARRAY element.
describe customer.c_orders.o_lineitems;
+------+----------------------------------+
| name | type                             |
+------+----------------------------------+
| item | struct<                          |
|      |   l_partkey:bigint,              |
|      |   l_suppkey:bigint,              |
... more struct fields ...
|      |   l_comment:string               |
|      | >                                |
| pos  | bigint                           |
+------+----------------------------------+

-- #6: The STRUCT representing ARRAY elements nested inside
--     another ARRAY of STRUCTs. The lack of any complex types
--     in this output means this is as far as DESCRIBE can
--     descend into the table layout.
describe customer.c_orders.o_lineitems.item;
+-----------------+---------------+
| name            | type          |
+-----------------+---------------+
| l_partkey       | bigint        |
```

```
| l_suppkey       | bigint        |
... more scalar columns ...
| l_comment       | string        |
+-----------------+---------------+
```

Usage notes:

After the `impalad` daemons are restarted, the first query against a table can take longer than subsequent queries, because the metadata for the table is loaded before the query is processed. This one-time delay for each table can cause misleading results in benchmark tests or cause unnecessary concern. To "warm up" the Impala metadata cache, you can issue a `DESCRIBE` statement in advance for each table you intend to access later.

When you are dealing with data files stored in HDFS, sometimes it is important to know details such as the path of the data files for an Impala table, and the hostname for the namenode. You can get this information from the `DESCRIBE FORMATTED` output. You specify HDFS URIs or path specifications with statements such as `LOAD DATA` and the `LOCATION` clause of `CREATE TABLE` or `ALTER TABLE`. You might also use HDFS URIs or paths with Linux commands such as `hadoop` and `hdfs` to copy, rename, and so on, data files in HDFS.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the `SYNC_DDL` query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.

Each table can also have associated table statistics and column statistics. To see these categories of information, use the `SHOW TABLE STATS` *`table_name`* and `SHOW COLUMN STATS` *`table_name`* statements. See SHOW Statement on page 363 for details.



Important: After adding or replacing data in a table used in performance-critical queries, issue a `COMPUTE STATS` statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any `INSERT`, `LOAD DATA`, or `CREATE TABLE AS SELECT` statement in Impala, or after loading data through Hive and doing a `REFRESH` *`table_name`* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following example shows the results of both a standard `DESCRIBE` and `DESCRIBE FORMATTED` for different kinds of schema objects:

- `DESCRIBE` for a table or a view returns the name, type, and comment for each of the columns. For a view, if the column value is computed by an expression, the column name is automatically generated as `_c0`, `_c1`, and so on depending on the ordinal number of the column.
- A table created with no special format or storage clauses is designated as a `MANAGED_TABLE` (an "internal table" in Impala terminology). Its data files are stored in an HDFS directory under the default Hive data directory. By default, it uses Text data format.
- A view is designated as `VIRTUAL_VIEW` in `DESCRIBE FORMATTED` output. Some of its properties are `NULL` or blank because they are inherited from the base table. The text of the query that defines the view is part of the `DESCRIBE FORMATTED` output.
- A table with additional clauses in the `CREATE TABLE` statement has differences in `DESCRIBE FORMATTED` output. The output for `T2` includes the `EXTERNAL_TABLE` keyword because of the `CREATE EXTERNAL TABLE` syntax, and different `InputFormat` and `OutputFormat` fields to reflect the Parquet file format.

```
[localhost:21000] > create table t1 (x int, y int, s string);
Query: create table t1 (x int, y int, s string)
[localhost:21000] > describe t1;
Query: describe t1
Query finished, fetching results ...
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| y    | int    |         |
```

```
| s    | string |         |
+------+--------+---------+
Returned 3 row(s) in 0.13s
[localhost:21000] > describe formatted t1;
Query: describe formatted t1
Query finished, fetching results ...
+------------------------------+----------------------------------------------+------------+
| name                         | type                                         | comment    |
+------------------------------+----------------------------------------------+------------+
| # col_name                   | data_type                                    | comment    |
|                              | NULL                                         | NULL       |
| x                            | int                                          | None       |
| y                            | int                                          | None       |
| s                            | string                                       | None       |
|                              | NULL                                         | NULL       |
| # Detailed Table Information | NULL                                         | NULL       |
| Database:                    | describe_formatted                           | NULL       |
| Owner:                       | doc_demo                                     | NULL       |
| CreateTime:                  | Mon Jul 22 17:03:16 EDT 2013                 | NULL       |
| LastAccessTime:              | UNKNOWN                                      | NULL       |
| Protect Mode:                | None                                         | NULL       |
| Retention:                   | 0                                            | NULL       |
| Location:                    | hdfs://127.0.0.1:8020/user/hive/warehouse/   |            |
|                              |   describe_formatted.db/t1                   | NULL       |
| Table Type:                  | MANAGED_TABLE                                | NULL       |
| Table Parameters:            | NULL                                         | NULL       |
|                              | transient_lastDdlTime                        | 1374526996 |
|                              | NULL                                         | NULL       |
| # Storage Information        | NULL                                         | NULL       |
| SerDe Library:               | org.apache.hadoop.hive.serde2.lazy.          |            |
|                              |   LazySimpleSerDe                            | NULL       |
| InputFormat:                 | org.apache.hadoop.mapred.TextInputFormat     | NULL       |
| OutputFormat:                | org.apache.hadoop.hive.ql.io.                |            |
|                              |   HiveIgnoreKeyTextOutputFormat              | NULL       |
| Compressed:                  | No                                           | NULL       |
| Num Buckets:                 | 0                                            | NULL       |
| Bucket Columns:              | []                                           | NULL       |
| Sort Columns:                | []                                           | NULL       |
+------------------------------+----------------------------------------------+------------+
Returned 26 row(s) in 0.03s
[localhost:21000] > create view v1 as select x, upper(s) from t1;
Query: create view v1 as select x, upper(s) from t1
[localhost:21000] > describe v1;
```

```
Query: describe v1
Query finished, fetching results ...
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| _c1  | string |         |
+------+--------+---------+
Returned 2 row(s) in 0.10s
[localhost:21000] > describe formatted v1;
Query: describe formatted v1
Query finished, fetching results ...
+------------------------------+------------------------------+----------------------+
| name                         | type                         | comment              |
+------------------------------+------------------------------+----------------------+
| # col_name                   | data_type                    | comment              |
|                              | NULL                         | NULL                 |
| x                            | int                          | None                 |
| _c1                          | string                       | None                 |
|                              | NULL                         | NULL                 |
| # Detailed Table Information | NULL                         | NULL                 |
| Database:                    | describe_formatted           | NULL                 |
| Owner:                       | doc_demo                     | NULL                 |
| CreateTime:                  | Mon Jul 22 16:56:38 EDT 2013 | NULL                 |
| LastAccessTime:              | UNKNOWN                      | NULL                 |
| Protect Mode:                | None                         | NULL                 |
| Retention:                   | 0                            | NULL                 |
| Table Type:                  | VIRTUAL_VIEW                 | NULL                 |
| Table Parameters:            | NULL                         | NULL                 |
|                              | transient_lastDdlTime        | 1374526598           |
|                              | NULL                         | NULL                 |
| # Storage Information        | NULL                         | NULL                 |
| SerDe Library:               | null                         | NULL                 |
| InputFormat:                 | null                         | NULL                 |
| OutputFormat:                | null                         | NULL                 |
| Compressed:                  | No                           | NULL                 |
| Num Buckets:                 | 0                            | NULL                 |
| Bucket Columns:              | []                           | NULL                 |
| Sort Columns:                | []                           | NULL                 |
|                              | NULL                         | NULL                 |
| # View Information           | NULL                         | NULL                 |
| View Original Text:          | SELECT x, upper(s) FROM t1   | NULL                 |
| View Expanded Text:          | SELECT x, upper(s) FROM t1   | NULL                 |
+------------------------------+------------------------------+----------------------+
Returned 28 row(s) in 0.03s
[localhost:21000] > create external table t2 (x int, y int, s string) stored as parquet
 location '/user/doc_demo/sample_data';
[localhost:21000] > describe formatted t2;
Query: describe formatted t2
Query finished, fetching results ...
+------------------------------+-------------------------------------------------------+-----------+
| name                         | type                                                  |
comment    |
+------------------------------+-------------------------------------------------------+-----------+
| # col_name                   | data_type                                             |
comment    |
|                              | NULL                                                  |
NULL       |
| x                            | int                                                   |
None       |
| y                            | int                                                   |
None       |
| s                            | string                                                |
None       |
|                              | NULL                                                  |
NULL       |
| # Detailed Table Information | NULL                                                  |
NULL       |
| Database:                    | describe_formatted                                    |
NULL       |
| Owner:                       | doc_demo                                              |
NULL       |
| CreateTime:                  | Mon Jul 22 17:01:47 EDT 2013                          |
```

```
NULL       |
| LastAccessTime:             | UNKNOWN                                                    |
NULL       |
| Protect Mode:               | None                                                       |
NULL       |
| Retention:                  | 0                                                          |
NULL       |
| Location:                   | hdfs://127.0.0.1:8020/user/doc_demo/sample_data    |
NULL       |
| Table Type:                 | EXTERNAL_TABLE                                             |
NULL       |
| Table Parameters:           | NULL                                                       |
NULL       |
|                             | EXTERNAL                                                   |
TRUE       |
|                             | transient_lastDdlTime                                      |
1374526907 |
|                             | NULL                                                       |
NULL       |
| # Storage Information       | NULL                                                       |
NULL       |
| SerDe Library:              | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe |
NULL       |
| InputFormat:                | com.cloudera.impala.hive.serde.ParquetInputFormat  |
NULL       |
| OutputFormat:               | com.cloudera.impala.hive.serde.ParquetOutputFormat |
NULL       |
| Compressed:                 | No                                                         |
NULL       |
| Num Buckets:                | 0                                                          |
NULL       |
| Bucket Columns:             | []                                                         |
NULL       |
| Sort Columns:               | []                                                         |
NULL       |
+-----------------------------+------------------------------------------------------------+-----------+
Returned 27 row(s) in 0.17s
```

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its `LOCATION` attribute.)

Cloudera Manager considerations:

A change in the behavior of metadata loading in CDH 5.12 / Impala 2.9 could lead to certain long-running statements being left out of the Cloudera Manager list of Impala queries until the statements are completed.

Prior to CDH 5.12 / Impala 2.9, statements such as `DESCRIBE` could cause the Impala web UI, and Cloudera Manager monitoring pages that rely on information from the web UI, to become unresponsive while they ran. The first time Impala references a large table, for example one with thousands of partitions, the statement might take longer than normal while the metadata for the table is loaded. In CDH 5.12 / Impala 2.9 and higher, the Impala web UI and associated Cloudera Manager monitoring pages are more responsive while a metadata loading operation is in progress.

Although the statement that loads the metadata shows up on the Impala web UI **/queries** page immediately, it does not show up in the Cloudera Manager list of queries until the metadata is finished loading. For example, the first `DESCRIBE` of a large partitioned table might take 30 minutes due to metadata loading, and the statement does not show up in Cloudera Manager during those 30 minutes.

Kudu considerations:

The information displayed for Kudu tables includes the additional attributes that are only applicable for Kudu tables:

- Whether or not the column is part of the primary key. Every Kudu table has a `true` value here for at least one column. There could be multiple `true` values, for tables with composite primary keys.

- Whether or not the column is nullable. Specified by the NULL or NOT NULL attributes on the CREATE TABLE statement. Columns that are part of the primary key are automatically non-nullable.
- The default value, if any, for the column. Specified by the DEFAULT attribute on the CREATE TABLE statement. If the default value is NULL, that is not indicated in this column. It is implied by nullable being true and no other default value specified.
- The encoding used for values in the column. Specified by the ENCODING attribute on the CREATE TABLE statement.
- The compression used for values in the column. Specified by the COMPRESSION attribute on the CREATE TABLE statement.
- The block size (in bytes) used for the underlying Kudu storage layer for the column. Specified by the BLOCK_SIZE attribute on the CREATE TABLE statement.

The following example shows DESCRIBE output for a simple Kudu table, with a single-column primary key and all column attributes left with their default values:

```
describe million_rows;
+------+--------+---------+------------+---------+--------------+--------------+-------------------+------------+
| name | type   | comment | primary_key | nullable | default_value | encoding      |
compression          | block_size |
+------+--------+---------+------------+---------+--------------+--------------+-------------------+------------+
| id   | string |         | true        | false    |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| s    | string |         | false       | false    |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
+------+--------+---------+------------+---------+--------------+--------------+-------------------+------------+
```

The following example shows DESCRIBE output for a Kudu table with a two-column primary key, and Kudu-specific attributes applied to some columns:

```
create table kudu_describe_example
(
  c1 int, c2 int,
  c3 string, c4 string not null, c5 string default 'n/a', c6 string default '',
  c7 bigint not null, c8 bigint null default null, c9 bigint default -1 encoding
bit_shuffle,
  primary key(c1,c2)
)
partition by hash (c1, c2) partitions 10 stored as kudu;

describe kudu_describe_example;
+------+--------+---------+------------+---------+--------------+--------------+-------------------+------------+
| name | type   | comment | primary_key | nullable | default_value | encoding      |
compression          | block_size |
+------+--------+---------+------------+---------+--------------+--------------+-------------------+------------+
| c1   | int    |         | true        | false    |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c2   | int    |         | true        | false    |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c3   | string |         | false       | true     |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c4   | string |         | false       | false    |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c5   | string |         | false       | true     | n/a           | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c6   | string |         | false       | true     |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c7   | bigint |         | false       | false    |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c8   | bigint |         | false       | true     |               | AUTO_ENCODING |
DEFAULT_COMPRESSION | 0           |
| c9   | bigint |         | false       | true     | -1            | BIT_SHUFFLE   |
DEFAULT_COMPRESSION | 0           |
+------+--------+---------+------------+---------+--------------+--------------+-------------------+------------+
```

Related information:

Overview of Impala Tables on page 196, CREATE TABLE Statement on page 234, SHOW TABLES Statement on page 369, SHOW CREATE TABLE Statement on page 370

DROP DATABASE Statement

Removes a database from the system. The physical operations involve removing the metadata for the database from the metastore, and deleting the corresponding `*.db` directory from HDFS.

Syntax:

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT | CASCADE];
```

Statement type: DDL

Usage notes:

By default, the database must be empty before it can be dropped, to avoid losing any data.

In CDH 5.5 / Impala 2.3 and higher, you can include the CASCADE clause to make Impala drop all tables and other objects in the database before dropping the database itself. The RESTRICT clause enforces the original requirement that the database be empty before being dropped. Because the RESTRICT behavior is still the default, this clause is optional.

The automatic dropping resulting from the CASCADE clause follows the same rules as the corresponding DROP TABLE, DROP VIEW, and DROP FUNCTION statements. In particular, the HDFS directories and data files for any external tables are left behind when the tables are removed.

When you do not use the CASCADE clause, drop or move all the objects inside the database manually before dropping the database itself:

- Use the SHOW TABLES statement to locate all tables and views in the database, and issue DROP TABLE and DROP VIEW statements to remove them all.
- Use the SHOW FUNCTIONS and SHOW AGGREGATE FUNCTIONS statements to locate all user-defined functions in the database, and issue DROP FUNCTION and DROP AGGREGATE FUNCTION statements to remove them all.
- To keep tables or views contained by a database while removing the database itself, use ALTER TABLE and ALTER VIEW to move the relevant objects to a different database before dropping the original database.

You cannot drop the current database, that is, the database your session connected to either through the USE statement or the `-d` option of `impala-shell`. Issue a USE statement to switch to a different database first. Because the `default` database is always available, issuing USE `default` is a convenient way to leave the current database before dropping it.

Hive considerations:

When you drop a database in Impala, the database can no longer be used by Hive.

Examples:

See CREATE DATABASE Statement on page 226 for examples covering CREATE DATABASE, USE, and DROP DATABASE.

Amazon S3 considerations:

In CDH 5.8 / Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to CDH 5.8 / Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed. See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have write permission for the directory associated with the database.

Examples:

```
create database first_db;
use first_db;
create table t1 (x int);

create database second_db;
use second_db;
-- Each database has its own namespace for tables.
-- You can reuse the same table names in each database.
create table t1 (s string);

create database temp;

-- You can either USE a database after creating it,
-- or qualify all references to the table name with the name of the database.
-- Here, tables T2 and T3 are both created in the TEMP database.

create table temp.t2 (x int, y int);
use database temp;
create table t3 (s string);

-- You cannot drop a database while it is selected by the USE statement.
drop database temp;
ERROR: AnalysisException: Cannot drop current default database: temp

-- The always-available database 'default' is a convenient one to USE
-- before dropping a database you created.
use default;

-- Before dropping a database, first drop all the tables inside it,
-- or in CDH 5.5 / Impala 2.3 and higher use the CASCADE clause.
drop database temp;
ERROR: ImpalaRuntimeException: Error making 'dropDatabase' RPC to Hive Metastore:
CAUSED BY: InvalidOperationException: Database temp is not empty
show tables in temp;
+------+
| name |
+------+
| t3   |
+------+

-- CDH 5.5 / Impala 2.3 and higher:
drop database temp cascade;

-- Earlier releases:
drop table temp.t3;
drop database temp;
```

Related information:

Overview of Impala Databases on page 193, CREATE DATABASE Statement on page 226, USE Statement on page 385, SHOW DATABASES on page 368, DROP TABLE Statement on page 268

DROP FUNCTION Statement

Removes a user-defined function (UDF), so that it is not available for execution during Impala SELECT or INSERT operations.

Syntax:

To drop C++ UDFs and UDAs:

```
DROP [AGGREGATE] FUNCTION [IF EXISTS] [db_name.]function_name(type[, type...])
```



Note:

The preceding syntax, which includes the function signature, also applies to Java UDFs that were created using the corresponding CREATE FUNCTION syntax that includes the argument and return types. After upgrading to CDH 5.7 / Impala 2.5 or higher, consider re-creating all Java UDFs with the CREATE FUNCTION syntax that does not include the function signature. Java UDFs created this way are now persisted in the metastore database and do not need to be re-created after an Impala restart.

To drop Java UDFs (created using the CREATE FUNCTION syntax with no function signature):

```
DROP FUNCTION [IF EXISTS] [db_name.]function_name
```

Statement type: DDL

Usage notes:

Because the same function name could be overloaded with different argument signatures, you specify the argument types to identify the exact function to drop.

Restrictions:

In CDH 5.7 / Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new CREATE FUNCTION syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old CREATE FUNCTION syntax do not persist across restarts because they are held in the memory of the catalogd daemon. Until you re-create such Java UDFs using the new CREATE FUNCTION syntax, you must reload those Java-based UDFs by running the original CREATE FUNCTION statements again each time you restart the catalogd daemon. Prior to CDH 5.7 / Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, does not need any particular HDFS permissions to perform this statement. All read and write operations are on the metastore database, not HDFS files and directories.

Examples:

The following example shows how to drop Java functions created with the signatureless CREATE FUNCTION syntax in CDH 5.7 / Impala 2.5 and higher. Issuing DROP FUNCTION *function_name* removes all the overloaded functions under that name. (See CREATE FUNCTION Statement on page 228 for a longer example showing how to set up such functions in the first place.)

```
create function my_func location '/user/impala/udfs/udf-examples-cdh570.jar'
  symbol='com.cloudera.impala.TestUdf';

show functions;
+-------------+---------------------------------------+-------------+---------------+
| return type | signature                             | binary type | is persistent |
+-------------+---------------------------------------+-------------+---------------+
| BIGINT      | my_func(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | my_func(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | my_func(BOOLEAN, BOOLEAN)             | JAVA        | true          |
...
| BIGINT      | testudf(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true          |
...

drop function my_func;
show functions;
+-------------+---------------------------------------+-------------+---------------+
| return type | signature                             | binary type | is persistent |
```

```
+-------------+---------------------------------------+-------------+---------------+
| BIGINT      | testudf(BIGINT)                       | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true          |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true          |
...
```

Related information:

User-Defined Functions (UDFs) on page 525, CREATE FUNCTION Statement on page 228

DROP ROLE Statement (CDH 5.2 or higher only)

The DROP ROLE statement removes a role from the metastore database. Once dropped, the role is revoked for all users to whom it was previously assigned, and all privileges granted to that role are revoked. Queries that are already executing are not affected. Impala verifies the role information approximately every 60 seconds, so the effects of DROP ROLE might not take effect for new Impala queries for a brief period.

Syntax:

```
DROP ROLE role_name
```

Required privileges:

Only administrative users (initially, a predefined set of users specified in the Sentry service configuration file) can use this statement.

Compatibility:

Impala makes use of any roles and privileges specified by the GRANT and REVOKE statements in Hive, and Hive makes use of any roles and privileges specified by the GRANT and REVOKE statements in Impala. The Impala GRANT and REVOKE statements for privileges do not require the ROLE keyword to be repeated before each role name, unlike the equivalent Hive statements.

Related information:

Enabling Sentry Authorization for Impala on page 87, GRANT Statement (CDH 5.2 or higher only) on page 273 REVOKE Statement (CDH 5.2 or higher only) on page 293, CREATE ROLE Statement (CDH 5.2 or higher only) on page 234, SHOW Statement on page 363

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

DROP STATS Statement

Removes the specified statistics from a table or partition. The statistics were originally created by the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement.

Syntax:

```
DROP STATS [database_name.]table_name
DROP INCREMENTAL STATS [database_name.]table_name PARTITION (partition_spec)

partition_spec ::= partition_col=constant_value
```

The PARTITION clause is only allowed in combination with the INCREMENTAL clause. It is optional for COMPUTE INCREMENTAL STATS, and required for DROP INCREMENTAL STATS. Whenever you specify partitions through the PARTITION (*partition_spec*) clause in a COMPUTE INCREMENTAL STATS or DROP INCREMENTAL STATS statement, you must include all the partitioning columns in the specification, and specify constant values for all the partition key columns.

DROP STATS removes all statistics from the table, whether created by COMPUTE STATS or COMPUTE INCREMENTAL STATS.

DROP INCREMENTAL STATS only affects incremental statistics for a single partition, specified through the PARTITION clause. The incremental stats are marked as outdated, so that they are recomputed by the next COMPUTE INCREMENTAL STATS statement.

Usage notes:

You typically use this statement when the statistics for a table or a partition have become stale due to data files being added to or removed from the associated HDFS data directories, whether by manual HDFS operations or INSERT, INSERT OVERWRITE, or LOAD DATA statements, or adding or dropping partitions.

When a table or partition has no associated statistics, Impala treats it as essentially zero-sized when constructing the execution plan for a query. In particular, the statistics influence the order in which tables are joined in a join query. To ensure proper query planning and good query performance and scalability, make sure to run COMPUTE STATS or COMPUTE INCREMENTAL STATS on the table or partition after removing any stale statistics.

Dropping the statistics is not required for an unpartitioned table or a partitioned table covered by the original type of statistics. A subsequent COMPUTE STATS statement replaces any existing statistics with new ones, for all partitions, regardless of whether the old ones were outdated. Therefore, this statement was rarely used before the introduction of incremental statistics.

Dropping the statistics is required for a partitioned table containing incremental statistics, to make a subsequent COMPUTE INCREMENTAL STATS statement rescan an existing partition. See Table and Column Statistics on page 575 for information about incremental statistics, a new feature available in Impala 2.1.0 and higher.

Statement type: DDL

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, does not need any particular HDFS permissions to perform this statement. All read and write operations are on the metastore database, not HDFS files and directories.

Examples:

The following example shows a partitioned table that has associated statistics produced by the COMPUTE INCREMENTAL STATS statement, and how the situation evolves as statistics are dropped from specific partitions, then the entire table.

Initially, all table and column statistics are filled in.

```
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+-----------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+-----------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+-----------------
show column stats item_partitioned;
+------------------+-----------+------------------+--------+----------+--------------
| Column           | Type      | #Distinct Values | #Nulls | Max Size | Avg Size
+------------------+-----------+------------------+--------+----------+--------------
| i_item_sk        | INT       | 19443            | -1     | 4        | 4
| i_item_id        | STRING    | 9025             | -1     | 16       | 16
| i_rec_start_date | TIMESTAMP | 4                | -1     | 16       | 16
| i_rec_end_date   | TIMESTAMP | 3                | -1     | 16       | 16
| i_item_desc      | STRING    | 13330            | -1     | 200      | 100.302803039
| i_current_price  | FLOAT     | 2807             | -1     | 4        | 4
```

```
| i_wholesale_cost | FLOAT     | 2105              | -1     | 4         | 4
| i_brand_id       | INT       | 965               | -1     | 4         | 4
| i_brand          | STRING    | 725               | -1     | 22        | 16.1776008605
| i_class_id       | INT       | 16                | -1     | 4         | 4
| i_class          | STRING    | 101               | -1     | 15        | 7.76749992370
| i_category_id    | INT       | 10                | -1     | 4         | 4
| i_manufact_id    | INT       | 1857              | -1     | 4         | 4
| i_manufact       | STRING    | 1028              | -1     | 15        | 11.3295001983
| i_size           | STRING    | 8                 | -1     | 11        | 4.33459997177
| i_formulation    | STRING    | 12884             | -1     | 20        | 19.9799995422
| i_color          | STRING    | 92                | -1     | 10        | 5.38089990615
| i_units          | STRING    | 22                | -1     | 7         | 4.18690013885
| i_container      | STRING    | 2                 | -1     | 7         | 6.99259996414
| i_manager_id     | INT       | 105               | -1     | 4         | 4
| i_product_name   | STRING    | 19094             | -1     | 25        | 18.0233001708
| i_category       | STRING    | 10                | 0      | -1        | -1
+------------------+-----------+-------------------+--------+-----------+-------------
```

To remove statistics for particular partitions, use the `DROP INCREMENTAL STATS` statement. After removing statistics for two partitions, the table-level statistics reflect that change in the `#Rows` and `Incremental stats` fields. The counts, maximums, and averages of the column-level statistics are unaffected.



Note: (It is possible that the row count might be preserved in future after a `DROP INCREMENTAL STATS` statement. Check the resolution of the issue IMPALA-1615.)

```
drop incremental stats item_partitioned partition (i_category='Sports');
drop incremental stats item_partitioned partition (i_category='Electronics');

show table stats item_partitioned
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------
show column stats item_partitioned
+------------------+-----------+------------------+--------+----------+--------------
| Column           | Type      | #Distinct Values | #Nulls | Max Size | Avg Size
+------------------+-----------+------------------+--------+----------+--------------
| i_item_sk        | INT       | 19443            | -1     | 4        | 4
| i_item_id        | STRING    | 9025             | -1     | 16       | 16
| i_rec_start_date | TIMESTAMP | 4                | -1     | 16       | 16
| i_rec_end_date   | TIMESTAMP | 3                | -1     | 16       | 16
| i_item_desc      | STRING    | 13330            | -1     | 200      | 100.302803039
| i_current_price  | FLOAT     | 2807             | -1     | 4        | 4
| i_wholesale_cost | FLOAT     | 2105             | -1     | 4        | 4
| i_brand_id       | INT       | 965              | -1     | 4        | 4
| i_brand          | STRING    | 725              | -1     | 22       | 16.1776008605
| i_class_id       | INT       | 16               | -1     | 4        | 4
| i_class          | STRING    | 101              | -1     | 15       | 7.76749992370
| i_category_id    | INT       | 10               | -1     | 4        | 4
| i_manufact_id    | INT       | 1857             | -1     | 4        | 4
| i_manufact       | STRING    | 1028             | -1     | 15       | 11.3295001983
| i_size           | STRING    | 8                | -1     | 11       | 4.33459997177
| i_formulation    | STRING    | 12884            | -1     | 20       | 19.9799995422
| i_color          | STRING    | 92               | -1     | 10       | 5.38089990615
| i_units          | STRING    | 22               | -1     | 7        | 4.18690013885
| i_container      | STRING    | 2                | -1     | 7        | 6.99259996414
| i_manager_id     | INT       | 105              | -1     | 4        | 4
| i_product_name   | STRING    | 19094            | -1     | 25       | 18.0233001708
```

```
| i_category       | STRING    | 10               | 0      | -1       | -1
+------------------+-----------+------------------+--------+----------+-------------
```

To remove all statistics from the table, whether produced by COMPUTE STATS or COMPUTE INCREMENTAL STATS, use the DROP STATS statement without the INCREMENTAL clause). Now, both table-level and column-level statistics are reset.

```
drop stats item_partitioned;

show table stats item_partitioned
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | -1    | 1      | 223.74KB | NOT CACHED   | PARQUET | false
| Children    | -1    | 1      | 230.05KB | NOT CACHED   | PARQUET | false
| Electronics | -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home        | -1    | 1      | 232.56KB | NOT CACHED   | PARQUET | false
| Jewelry     | -1    | 1      | 223.72KB | NOT CACHED   | PARQUET | false
| Men         | -1    | 1      | 231.25KB | NOT CACHED   | PARQUET | false
| Music       | -1    | 1      | 237.90KB | NOT CACHED   | PARQUET | false
| Shoes       | -1    | 1      | 234.90KB | NOT CACHED   | PARQUET | false
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | -1    | 1      | 226.27KB | NOT CACHED   | PARQUET | false
| Total       | -1    | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------
show column stats item_partitioned
+------------------+-----------+------------------+--------+----------+----------+
| Column           | Type      | #Distinct Values | #Nulls | Max Size | Avg Size |
+------------------+-----------+------------------+--------+----------+----------+
| i_item_sk        | INT       | -1               | -1     | 4        | 4        |
| i_item_id        | STRING    | -1               | -1     | -1       | -1       |
| i_rec_start_date | TIMESTAMP | -1               | -1     | 16       | 16       |
| i_rec_end_date   | TIMESTAMP | -1               | -1     | 16       | 16       |
| i_item_desc      | STRING    | -1               | -1     | -1       | -1       |
| i_current_price  | FLOAT     | -1               | -1     | 4        | 4        |
| i_wholesale_cost | FLOAT     | -1               | -1     | 4        | 4        |
| i_brand_id       | INT       | -1               | -1     | 4        | 4        |
| i_brand          | STRING    | -1               | -1     | -1       | -1       |
| i_class_id       | INT       | -1               | -1     | 4        | 4        |
| i_class          | STRING    | -1               | -1     | -1       | -1       |
| i_category_id    | INT       | -1               | -1     | 4        | 4        |
| i_manufact_id    | INT       | -1               | -1     | 4        | 4        |
| i_manufact       | STRING    | -1               | -1     | -1       | -1       |
| i_size           | STRING    | -1               | -1     | -1       | -1       |
| i_formulation    | STRING    | -1               | -1     | -1       | -1       |
| i_color          | STRING    | -1               | -1     | -1       | -1       |
| i_units          | STRING    | -1               | -1     | -1       | -1       |
| i_container      | STRING    | -1               | -1     | -1       | -1       |
| i_manager_id     | INT       | -1               | -1     | 4        | 4        |
| i_product_name   | STRING    | -1               | -1     | -1       | -1       |
| i_category       | STRING    | 10               | 0      | -1       | -1       |
+------------------+-----------+------------------+--------+----------+----------+
```

Related information:

COMPUTE STATS Statement on page 219, SHOW TABLE STATS Statement on page 373, SHOW COLUMN STATS Statement on page 375, Table and Column Statistics on page 575

DROP TABLE Statement

Removes an Impala table. Also removes the underlying HDFS data files for internal tables, although not for external tables.

Syntax:

```
DROP TABLE [IF EXISTS] [db_name.]table_name [PURGE]
```

IF EXISTS clause:

The optional IF EXISTS clause makes the statement succeed whether or not the table exists. If the table does exist, it is dropped; if it does not exist, the statement has no effect. This capability is useful in standardized setup scripts that remove existing schema objects and create new ones. By using some combination of IF EXISTS for the DROP statements and IF NOT EXISTS clauses for the CREATE statements, the script can run successfully the first time you run it (when the objects do not exist yet) and subsequent times (when some or all of the objects do already exist).

PURGE clause:

The optional PURGE keyword, available in CDH 5.5 / Impala 2.3 and higher, causes Impala to remove the associated HDFS data files immediately, rather than going through the HDFS trashcan mechanism. Use this keyword when dropping a table if it is crucial to remove the data as quickly as possible to free up space, or if there is a problem with the trashcan, such as the trash cannot being configured or being in a different HDFS encryption zone than the data files.

Statement type: DDL

Usage notes:

By default, Impala removes the associated HDFS directory and data files for the table. If you issue a DROP TABLE and the data files are not deleted, it might be for the following reasons:

- If the table was created with the EXTERNAL clause, Impala leaves all files and directories untouched. Use external tables when the data is under the control of other Hadoop components, and Impala is only used to query the data files from their original locations.
- Impala might leave the data files behind unintentionally, if there is no HDFS location available to hold the HDFS trashcan for the impala user. See User Account Requirements on page 25 for the procedure to set up the required HDFS home directory.

Make sure that you are in the correct database before dropping a table, either by issuing a USE statement first or by using a fully qualified name *db_name*.*table_name*.

If you intend to issue a DROP DATABASE statement, first issue DROP TABLE statements to remove all the tables in that database.

Examples:

```
create database temporary;
use temporary;
create table unimportant (x int);
create table trivial (s string);
-- Drop a table in the current database.
drop table unimportant;
-- Switch to a different database.
use default;
-- To drop a table in a different database...
drop table trivial;
ERROR: AnalysisException: Table does not exist: default.trivial
-- ...use a fully qualified name.
drop table temporary.trivial;
```

For other tips about managing and reclaiming Impala disk space, see Managing Disk Space for Impala Data on page 77.

Amazon S3 considerations:

The DROP TABLE statement can remove data files from S3 if the associated S3 table is an internal table. In CDH 5.8 / Impala 2.6 and higher, as part of improved support for writing to S3, Impala also removes the associated folder when dropping an internal table that resides on S3. See Using Impala with the Amazon S3 Filesystem on page 692 for details about working with S3 tables.

For best compatibility with the S3 write support in CDH 5.8 / Impala 2.6 and higher:

- Use native Hadoop techniques to create data files in S3 for querying through Impala.
- Use the PURGE clause of DROP TABLE when dropping internal (managed) tables.

By default, when you drop an internal (managed) table, the data files are moved to the HDFS trashcan. This operation is expensive for tables that reside on the Amazon S3 filesystem. Therefore, for S3 tables, prefer to use DROP TABLE

table_name PURGE rather than the default DROP TABLE statement. The PURGE clause makes Impala delete the data files immediately, skipping the HDFS trashcan. For the PURGE clause to work effectively, you must originally create the data files on S3 using one of the tools from the Hadoop ecosystem, such as hadoop fs -cp, or INSERT in Impala or Hive.

In CDH 5.8 / Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to CDH 5.8 / Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed. See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

Cancellation: Cannot be cancelled.

HDFS permissions:

For an internal table, the user ID that the impalad daemon runs under, typically the impala user, must have write permission for all the files and directories that make up the table.

For an external table, dropping the table only involves changes to metadata in the metastore database. Because Impala does not remove any HDFS files or directories when external tables are dropped, no particular permissions are needed for the associated HDFS files or directories.

Kudu considerations:

Kudu tables can be managed or external, the same as with HDFS-based tables. For a managed table, the underlying Kudu table and its data are removed by DROP TABLE. For an external table, the underlying Kudu table and its data remain after a DROP TABLE.

Related information:

Overview of Impala Tables on page 196, ALTER TABLE Statement on page 205, CREATE TABLE Statement on page 234, Partitioning for Impala Tables on page 625, Internal Tables on page 197, External Tables on page 197

DROP VIEW Statement

Removes the specified view, which was originally created by the CREATE VIEW statement. Because a view is purely a logical construct (an alias for a query) with no physical data behind it, DROP VIEW only involves changes to metadata in the metastore database, not any data files in HDFS.

Syntax:

```
DROP VIEW [IF EXISTS] [db_name.]view_name
```

Statement type: DDL

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

The following example creates a series of views and then drops them. These examples illustrate how views are associated with a particular database, and both the view definitions and the view names for CREATE VIEW and DROP VIEW can refer to a view in the current database or a fully qualified view name.

```
-- Create and drop a view in the current database.
CREATE VIEW few_rows_from_t1 AS SELECT * FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;

-- Create and drop a view referencing a table in a different database.
CREATE VIEW table_from_other_db AS SELECT x FROM db1.foo WHERE x IS NOT NULL;
DROP VIEW table_from_other_db;

USE db1;
```

```
-- Create a view in a different database.
CREATE VIEW db2.v1 AS SELECT * FROM db2.foo;
-- Switch into the other database and drop the view.
USE db2;
DROP VIEW v1;

USE db1;
-- Create a view in a different database.
CREATE VIEW db2.v1 AS SELECT * FROM db2.foo;
-- Drop a view in the other database.
DROP VIEW db2.v1;
```

Related information:

Overview of Impala Views on page 199, CREATE VIEW Statement on page 248, ALTER VIEW Statement on page 218

EXPLAIN Statement

Returns the execution plan for a statement, showing the low-level mechanisms that Impala will use to read the data, divide the work among nodes in the cluster, and transmit intermediate and final results across the network. Use `explain` followed by a complete `SELECT` query. For example:

Syntax:

```
EXPLAIN { select_query | ctas_stmt | insert_stmt }
```

The *select_query* is a `SELECT` statement, optionally prefixed by a `WITH` clause. See SELECT Statement on page 295 for details.

The *insert_stmt* is an `INSERT` statement that inserts into or overwrites an existing table. It can use either the `INSERT ... SELECT` or `INSERT ... VALUES` syntax. See INSERT Statement on page 277 for details.

The *ctas_stmt* is a `CREATE TABLE` statement using the `AS SELECT` clause, typically abbreviated as a "CTAS" operation. See CREATE TABLE Statement on page 234 for details.

Usage notes:

You can interpret the output to judge whether the query is performing efficiently, and adjust the query and/or the schema if not. For example, you might change the tests in the `WHERE` clause, add hints to make join operations more efficient, introduce subqueries, change the order of tables in a join, add or change partitioning for a table, collect column statistics and/or table statistics in Hive, or any other performance tuning steps.

The `EXPLAIN` output reminds you if table or column statistics are missing from any table involved in the query. These statistics are important for optimizing queries involving large tables or multi-table joins. See COMPUTE STATS Statement on page 219 for how to gather statistics, and Table and Column Statistics on page 575 for how to use this information for query tuning.

Read the `EXPLAIN` plan from bottom to top:

- The last part of the plan shows the low-level details such as the expected amount of data that will be read, where you can judge the effectiveness of your partitioning strategy and estimate how long it will take to scan a table based on total data size and the size of the cluster.
- As you work your way up, next you see the operations that will be parallelized and performed on each Impala node.
- At the higher levels, you see how data flows when intermediate result sets are combined and transmitted from one node to another.
- See EXPLAIN_LEVEL Query Option on page 331 for details about the `EXPLAIN_LEVEL` query option, which lets you customize how much detail to show in the `EXPLAIN` plan depending on whether you are doing high-level or low-level tuning, dealing with logical or physical aspects of the query.

If you come from a traditional database background and are not familiar with data warehousing, keep in mind that Impala is optimized for full table scans across very large tables. The structure and distribution of this data is typically not suitable for the kind of indexing and single-row lookups that are common in OLTP environments. Seeing a query

scan entirely through a large table is common, not necessarily an indication of an inefficient query. Of course, if you can reduce the volume of scanned data by orders of magnitude, for example by using a query that affects only certain partitions within a partitioned table, then you might be able to optimize a query so that it executes in seconds rather than minutes.

For more information and examples to help you interpret EXPLAIN output, see Using the EXPLAIN Plan for Performance Tuning on page 602.

Extended EXPLAIN output:

For performance tuning of complex queries, and capacity planning (such as using the admission control and resource management features), you can enable more detailed and informative output for the EXPLAIN statement. In the `impala-shell` interpreter, issue the command `SET EXPLAIN_LEVEL=`*`level`*, where *level* is an integer from 0 to 3 or corresponding mnemonic values `minimal`, `standard`, `extended`, or `verbose`.

When extended EXPLAIN output is enabled, EXPLAIN statements print information about estimated memory requirements, minimum number of virtual cores, and so on.

Starting in CDH 6.2 / Impala 3.2, if the EXPLAIN_LEVEL option is set to EXTENDED level or VERBOSE, the output contains the following additional information.

- The analyzed query, in the output header.

  The analyzed query may have been rewritten to include various optimizations and implicit casts. See the example below.

- The predicates in the plan output includes the same implicit casts and literals printed with a cast to show the type.

See EXPLAIN_LEVEL Query Option on page 331 for details and examples.

Examples:

This example shows how the standard EXPLAIN output moves from the lowest (physical) level to the higher (logical) levels. The query begins by scanning a certain amount of data; each node performs an aggregation operation (evaluating `COUNT(*)`) on some subset of data that is local to that node; the intermediate results are transmitted back to the coordinator node (labelled here as the EXCHANGE node); lastly, the intermediate results are summed to display the final result.

```
[impalad-host:21000] > explain select count(*) from customer_address;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=42.00MB VCores=1 |
|                                                          |
| 03:AGGREGATE [MERGE FINALIZE]                            |
| |  output: sum(count(*))                                 |
| |                                                        |
| 02:EXCHANGE [PARTITION=UNPARTITIONED]                    |
| |                                                        |
| 01:AGGREGATE                                             |
| |  output: count(*)                                      |
| |                                                        |
| 00:SCAN HDFS [default.customer_address]                  |
|    partitions=1/1 size=5.25MB                            |
+----------------------------------------------------------+
```

The following example shows an extended EXPLAIN output. Note that the analyzed query was rewritten to include:

- The 'constant folding' optimization, which simplified the expression in the original query, '`1000 / 100`' to '`10`'.
- The implicit casts in the WHERE clause.

```
EXPLAIN SELECT * FROM functional_kudu.alltypestiny WHERE bigint_col < 1000 / 100;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| ...
| Analyzed query: SELECT * FROM mytable WHERE CAST(bigint_col AS DOUBLE) < CAST(10 AS
```

```
DOUBLE)
| ...
| 00:SCAN KUDU [functional_kudu.alltypestiny]
| predicates: CAST(bigint_col AS DOUBLE) < CAST(10 AS DOUBLE)
...
```

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts. See http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/sg_redaction.html for details.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read and execute permissions for all applicable directories in all source tables for the query that is being explained. (A `SELECT` operation could read files from multiple different HDFS directories if the source table is partitioned.)

Kudu considerations:

The `EXPLAIN` statement displays equivalent plan information for queries against Kudu tables as for queries against HDFS-based tables.

To see which predicates Impala can "push down" to Kudu for efficient evaluation, without transmitting unnecessary rows back to Impala, look for the `kudu predicates` item in the scan phase of the query. The label `kudu predicates` indicates a condition that can be evaluated efficiently on the Kudu side. The label `predicates` in a `SCAN KUDU` node indicates a condition that is evaluated by Impala. For example, in a table with primary key column `X` and non-primary key column `Y`, you can see that some operators in the `WHERE` clause are evaluated immediately by Kudu and others are evaluated later by Impala:

```
EXPLAIN SELECT x,y from kudu_table WHERE
  x = 1 AND y NOT IN (2,3) AND z = 1
  AND a IS NOT NULL AND b > 0 AND length(s) > 5;
+----------------
| Explain String
+----------------
...
| 00:SCAN KUDU [kudu_table]
|    predicates: y NOT IN (2, 3), length(s) > 5
|    kudu predicates: a IS NOT NULL, b > 0, x = 1, z = 1
```

Only binary predicates, `IS NULL` and `IS NOT NULL` (in CDH 5.12 and higher), and `IN` predicates containing literal values that exactly match the types in the Kudu table, and do not require any casting, can be pushed to Kudu.

Related information:

SELECT Statement on page 295, INSERT Statement on page 277, CREATE TABLE Statement on page 234, Understanding Impala Query Performance - EXPLAIN Plans and Query Profiles on page 602

GRANT Statement (CDH 5.2 or higher only)

The `GRANT` statement grants a privilege on a specified object to a role or grants a role to a group.

Syntax:

```
GRANT ROLE role_name TO GROUP group_name

GRANT privilege ON object_type object_name
   TO [ROLE] roleName
   [WITH GRANT OPTION]
```

```
privilege ::= ALL | CREATE | INSERT | REFRESH | SELECT | SELECT(column_name)
object_type ::= SERVER | URI | DATABASE | TABLE
```

Typically, the object name is an identifier. For URIs, it is a string literal.

Required privileges:

Only Sentry administrative users, users who belong to the groups defined in `sentry.service.admin.group` of the Sentry configuration can grant roles to a group.

The `WITH GRANT OPTION` clause allows members of the specified role to issue `GRANT` and `REVOKE` statements for those same privileges. Hence, if a role has the `ALL` privilege on a database and the `WITH GRANT OPTION` set, users granted that role can execute `GRANT`/`REVOKE` statements only for that database or child tables of the database. This means a user could revoke the privileges of the user that provided them the `GRANT OPTION`.

The ability to grant or revoke `SELECT` privilege on specific columns is available in CDH 5.5 / Impala 2.3 and higher. See Hive SQL Syntax for Use with Sentry for details.

Usage notes:

You can only grant the `ALL` privilege to the `URI` object. Finer-grained privileges mentioned below on a `URI` are not supported.

The table below lists the minimum level of privileges and the scope required to execute SQL statements in CDH 6.1 / CDH 5.16 and higher. The following notations are used:

- **ANY** denotes the `SELECT`, `INSERT`, `CREATE`, ***or*** `REFRESH` privilege.
- **ALL** privilege denotes the `SELECT`, `INSERT`, `CREATE`, ***and*** `REFRESH` privileges.
- The owner of an object effectively has the `ALL` privilege on the object.
- The parent levels of the specified scope are implicitly supported. For example, if a privilege is listed with the TABLE scope, the same privilege granted on DATABASE and SERVER will allow the user to execute that specific SQL statement on TABLE.

SQL Statement	Privileges	Scope
SELECT	SELECT	TABLE
WITH SELECT	SELECT	TABLE
EXPLAIN SELECT	SELECT	TABLE
INSERT	INSERT	TABLE
EXPLAIN INSERT	INSERT	TABLE
TRUNCATE	INSERT	TABLE
LOAD	INSERT	TABLE
	ALL	URI
CREATE DATABASE	CREATE	SERVER
CREATE DATABASE LOCATION	CREATE	SERVER
	ALL	URI
CREATE TABLE	CREATE	DATABASE
CREATE TABLE LIKE	CREATE	DATABASE
	SELECT, INSERT, ***or*** REFRESH	TABLE
CREATE TABLE AS SELECT	CREATE	DATABASE
	INSERT	DATABASE
	SELECT	TABLE

EXPLAIN CREATE TABLE AS SELECT	CREATE	DATABASE
	INSERT	DATABASE
	SELECT	TABLE
CREATE TABLE LOCATION	CREATE	TABLE
	ALL	URI
CREATE VIEW	CREATE	DATABASE
	SELECT	TABLE
ALTER DATABASE SET OWNER	ALL WITH GRANT	DATABASE
ALTER TABLE	ALL	TABLE
ALTER TABLE SET LOCATION	ALL	TABLE
	ALL	URI
ALTER TABLE RENAME	CREATE	DATABASE
	ALL	TABLE
ALTER TABLE SET OWNER	ALL WITH GRANT	TABLE
ALTER VIEW	ALL	TABLE
	SELECT	TABLE
ALTER VIEW RENAME	CREATE	DATABASE
	ALL	TABLE
ALTER VIEW SET OWNER	ALL WITH GRANT	VIEW
DROP DATABASE	ALL	DATABASE
DROP TABLE	ALL	TABLE
DROP VIEW	ALL	TABLE
CREATE FUNCTION	CREATE	DATABASE
	ALL	URI
DROP FUNCTION	ALL	DATABASE
COMPUTE STATS	ALL	TABLE
DROP STATS	ALL	TABLE
INVALIDATE METADATA	REFRESH	SERVER
INVALIDATE METADATA <table>	REFRESH	TABLE
REFRESH <table>	REFRESH	TABLE
REFRESH AUTHORIZATION	REFRESH	SERVER
REFRESH FUNCTIONS	REFRESH	DATABASE
COMMENT ON DATABASE	ALL	DATABASE
COMMENT ON TABLE	ALL	TABLE
COMMENT ON VIEW	ALL	TABLE
COMMENT ON COLUMN	ALL	TABLE

DESCRIBE DATABASE	SELECT, INSERT, ***or*** REFRESH	DATABASE
DESCRIBE <table/view>	SELECT, INSERT, ***or*** REFRESH	TABLE
If the user has the SELECT privilege at the COLUMN level, only the columns the user has access will show.	SELECT	COLUMN
USE	ANY	TABLE
SHOW DATABASES	ANY	TABLE
SHOW TABLES	ANY	TABLE
SHOW FUNCTIONS	SELECT, INSERT, ***or*** REFRESH	DATABASE
SHOW PARTITIONS	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW TABLE STATS	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW COLUMN STATS	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW FILES	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW CREATE TABLE	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW CREATE VIEW	SELECT, INSERT, ***or*** REFRESH	TABLE
SHOW CREATE FUNCTION	SELECT, INSERT, ***or*** REFRESH	DATABASE
SHOW RANGE PARTITIONS (Kudu only)	SELECT, INSERT, ***or*** REFRESH	TABLE
UPDATE (Kudu only)	ALL	TABLE
EXPLAIN UPDATE (Kudu only)	ALL	TABLE
UPSERT (Kudu only)	ALL	TABLE
WITH UPSERT (Kudu only)	ALL	TABLE
EXPLAIN UPSERT (Kudu only)	ALL	TABLE
DELETE (Kudu only)	ALL	TABLE
EXPLAIN DELETE (Kudu only)	ALL	TABLE

Compatibility:

- The Impala GRANT and REVOKE statements are available in CDH 5.2 / Impala 2.0 and later.
- In CDH 5.1 / Impala 1.4 and later, Impala can make use of any roles and privileges specified by the GRANT and REVOKE statements in Hive, when your system is configured to use the Sentry service instead of the file-based policy mechanism.
- The Impala GRANT and REVOKE statements for privileges do not require the ROLE keyword to be repeated before each role name, unlike the equivalent Hive statements.
- Currently, each Impala GRANT or REVOKE statement can only grant or revoke a single privilege to or from a single role.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Kudu considerations:

Access to Kudu tables must be granted to and revoked from roles with the following considerations:

- Only users with the ALL privilege on SERVER can create external Kudu tables.

- The ALL privileges on SERVER is required to specify the kudu.master_addresses property in the CREATE TABLE statements for managed tables as well as external tables.
- Access to Kudu tables is enforced at the table level and at the column level.
- The SELECT- and INSERT-specific permissions are supported.
- The DELETE, UPDATE, and UPSERT operations require the ALL privilege.

Because non-SQL APIs can access Kudu data without going through Sentry authorization, currently the Sentry support is considered preliminary and subject to change.

Related information:

Enabling Sentry Authorization for Impala on page 87, REVOKE Statement (CDH 5.2 or higher only) on page 293, CREATE ROLE Statement (CDH 5.2 or higher only) on page 234, DROP ROLE Statement (CDH 5.2 or higher only) on page 265, SHOW Statement on page 363

INSERT Statement

Impala supports inserting into tables and partitions that you create with the Impala CREATE TABLE statement, or pre-defined tables and partitions created through Hive.

Syntax:

```
[with_clause]
  INSERT [hint_clause] { INTO | OVERWRITE } [TABLE] table_name
  [(column_list)]
  [ PARTITION (partition_clause)]
{
    [hint_clause] select_statement
  | VALUES (value [, value ...]) [, (value [, value ...]) ...]
}

partition_clause ::= col_name [= constant] [, col_name [= constant] ...]

hint_clause ::=
  hint_with_dashes |
  hint_with_cstyle_delimiters |
  hint_with_brackets

hint_with_dashes ::= -- +SHUFFLE | -- +NOSHUFFLE -- +CLUSTERED

hint_with_cstyle_comments ::= /* +SHUFFLE */ | /* +NOSHUFFLE */ | /* +CLUSTERED */

hint_with_brackets ::= [SHUFFLE] | [NOSHUFFLE]
  (With this hint format, the square brackets are part of the syntax.)
```

Note: The square bracket style of hint is now deprecated and might be removed in a future release. For that reason, any newly added hints are not available with the square bracket syntax.

Appending or replacing (INTO and OVERWRITE clauses):

The INSERT INTO syntax appends data to a table. The existing data files are left as-is, and the inserted data is put into one or more new data files.

The INSERT OVERWRITE syntax replaces the data in a table. Currently, the overwritten data files are deleted immediately; they do not go through the HDFS trash mechanism.

Complex type considerations:

The INSERT statement currently does not support writing data files containing complex types (ARRAY, STRUCT, and MAP). To prepare Parquet data for such tables, you generate the data files outside Impala and then use LOAD DATA or CREATE EXTERNAL TABLE to associate those data files with the table. Currently, such tables must use the Parquet file format. See Complex Types (CDH 5.5 or higher only) on page 139 for details about working with complex types.

Kudu considerations:

Currently, the INSERT OVERWRITE syntax cannot be used with Kudu tables.

Kudu tables require a unique primary key for each row. If an INSERT statement attempts to insert a row with the same values for the primary key columns as an existing row, that row is discarded and the insert operation continues. When rows are discarded due to duplicate primary keys, the statement finishes with a warning, not an error. (This is a change from early releases of Kudu where the default was to return in error in such cases, and the syntax INSERT IGNORE was required to make the statement succeed. The IGNORE clause is no longer part of the INSERT syntax.)

For situations where you prefer to replace rows with duplicate primary key values, rather than discarding the new data, you can use the UPSERT statement instead of INSERT. UPSERT inserts rows that are entirely new, and for rows that match an existing primary key in the table, the non-primary-key columns are updated to reflect the values in the "upserted" data.

If you really want to store new rows, not replace existing ones, but cannot do so because of the primary key uniqueness constraint, consider recreating the table with additional columns included in the primary key.

See Using Impala to Query Kudu Tables on page 670 for more details about using Impala with Kudu.

Usage notes:

Impala currently supports:

- Copy data from another table using SELECT query. In Impala 1.2.1 and higher, you can combine CREATE TABLE and INSERT operations into a single step with the CREATE TABLE AS SELECT syntax, which bypasses the actual INSERT keyword.
- An optional WITH clause before the INSERT keyword, to define a subquery referenced in the SELECT portion.
- Create one or more new rows using constant expressions through VALUES clause. (The VALUES clause was added in Impala 1.0.1.)
- By default, the first column of each newly inserted row goes into the first column of the table, the second column into the second column, and so on.

  You can also specify the columns to be inserted, an arbitrarily ordered subset of the columns in the destination table, by specifying a column list immediately after the name of the destination table. This feature lets you adjust the inserted columns to match the layout of a SELECT statement, rather than the other way around. (This feature was added in Impala 1.1.)

  The number of columns mentioned in the column list (known as the "column permutation") must match the number of columns in the SELECT list or the VALUES tuples. The order of columns in the column permutation can be different than in the underlying table, and the columns of each input row are reordered to match. If the number of columns in the column permutation is less than in the destination table, all unmentioned columns are set to NULL.

- An optional hint clause immediately either before the SELECT keyword or after the INSERT keyword, to fine-tune the behavior when doing an INSERT ... SELECT operation into partitioned Parquet tables. The hint clause cannot be specified in multiple places. The hint keywords are [SHUFFLE] and [NOSHUFFLE], including the square brackets. Inserting into partitioned Parquet tables can be a resource-intensive operation because it potentially involves many files being written to HDFS simultaneously, and separate large memory buffers being allocated to buffer the data for each partition. For usage details, see Loading Data into Parquet Tables on page 644.



Note:

- Insert commands that partition or add files result in changes to Hive metadata. Because Impala uses Hive metadata, such changes may necessitate a metadata refresh. For more information, see the REFRESH function.
- Currently, Impala can only insert data into tables that use the text and Parquet formats. For other file formats, insert the data using Hive and use Impala to query it.
- As an alternative to the INSERT statement, if you have existing data files elsewhere in HDFS, the LOAD DATA statement can move those files into a table. This statement works with tables of any file format.

Statement type: DML (but still affected by SYNC_DDL query option)

Usage notes:

When you insert the results of an expression, particularly of a built-in function call, into a small numeric column such as INT, SMALLINT, TINYINT, or FLOAT, you might need to use a CAST( ) expression to coerce values into the appropriate type. Impala does not automatically convert from a larger type to a smaller one. For example, to insert cosine values into a FLOAT column, write CAST(COS(angle) AS FLOAT) in the INSERT statement to make the conversion explicit.

File format considerations:

Because Impala can read certain file formats that it cannot write, the INSERT statement does not work for all kinds of Impala tables. See How Impala Works with Hadoop File Formats on page 634 for details about what file formats are supported by the INSERT statement.

Any INSERT statement for a Parquet table requires enough free space in the HDFS filesystem to write one block. Because Parquet data files use a block size of 1 GB by default, an INSERT might fail (even for a very small amount of data) if your HDFS is running low on space.

If you connect to different Impala nodes within an impala-shell session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.



Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following example sets up new tables with the same definition as the TAB1 table from the Tutorial section, using different file formats, and demonstrates inserting data into the tables created with the STORED AS TEXTFILE and STORED AS PARQUET clauses:

```
CREATE DATABASE IF NOT EXISTS file_formats;
USE file_formats;

DROP TABLE IF EXISTS text_table;
CREATE TABLE text_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS TEXTFILE;

DROP TABLE IF EXISTS parquet_table;
CREATE TABLE parquet_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS PARQUET;
```

With the INSERT INTO TABLE syntax, each new set of inserted rows is appended to any existing data in the table. This is how you would record small amounts of data that arrive continuously, or ingest new batches of data alongside the existing data. For example, after running 2 INSERT INTO TABLE statements with 5 rows each, the table contains 10 rows total:

```
[localhost:21000] > insert into table text_table select * from default.tab1;
Inserted 5 rows in 0.41s

[localhost:21000] > insert into table text_table select * from default.tab1;
Inserted 5 rows in 0.46s

[localhost:21000] > select count(*) from text_table;
+----------+
| count(*) |
+----------+
| 10       |
```

```
+----------+
Returned 1 row(s) in 0.26s
```

With the INSERT OVERWRITE TABLE syntax, each new set of inserted rows replaces any existing data in the table. This is how you load data to query in a data warehousing scenario where you analyze just the data for a particular day, quarter, and so on, discarding the previous data each time. You might keep the entire set of data in one raw table, and transfer and transform certain rows into a more compact and efficient form to perform intensive analysis on that subset.

For example, here we insert 5 rows into a table using the INSERT INTO clause, then replace the data by inserting 3 rows with the INSERT OVERWRITE clause. Afterward, the table only contains the 3 rows from the final INSERT statement.

```
[localhost:21000] > insert into table parquet_table select * from default.tab1;
Inserted 5 rows in 0.35s

[localhost:21000] > insert overwrite table parquet_table select * from default.tab1
limit 3;
Inserted 3 rows in 0.43s
[localhost:21000] > select count(*) from parquet_table;
+----------+
| count(*) |
+----------+
| 3        |
+----------+
Returned 1 row(s) in 0.43s
```

The VALUES clause lets you insert one or more rows by specifying constant values for all the columns. The number, types, and order of the expressions must match the table definition.



Note: The INSERT ... VALUES technique is not suitable for loading large quantities of data into HDFS-based tables, because the insert operations cannot be parallelized, and each one produces a separate data file. Use it for setting up small dimension tables or tiny amounts of data for experimenting with SQL syntax, or with HBase tables. Do not use it for large ETL jobs or benchmark tests for load operations. Do not run scripts with thousands of INSERT ... VALUES statements that insert a single row each time. If you do run INSERT ... VALUES operations to load data into a staging table as one stage in an ETL pipeline, include multiple row values if possible within each VALUES clause, and use a separate database to make cleanup easier if the operation does produce many tiny files.

The following example shows how to insert one row or multiple rows, with expressions of different types, using literal values, expressions, and function return values:

```
create table val_test_1 (c1 int, c2 float, c3 string, c4 boolean, c5 timestamp);
insert into val_test_1 values (100, 99.9/10, 'abc', true, now());
create table val_test_2 (id int, token string);
insert overwrite val_test_2 values (1, 'a'), (2, 'b'), (-1,'xyzzy');
```

These examples show the type of “not implemented” error that you see when attempting to insert data into a table with a file format that Impala currently does not write to:

```
DROP TABLE IF EXISTS sequence_table;
CREATE TABLE sequence_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS SEQUENCEFILE;

DROP TABLE IF EXISTS rc_table;
CREATE TABLE rc_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS RCFILE;

[localhost:21000] > insert into table rc_table select * from default.tab1;
Remote error
Backend 0:RC_FILE not implemented.
```

```
[localhost:21000] > insert into table sequence_table select * from default.tab1;
Remote error
Backend 0:SEQUENCE_FILE not implemented.
```

The following examples show how you can copy the data in all the columns from one table to another, copy the data from only some columns, or specify the columns in the select list in a different order than they actually appear in the table:

```
-- Start with 2 identical tables.
create table t1 (c1 int, c2 int);
create table t2 like t1;

-- If there is no () part after the destination table name,
-- all columns must be specified, either as * or by name.
insert into t2 select * from t1;
insert into t2 select c1, c2 from t1;

-- With the () notation following the destination table name,
-- you can omit columns (all values for that column are NULL
-- in the destination table), and/or reorder the values
-- selected from the source table. This is the "column permutation" feature.
insert into t2 (c1) select c1 from t1;
insert into t2 (c2, c1) select c1, c2 from t1;

-- The column names can be entirely different in the source and destination tables.
-- You can copy any columns, not just the corresponding ones, from the source table.
-- But the number and type of selected columns must match the columns mentioned in the
 () part.
alter table t2 replace columns (x int, y int);
insert into t2 (y) select c1 from t1;
```

Sorting considerations: Although you can specify an `ORDER BY` clause in an `INSERT ... SELECT` statement, any `ORDER BY` clause is ignored and the results are not necessarily sorted. An `INSERT ... SELECT` operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

Concurrency considerations: Each `INSERT` operation creates new data files with unique names, so you can run multiple `INSERT INTO` statements simultaneously without filename conflicts. While data is being inserted into an Impala table, the data is staged temporarily in a subdirectory inside the data directory; during this period, you cannot issue queries against that table in Hive. If an `INSERT` operation fails, the temporary data file and the subdirectory could be left behind in the data directory. If so, remove the relevant subdirectory and any data files it contains manually, by issuing an `hdfs dfs -rm -r` command, specifying the full path of the work subdirectory, whose name ends in `_dir`.

VALUES Clause

The VALUES clause is a general-purpose way to specify the columns of one or more rows, typically within an `INSERT` statement.



Note: The `INSERT ... VALUES` technique is not suitable for loading large quantities of data into HDFS-based tables, because the insert operations cannot be parallelized, and each one produces a separate data file. Use it for setting up small dimension tables or tiny amounts of data for experimenting with SQL syntax, or with HBase tables. Do not use it for large ETL jobs or benchmark tests for load operations. Do not run scripts with thousands of `INSERT ... VALUES` statements that insert a single row each time. If you do run `INSERT ... VALUES` operations to load data into a staging table as one stage in an ETL pipeline, include multiple row values if possible within each VALUES clause, and use a separate database to make cleanup easier if the operation does produce many tiny files.

The following examples illustrate:

- How to insert a single row using a VALUES clause.
- How to insert multiple rows using a VALUES clause.

- How the row or rows from a VALUES clause can be appended to a table through INSERT INTO, or replace the contents of the table through INSERT OVERWRITE.
- How the entries in a VALUES clause can be literals, function results, or any other kind of expression. See Literals on page 167 for the notation to use for literal values, especially String Literals on page 168 for quoting and escaping conventions for strings. See SQL Operators on page 171 and Impala Built-In Functions on page 391 for other things you can include in expressions with the VALUES clause.

```
[localhost:21000] > describe val_example;
Query: describe val_example
Query finished, fetching results ...
+-------+---------+---------+
| name  | type    | comment |
+-------+---------+---------+
| id    | int     |         |
| col_1 | boolean |         |
| col_2 | double  |         |
+-------+---------+---------+

[localhost:21000] > insert into val_example values (1,true,100.0);
Inserted 1 rows in 0.30s
[localhost:21000] > select * from val_example;
+----+-------+-------+
| id | col_1 | col_2 |
+----+-------+-------+
| 1  | true  | 100   |
+----+-------+-------+

[localhost:21000] > insert overwrite val_example values (10,false,pow(2,5)),
(50,true,10/3);
Inserted 2 rows in 0.16s
[localhost:21000] > select * from val_example;
+----+-------+-------------------+
| id | col_1 | col_2             |
+----+-------+-------------------+
| 10 | false | 32                |
| 50 | true  | 3.333333333333333 |
+----+-------+-------------------+
```

When used in an INSERT statement, the Impala VALUES clause can specify some or all of the columns in the destination table, and the columns can be specified in a different order than they actually appear in the table. To specify a different set or order of columns than in the table, use the syntax:

```
INSERT INTO destination
  (col_x, col_y, col_z)
  VALUES
  (val_x, val_y, val_z);
```

Any columns in the table that are not listed in the INSERT statement are set to NULL.

HDFS considerations:

Impala physically writes all inserted files under the ownership of its default user, typically impala. Therefore, this user must have HDFS write permission in the corresponding table directory.

The permission requirement is independent of the authorization performed by the Sentry framework. (If the connected user is not authorized to insert into a table, Sentry blocks that operation immediately, regardless of the privileges available to the impala user.) Files created by Impala are not owned by and do not inherit permissions from the connected user.

The number of data files produced by an INSERT statement depends on the size of the cluster, the number of data blocks that are processed, the partition key columns in a partitioned table, and the mechanism Impala uses for dividing the work in parallel. Do not assume that an INSERT statement will produce some particular number of output files. In case of performance issues with data written by Impala, check that the output files do not suffer from issues such as many tiny files or many tiny partitions. (In the Hadoop context, even files or partitions of a few tens of megabytes are considered “tiny”.)

The INSERT statement has always left behind a hidden work directory inside the data directory of the table. Formerly, this hidden work directory was named .impala_insert_staging . In Impala 2.0.1 and later, this directory name is changed to _impala_insert_staging . (While HDFS tools are expected to treat names beginning either with underscore and dot as hidden, in practice names beginning with an underscore are more widely supported.) If you have any scripts, cleanup jobs, and so on that rely on the name of this work directory, adjust them to use the new name.

HBase considerations:

You can use the INSERT statement with HBase tables as follows:

- You can insert a single row or a small set of rows into an HBase table with the INSERT ... VALUES syntax. This is a good use case for HBase tables with Impala, because HBase tables are not subject to the same kind of fragmentation from many small insert operations as HDFS tables are.
- You can insert any number of rows at once into an HBase table using the INSERT ... SELECT syntax.
- If more than one inserted row has the same value for the HBase key column, only the last inserted row with that value is visible to Impala queries. You can take advantage of this fact with INSERT ... VALUES statements to effectively update rows one at a time, by inserting new rows with the same key values as existing rows. Be aware that after an INSERT ... SELECT operation copying from an HDFS table, the HBase table might contain fewer rows than were inserted, if the key column in the source table contained duplicate values.
- You cannot INSERT OVERWRITE into an HBase table. New rows are always appended.
- When you create an Impala or Hive table that maps to an HBase table, the column order you specify with the INSERT statement might be different than the order you declare with the CREATE TABLE statement. Behind the scenes, HBase arranges the columns based on how they are divided into column families. This might cause a mismatch during insert operations, especially if you use the syntax INSERT INTO *hbase_table* SELECT * FROM *hdfs_table*. Before inserting data, verify the column order by issuing a DESCRIBE statement for the table, and adjust the order of the select list in the INSERT statement.

See Using Impala to Query HBase Tables on page 684 for more details about using Impala with HBase.

Amazon S3 considerations:

In CDH 5.8 / Impala 2.6 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Amazon Simple Storage Service (S3). The syntax of the DML statements is the same as for any other tables, because the S3 location for tables and partitions is specified by an s3a:// prefix in the LOCATION attribute of CREATE TABLE or ALTER TABLE statements. If you bring data into S3 using the normal S3 transfer mechanisms instead of Impala DML statements, issue a REFRESH statement for the table before using Impala to query the S3 data.

Because of differences between S3 and traditional filesystems, DML operations for S3 tables can take longer than for tables on HDFS. For example, both the LOAD DATA statement and the final stage of the INSERT and CREATE TABLE AS SELECT statements involve moving files from one directory to another. (In the case of INSERT and CREATE TABLE AS SELECT, the files are moved from a temporary staging directory to the final destination directory.) Because S3 does not support a "rename" operation for existing objects, in these cases Impala actually copies the data files from one location to another and then removes the original files. In CDH 5.8 / Impala 2.6, the S3_SKIP_INSERT_STAGING query option provides a way to speed up INSERT statements for S3 tables and partitions, with the tradeoff that a problem during statement execution could leave data in an inconsistent state. It does not apply to INSERT OVERWRITE or LOAD DATA statements. See S3_SKIP_INSERT_STAGING Query Option (CDH 5.8 or higher only) on page 358 for details.

See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

ADLS considerations:

In CDH 5.12 / Impala 2.9 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Azure Data Lake Store (ADLS). ADLS Gen2 is supported in CDH 6.1 and higher.

In theCREATE TABLE or ALTER TABLE statements, specify the ADLS location for tables and partitions with the `adl://` prefix for ADLS Gen1 and `abfs://` or `abfss://` for ADLS Gen2 in the `LOCATION` attribute.

If you bring data into ADLS using the normal ADLS transfer mechanisms instead of Impala DML statements, issue a `REFRESH` statement for the table before using Impala to query the ADLS data.

See Using Impala with the Azure Data Lake Store (ADLS) on page 701 for details about reading and writing ADLS data with Impala.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts. See http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/sg_redaction.html for details.

Cancellation: Can be cancelled. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the **Cancel** button from the **Watch** page in Hue, **Actions > Cancel** from the **Queries** list in Cloudera Manager, or **Cancel** from the list of in-flight queries (for a particular node) on the **Queries** tab in the Impala web UI (port 25000).

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read permission for the files in the source directory of an `INSERT ... SELECT` operation, and write permission for all affected directories in the destination table. (An `INSERT` operation could write files to multiple different HDFS directories if the destination table is partitioned.) This user must also have write permission to create a temporary work directory in the top-level HDFS directory of the destination table. An `INSERT OVERWRITE` operation does not require write permission on the original data files in the table, only on the table directories themselves.

Restrictions:

For `INSERT` operations into `CHAR` or `VARCHAR` columns, you must cast all `STRING` literals or expressions returning `STRING` to to a `CHAR` or `VARCHAR` type with the appropriate length.

Related startup options:

By default, if an `INSERT` statement creates any new subdirectories underneath a partitioned table, those subdirectories are assigned default HDFS permissions for the `impala` user. To make each subdirectory have the same permissions as its parent directory in HDFS, specify the `--insert_inherit_permissions` startup option for the `impalad` daemon.

Inserting Into Partitioned Tables with PARTITION Clause

For a partitioned table, the optional `PARTITION` clause identifies which partition or partitions the values are inserted into.

All examples in this section will use the table declared as below:

```
CREATE TABLE t1 (w INT) PARTITIONED BY (x INT, y STRING);
```

Static Partition Inserts

In a static partition insert where a partition key column is given a constant value, such as `PARTITION (year=2012, month=2)`, the rows are inserted with the same values specified for those partition key columns.

The number of columns in the `SELECT` list must equal the number of columns in the column permutation.

The `PARTITION` clause must be used for static partitioning inserts.

Example:

The following statement will insert the some_other_table.c1 values for the w column, and all the rows inserted will have the same x value of 10, and the same y value of 'a'.

```
INSERT INTO t1 PARTITION (x=10, y='a')
            SELECT c1 FROM some_other_table;
```

Dynamic Partition Inserts

In a dynamic partition insert where a partition key column is in the INSERT statement but not assigned a value, such as in PARTITION (year, region)(both columns unassigned) or PARTITION(year, region='CA') (year column unassigned), the unassigned columns are filled in with the final columns of the SELECT or VALUES clause. In this case, the number of columns in the SELECT list must equal the number of columns in the column permutation plus the number of partition key columns not assigned a constant value.

See Static and Dynamic Partitioning Clauses for examples and performance characteristics of static and dynamic partitioned inserts.

The following rules apply to dynamic partition inserts.

- The columns are bound in the order they appear in the INSERT statement.

  The table below shows the values inserted with the INSERT statements of different column orders.

	Column w Value	Column x Value	Column y Value
INSERT INTO t1 (w, x, y) VALUES (1, 2, 'c');	1	2	'c'
INSERT INTO t1 (x,w) PARTITION (y) VALUES (1, 2, 'c');	2	1	'c'

- When a partition clause is specified but the non-partition columns are not specified in the INSERT statement, as in the first example below, the non-partition columns are treated as though they had been specified before the PARTITION clause in the SQL.

  Example: These three statements are equivalent, inserting 1 to w, 2 to x, and 'c' to y columns.

```
INSERT INTO t1 PARTITION (x,y) VALUES (1, 2, 'c');
INSERT INTO t1 (w) PARTITION (x, y) VALUES (1, 2, 'c');
INSERT INTO t1 PARTITION (x, y='c') VALUES (1, 2);
```

- The PARTITION clause is not required for dynamic partition, but all the partition columns must be explicitly present in the INSERT statement in the column list or in the PARTITION clause. The partition columns cannot be defaulted to NULL.

  Example:

  The following statements are valid because the partition columns, x and y, are present in the INSERT statements, either in the PARTITION clause or in the column list.

```
INSERT INTO t1 PARTITION (x,y) VALUES (1, 2, 'c');
INSERT INTO t1 (w, x) PARTITION (y) VALUES (1, 2, 'c');
```

  The following statement is not valid for the partitioned table as defined above because the partition columns, x and y, are not present in the INSERT statement.

```
INSERT INTO t1 VALUES (1, 2, 'c');
```

- If partition columns do not exist in the source table, you can specify a specific value for that column in the PARTITION clause.

Example: The `source` table only contains the column `w` and `y`. The value, `20`, specified in the `PARTITION` clause, is inserted into the `x` column.

```
INSERT INTO t1 PARTITION (x=20, y) SELECT * FROM source;
```

INVALIDATE METADATA Statement

The `INVALIDATE METADATA` statement marks the metadata for one or all tables as stale. The next time the Impala service performs a query against a table whose metadata is invalidated, Impala reloads the associated metadata before the query proceeds. As this is a very expensive operation compared to the incremental metadata update done by the `REFRESH` statement, when possible, prefer `REFRESH` rather than `INVALIDATE METADATA`.

`INVALIDATE METADATA` is required when the following changes are made outside of Impala, in Hive and other Hive client, such as SparkSQL:

- Metadata of existing tables changes.
- New tables are added, and Impala will use the tables.
- The SERVER or DATABASE level Sentry privileges are changed from outside of Impala.
- Block metadata changes, but the files remain the same (HDFS rebalance).
- UDF jars change.
- Some tables are no longer queried, and you want to remove their metadata from the catalog and coordinator caches to reduce memory requirements.

No `INVALIDATE METADATA` is needed when the changes are made by `impalad`.

See Overview of Impala Metadata and the Metastore on page 22 for the information about the way Impala uses metadata and how it shares the same metastore database as Hive.

Once issued, the `INVALIDATE METADATA` statement cannot be cancelled.

Syntax:

```
INVALIDATE METADATA [[db_name.]table_name]
```

If there is no table specified, the cached metadata for all tables is flushed and synced with Hive Metastore (HMS). If tables were dropped from the HMS, they will be removed from the catalog, and if new tables were added, they will show up in the catalog.

If you specify a table name, only the metadata for that one table is flushed and synced with the HMS.

Usage notes:

To return accurate query results, Impala need to keep the metadata current for the databases and tables queried. Therefore, if some other entity modifies information used by Impala in the metastore, the information cached by Impala must be updated via `INVALIDATE METADATA` or `REFRESH`.

`INVALIDATE METADATA` and `REFRESH` are counterparts:

- `INVALIDATE METADATA` is an asynchronous operation that simply discards the loaded metadata from the catalog and coordinator caches. After that operation, the catalog and all the Impala coordinators know only about the existence of databases and tables, nothing more. Metadata loading for tables is triggered by any subsequent queries.
- `REFRESH` reloads the metadata in sync with the coordinator executing the statement. Set the Impala SYNC_DLL option to `true` to achieve a fully synchronous metadata reload. `REFRESH` performs a lightweight reloading, not the full metadata reloading that occurs after a table has been invalidated. `REFRESH` cannot detect changes in block locations triggered by operations like HDFS balancer, which can cause remote reads during query execution that impact performance.

Use `REFRESH` after invalidating a specific table to separate the metadata load from the first query that's run against that table.

Examples:

This example illustrates creating a new database and new table in Hive, then doing an INVALIDATE METADATA statement in Impala using the fully qualified table name, after which both the new table and the new database are visible to Impala.

Before the INVALIDATE METADATA statement was issued, Impala would give a "not found" error if you tried to refer to those database or table names.

```
$ hive
hive> CREATE DATABASE new_db_from_hive;
hive> CREATE TABLE new_db_from_hive.new_table_from_hive (x INT);
hive> quit;

$ impala-shell
> REFRESH new_db_from_hive.new_table_from_hive;
ERROR: AnalysisException: Database does not exist: new_db_from_hive

> INVALIDATE METADATA new_db_from_hive.new_table_from_hive;

> SHOW DATABASES LIKE 'new*';
+--------------------+
| new_db_from_hive   |
+--------------------+

> SHOW TABLES IN new_db_from_hive;
+--------------------+
| new_table_from_hive |
+--------------------+
```

Use the REFRESH statement for incremental metadata update.

```
> REFRESH new_table_from_hive;
```

For more examples of using INVALIDATE METADATA with a combination of Impala and Hive operations, see Switching Back and Forth Between Impala and Hive on page 56.

HDFS considerations:

By default, the INVALIDATE METADATA command checks HDFS permissions of the underlying data files and directories, caching this information so that a statement can be cancelled immediately if for example the `impala` user does not have permission to write to the data directory for the table. (This checking does not apply when the `catalogd` configuration option `--load_catalog_in_background` is set to `false`, which it is by default.) Impala reports any lack of write permissions as an INFO message in the log file.

If you change HDFS permissions to make data readable or writeable by the Impala user, issue another INVALIDATE METADATA to make Impala aware of the change.

Kudu considerations:

By default, much of the metadata for Kudu tables is handled by the underlying storage layer. Kudu tables have less reliance on the Metastore database, and require less metadata caching on the Impala side. For example, information about partitions in Kudu tables is managed by Kudu, and Impala does not cache any block locality metadata for Kudu tables. If the Kudu service is not integrated with the Hive Metastore, Impala will manage Kudu table metadata in the Hive Metastore.

The REFRESH and INVALIDATE METADATA statements are needed less frequently for Kudu tables than for HDFS-backed tables. Neither statement is needed when data is added to, removed, or updated in a Kudu table, even if the changes are made directly to Kudu through a client program using the Kudu API. Run REFRESH *table_name* or INVALIDATE METADATA *table_name* for a Kudu table only after making a change to the Kudu table schema, such as adding or dropping a column.

Related information:

Overview of Impala Metadata and the Metastore on page 22, REFRESH Statement on page 291

LOAD DATA Statement

The LOAD DATA statement streamlines the ETL process for an internal Impala table by moving a data file or all the data files in a directory from an HDFS location into the Impala data directory for that table.

Syntax:

```
LOAD DATA INPATH 'hdfs_file_or_directory_path' [OVERWRITE] INTO TABLE tablename
  [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

When the LOAD DATA statement operates on a partitioned table, it always operates on one partition at a time. Specify the PARTITION clauses and list all the partition key columns, with a constant value specified for each.

Statement type: DML (but still affected by SYNC_DDL query option)

Usage notes:

- The loaded data files are moved, not copied, into the Impala data directory.
- You can specify the HDFS path of a single file to be moved, or the HDFS path of a directory to move all the files inside that directory. You cannot specify any sort of wildcard to take only some of the files from a directory. When loading a directory full of data files, keep all the data files at the top level, with no nested directories underneath.
- Currently, the Impala LOAD DATA statement only imports files from HDFS, not from the local filesystem. It does not support the LOCAL keyword of the Hive LOAD DATA statement. You must specify a path, not an hdfs:// URI.
- In the interest of speed, only limited error checking is done. If the loaded files have the wrong file format, different columns than the destination table, or other kind of mismatch, Impala does not raise any error for the LOAD DATA statement. Querying the table afterward could produce a runtime error or unexpected results. Currently, the only checking the LOAD DATA statement does is to avoid mixing together uncompressed and LZO-compressed text files in the same table.
- When you specify an HDFS directory name as the LOAD DATA argument, any hidden files in that directory (files whose names start with a .) are not moved to the Impala data directory.
- The operation fails if the source directory contains any non-hidden directories. Prior to CDH 5.7 / Impala 2.5 if the source directory contained any subdirectory, even a hidden one such as _impala_insert_staging, the LOAD DATA statement would fail. In CDH 5.7 / Impala 2.5 and higher, LOAD DATA ignores hidden subdirectories in the source directory, and only fails if any of the subdirectories are non-hidden.
- The loaded data files retain their original names in the new location, unless a name conflicts with an existing data file, in which case the name of the new file is modified slightly to be unique. (The name-mangling is a slight difference from the Hive LOAD DATA statement, which replaces identically named files.)
- By providing an easy way to transport files from known locations in HDFS into the Impala data directory structure, the LOAD DATA statement lets you avoid memorizing the locations and layout of HDFS directory tree containing the Impala databases and tables. (For a quick way to check the location of the data files for an Impala table, issue the statement DESCRIBE FORMATTED *table_name*.)
- The PARTITION clause is especially convenient for ingesting new data for a partitioned table. As you receive new data for a time period, geographic region, or other division that corresponds to one or more partitioning columns, you can load that data straight into the appropriate Impala data directory, which might be nested several levels down if the table is partitioned by multiple columns. When the table is partitioned, you must specify constant values for all the partitioning columns.

Complex type considerations:

Because Impala currently cannot create Parquet data files containing complex types (ARRAY, STRUCT, and MAP), the LOAD DATA statement is especially important when working with tables containing complex type columns. You create the Parquet data files outside Impala, then use either LOAD DATA, an external table, or HDFS-level file operations followed by REFRESH to associate the data files with the corresponding table. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types.

If you connect to different Impala nodes within an impala-shell session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes. See SYNC_DDL Query Option on page 361 for details.



Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

First, we use a trivial Python script to write different numbers of strings (one per line) into files stored in the doc_demo HDFS user account. (Substitute the path for your own HDFS user account when doing hdfs dfs operations like these.)

```
random_strings.py 1000 | hdfs dfs -put - /user/doc_demo/thousand_strings.txt
random_strings.py 100 | hdfs dfs -put - /user/doc_demo/hundred_strings.txt
random_strings.py 10 | hdfs dfs -put - /user/doc_demo/ten_strings.txt
```

Next, we create a table and load an initial set of data into it. Remember, unless you specify a STORED AS clause, Impala tables default to TEXTFILE format with Ctrl-A (hex 01) as the field delimiter. This example uses a single-column table, so the delimiter is not significant. For large-scale ETL jobs, you would typically use binary format data files such as Parquet or Avro, and load them into Impala tables that use the corresponding file format.

```
[localhost:21000] > create table t1 (s string);
[localhost:21000] > load data inpath '/user/doc_demo/thousand_strings.txt' into table
t1;
Query finished, fetching results ...
+----------------------------------------------------------+
| summary                                                  |
+----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 1 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.61s
[kilo2-202-961.cs1cloud.internal:21000] > select count(*) from t1;
Query finished, fetching results ...
+------+
| _c0  |
+------+
| 1000 |
+------+
Returned 1 row(s) in 0.67s
[localhost:21000] > load data inpath '/user/doc_demo/thousand_strings.txt' into table
t1;
ERROR: AnalysisException: INPATH location '/user/doc_demo/thousand_strings.txt' does
not exist.
```

As indicated by the message at the end of the previous example, the data file was moved from its original location. The following example illustrates how the data file was moved into the Impala data directory for the destination table, keeping its original filename:

```
$ hdfs dfs -ls /user/hive/warehouse/load_data_testing.db/t1
Found 1 items
-rw-r--r--   1 doc_demo doc_demo      13926 2013-06-26 15:40
/user/hive/warehouse/load_data_testing.db/t1/thousand_strings.txt
```

The following example demonstrates the difference between the INTO TABLE and OVERWRITE TABLE clauses. The table already contains 1000 rows. After issuing the LOAD DATA statement with the INTO TABLE clause, the table contains 100 more rows, for a total of 1100. After issuing the LOAD DATA statement with the OVERWRITE INTO TABLE clause, the former contents are gone, and now the table only contains the 10 rows from the just-loaded data file.

```
[localhost:21000] > load data inpath '/user/doc_demo/hundred_strings.txt' into table
t1;
Query finished, fetching results ...
+----------------------------------------------------------+
| summary                                                  |
+----------------------------------------------------------+
```

```
| Loaded 1 file(s). Total files in destination location: 2 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.24s
[localhost:21000] > select count(*) from t1;
Query finished, fetching results ...
+------+
| _c0  |
+------+
| 1100 |
+------+
Returned 1 row(s) in 0.55s
[localhost:21000] > load data inpath '/user/doc_demo/ten_strings.txt' overwrite into
table t1;
Query finished, fetching results ...
+----------------------------------------------------------+
| summary                                                  |
+----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 1 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.26s
[localhost:21000] > select count(*) from t1;
Query finished, fetching results ...
+-----+
| _c0 |
+-----+
| 10  |
+-----+
Returned 1 row(s) in 0.62s
```

Amazon S3 considerations:

In CDH 5.8 / Impala 2.6 and higher, the Impala DML statements (`INSERT`, `LOAD DATA`, and `CREATE TABLE AS SELECT`) can write data into a table or partition that resides in the Amazon Simple Storage Service (S3). The syntax of the DML statements is the same as for any other tables, because the S3 location for tables and partitions is specified by an `s3a://` prefix in the `LOCATION` attribute of `CREATE TABLE` or `ALTER TABLE` statements. If you bring data into S3 using the normal S3 transfer mechanisms instead of Impala DML statements, issue a `REFRESH` statement for the table before using Impala to query the S3 data.

Because of differences between S3 and traditional filesystems, DML operations for S3 tables can take longer than for tables on HDFS. For example, both the `LOAD DATA` statement and the final stage of the `INSERT` and `CREATE TABLE AS SELECT` statements involve moving files from one directory to another. (In the case of `INSERT` and `CREATE TABLE AS SELECT`, the files are moved from a temporary staging directory to the final destination directory.) Because S3 does not support a "rename" operation for existing objects, in these cases Impala actually copies the data files from one location to another and then removes the original files. In CDH 5.8 / Impala 2.6, the `S3_SKIP_INSERT_STAGING` query option provides a way to speed up `INSERT` statements for S3 tables and partitions, with the tradeoff that a problem during statement execution could leave data in an inconsistent state. It does not apply to `INSERT OVERWRITE` or `LOAD DATA` statements. See S3_SKIP_INSERT_STAGING Query Option (CDH 5.8 or higher only) on page 358 for details.

See Using Impala with the Amazon S3 Filesystem on page 692 for details about reading and writing S3 data with Impala.

ADLS considerations:

In CDH 5.12 / Impala 2.9 and higher, the Impala DML statements (`INSERT`, `LOAD DATA`, and `CREATE TABLE AS SELECT`) can write data into a table or partition that resides in the Azure Data Lake Store (ADLS). ADLS Gen2 is supported in CDH 6.1 and higher.

In the`CREATE TABLE` or `ALTER TABLE` statements, specify the ADLS location for tables and partitions with the `adl://` prefix for ADLS Gen1 and `abfs://` or `abfss://` for ADLS Gen2 in the `LOCATION` attribute.

If you bring data into ADLS using the normal ADLS transfer mechanisms instead of Impala DML statements, issue a `REFRESH` statement for the table before using Impala to query the ADLS data.

See Using Impala with the Azure Data Lake Store (ADLS) on page 701 for details about reading and writing ADLS data with Impala.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read and write permissions for the files in the source directory, and write permission for the destination directory.

Kudu considerations:

The `LOAD DATA` statement cannot be used with Kudu tables.

HBase considerations:

The `LOAD DATA` statement cannot be used with HBase tables.

Related information:

The `LOAD DATA` statement is an alternative to the `INSERT` statement. Use `LOAD DATA` when you have the data files in HDFS but outside of any Impala table.

The `LOAD DATA` statement is also an alternative to the `CREATE EXTERNAL TABLE` statement. Use `LOAD DATA` when it is appropriate to move the data files under Impala control rather than querying them from their original location. See External Tables on page 197 for information about working with external tables.

REFRESH Statement

The `REFRESH` statement reloads the metadata for the table from the metastore database and does an incremental reload of the file and block metadata from the HDFS NameNode. `REFRESH` is used to avoid inconsistencies between Impala and external metadata sources, namely Hive Metastore (HMS) and NameNodes.

The `REFRESH` statement is only required if you load data from outside of Impala. Updated metadata, as a result of running `REFRESH`, is broadcast to all Impala coordinators.

See Overview of Impala Metadata and the Metastore on page 22 for the information about the way Impala uses metadata and how it shares the same metastore database as Hive.

Syntax:

```
REFRESH [db_name.]table_name [PARTITION (key_col1=val1 [, key_col2=val2...])]
```

Usage notes:

The table name is a required parameter, and the table must already exist and be known to Impala.

Only the metadata for the specified table is reloaded.

Use the `REFRESH` statement to load the latest metastore metadata for a particular table after one of the following scenarios happens outside of Impala:

- Deleting, adding, or modifying files.

  For example, after loading new data files into the HDFS data directory for the table, appending to an existing HDFS file, inserting data from Hive via `INSERT` or `LOAD DATA`.

- Deleting, adding, or modifying partitions.

  For example, after issuing `ALTER TABLE` or other table-modifying SQL statement in Hive



Note:

In CDH 5.5 / Impala 2.3 and higher, the `ALTER TABLE table_name RECOVER PARTITIONS` statement is a faster alternative to `REFRESH` when you are only adding new partition directories through Hive or manual HDFS operations. See ALTER TABLE Statement on page 205 for details.

`INVALIDATE METADATA` and `REFRESH` are counterparts:

- `INVALIDATE METADATA` is an asynchronous operation that simply discards the loaded metadata from the catalog and coordinator caches. After that operation, the catalog and all the Impala coordinators know only about the existence of databases and tables, nothing more. Metadata loading for tables is triggered by any subsequent queries.
- `REFRESH` reloads the metadata in sync with the coordinator executing the statement. Set the Impala SYNC_DLL option to `true` to achieve a fully synchronous metadata reload. `REFRESH` performs a lightweight reloading, not the full metadata reloading that occurs after a table has been invalidated. `REFRESH` cannot detect changes in block locations triggered by operations like HDFS balancer, which can cause remote reads during query execution that impact performance.

Refreshing a single partition:

In CDH 5.9 / Impala 2.7 and higher, the `REFRESH` statement can apply to a single partition at a time, rather than the whole table. Include the optional `PARTITION (`*`partition_spec`*`)` clause and specify values for each of the partition key columns.

The following rules apply:

- The `PARTITION` clause of the `REFRESH` statement must include all the partition key columns.
- The order of the partition key columns does not have to match the column order in the table.
- Specifying a nonexistent partition does not cause an error.
- The partition can be one that Impala created and is already aware of, or a new partition created through Hive.

The following examples demonstrates the above rules.

```
-- Partition doesn't exist.
REFRESH p2 PARTITION (y=0, z=3);
REFRESH p2 PARTITION (y=0, z=-1)

-- Key columns specified in a different order than the table definition.
REFRESH p2 PARTITION (z=1, y=0)

-- Incomplete partition spec causes an error.
REFRESH p2 PARTITION (y=0)
ERROR: AnalysisException: Items in partition spec must exactly match the partition
columns in the table definition: default.p2 (1 vs 2)
```

For examples of using `REFRESH` and `INVALIDATE METADATA` with a combination of Impala and Hive operations, see Switching Back and Forth Between Impala and Hive on page 56.

Related impala-shell options:

Due to the expense of reloading the metadata for all tables, the `impala-shell -r` option is not recommended.

HDFS considerations:

All HDFS and Sentry permissions and privilege requirements are the same whether you refresh the entire table or a single partition.

The `REFRESH` statement checks HDFS permissions of the underlying data files and directories, caching this information so that a statement can be cancelled immediately if for example the `impala` user does not have permission to write to the data directory for the table. Impala reports any lack of write permissions as an `INFO` message in the log file.

If you change HDFS permissions to make data readable or writeable by the Impala user, issue another `REFRESH` to make Impala aware of the change.

Kudu considerations:

By default, much of the metadata for Kudu tables is handled by the underlying storage layer. Kudu tables have less reliance on the Metastore database, and require less metadata caching on the Impala side. For example, information about partitions in Kudu tables is managed by Kudu, and Impala does not cache any block locality metadata for Kudu tables. If the Kudu service is not integrated with the Hive Metastore, Impala will manage Kudu table metadata in the Hive Metastore.

The REFRESH and INVALIDATE METADATA statements are needed less frequently for Kudu tables than for HDFS-backed tables. Neither statement is needed when data is added to, removed, or updated in a Kudu table, even if the changes are made directly to Kudu through a client program using the Kudu API. Run REFRESH *table_name* or INVALIDATE METADATA *table_name* for a Kudu table only after making a change to the Kudu table schema, such as adding or dropping a column.

Related information:

Overview of Impala Metadata and the Metastore on page 22, INVALIDATE METADATA Statement on page 286

REFRESH AUTHORIZATION Statement

The REFRESH AUTHORIZATION statement explicitly refreshes authorization data, including privileges and principals. When there is an external update to authorization metadata, use this statement to force Impala to refresh its authorization data without having to wait for the Sentry polling or run INVALIDATE METADATA.

Once issued, the REFRESH AUTHORIZATION statement cannot be cancelled.

See Impala Authorization for information on enabling and using authorization in Impala.

Syntax:

```
REFRESH AUTHORIZATION
```

Usage notes: If authorization is not enabled, Impala returns an error.

Added in: Impala 3.2

REFRESH FUNCTIONS Statement

In CDH 5.12 / Impala 2.9 and higher, you can run the REFRESH FUNCTIONS statement to refresh the user-defined functions (UDFs) created outside of Impala. For example, you can add Java-based UDFs to the metastore database through the Hive CREATE FUNCTION statements and make those UDFs visible to Impala at the database level by subsequently running REFRESH FUNCTIONS.

The REFRESH FUNCTIONS statement is only required if you create or modify UDFs from outside of Impala. Updated metadata, as a result of running REFRESH FUNCTIONS, is broadcast to all Impala coordinators.

Once issued, the REFRESH FUNCTIONS statement cannot be cancelled.

Syntax:

REFRESH FUNCTIONS *db_name*

REVOKE Statement (CDH 5.2 or higher only)

The REVOKE statement revokes roles or privileges on a specified object from groups.

Syntax:

```
REVOKE ROLE role_name FROM GROUP group_name

REVOKE [GRANT OPTION FOR] privilege ON object_type object_name
  FROM [ROLE] role_name

privilege ::= ALL | CREATE | INSERT | REFRESH | SELECT | SELECT(column_name)

object_type ::= SERVER | URI | DATABASE | TABLE
```

See GRANT Statement (CDH 5.2 or higher only) for the required privileges and the scope for SQL operations.

The ALL privilege is a distinct privilege and not a union of all other privileges. Revoking SELECT, INSERT, etc. from a role that only has the ALL privilege has no effect. To reduce the privileges of that role you must REVOKE ALL and GRANT the desired privileges.

You cannot revoke a privilege granted with the `WITH GRANT OPTION`. If a privilege is granted with the `WITH GRANT OPTION`, first revoke the grant option, and then revoke the privilege.

You cannot revoke the GRANT privilege from a role without also revoking the privilege. To revoke the GRANT privilege, revoke the privilege that it applies to and then grant that privilege again without the WITH GRANT OPTION clause.

You cannot revoke a privilege granted with the `WITH GRANT OPTION`. If a privilege is granted with the `WITH GRANT OPTION`, first revoke the grant option, and then revoke the privilege.

For example:

```
GRANT ALL ON SERVER TO ROLE foo_role;
...
REVOKE GRANT OPTION FOR ALL ON SERVER FROM ROLE foo_role;
REVOKE ALL ON SERVER FROM ROLE foo_role;
```

Typically, the object name is an identifier. For URIs, it is a string literal.

The ability to grant or revoke SELECT privilege on specific columns is available in CDH 5.5 / Impala 2.3 and higher. See Hive SQL Syntax for Use with Sentry for details. for details.

Required privileges:

Only Sentry administrative users, users who belong to the groups defined in `sentry.service.admin.group` of the Sentry configuration can revoke a role from a group.

Compatibility:

- The Impala REVOKE statement is available in CDH 5.2 / Impala 2.0 and higher.
- In CDH 5.1 / Impala 1.4 and higher, Impala makes use of any roles and privileges specified by the GRANT and REVOKE statements in Hive, when your system is configured to use the Sentry service instead of the file-based policy mechanism.
- The Impala REVOKE statements do not require the ROLE keyword to be repeated before each role name, unlike the equivalent Hive statements.
- Currently, each Impala GRANT or REVOKE statement can only grant or revoke a single privilege to or from a single role.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Kudu considerations:

Access to Kudu tables must be granted to and revoked from roles with the following considerations:

- Only users with the ALL privilege on SERVER can create external Kudu tables.
- The ALL privileges on SERVER is required to specify the `kudu.master_addresses` property in the CREATE TABLE statements for managed tables as well as external tables.
- Access to Kudu tables is enforced at the table level and at the column level.
- The SELECT- and INSERT-specific permissions are supported.
- The DELETE, UPDATE, and UPSERT operations require the ALL privilege.

Because non-SQL APIs can access Kudu data without going through Sentry authorization, currently the Sentry support is considered preliminary and subject to change.

Related information:

Enabling Sentry Authorization for Impala on page 87, GRANT Statement (CDH 5.2 or higher only) on page 273 CREATE ROLE Statement (CDH 5.2 or higher only) on page 234, DROP ROLE Statement (CDH 5.2 or higher only) on page 265, SHOW Statement on page 363

SELECT Statement

The SELECT statement performs queries, retrieving data from one or more tables and producing result sets consisting of rows and columns.

The Impala INSERT statement also typically ends with a SELECT statement, to define data to copy from one table to another.

Syntax:

```
[WITH name AS (select_expression) [, ...] ]
SELECT
  [ALL | DISTINCT]
  [STRAIGHT_JOIN]
  expression [, expression ...]
FROM table_reference [, table_reference ...]
[[FULL | [LEFT | RIGHT] INNER | [LEFT | RIGHT] OUTER | [LEFT | RIGHT] SEMI | [LEFT |
RIGHT] ANTI | CROSS]
  JOIN table_reference
  [ON join_equality_clauses | USING (col1[, col2 ...]] ...
WHERE conditions
GROUP BY { column | expression [, ...] }
HAVING conditions
ORDER BY { column | expression [ASC | DESC] [NULLS FIRST | NULLS LAST] [, ...] }
LIMIT expression [OFFSET expression]
[UNION [ALL] select_statement] ...]

table_reference := { table_name | (subquery) }
  [ TABLESAMPLE SYSTEM(percentage) [REPEATABLE(seed)] ]
```

Impala SELECT queries support:

- SQL scalar data types: BOOLEAN, TINYINT, SMALLINT, INT, BIGINT, DECIMAL FLOAT, DOUBLE, TIMESTAMP, STRING, VARCHAR, CHAR.
- The complex data types ARRAY, STRUCT, and MAP, are available in CDH 5.5 / Impala 2.3 and higher. Queries involving these types typically involve special qualified names using dot notation for referring to the complex column fields, and join clauses for bringing the complex columns into the result set. See Complex Types (CDH 5.5 or higher only) on page 139 for details.
- An optional WITH clause before the SELECT keyword, to define a subquery whose name or column names can be referenced from later in the main query. This clause lets you abstract repeated clauses, such as aggregation functions, that are referenced multiple times in the same query.
- Subqueries in a FROM clause. In CDH 5.2 / Impala 2.0 and higher, subqueries can also go in the WHERE clause, for example with the IN(), EXISTS, and NOT EXISTS operators.
- WHERE, GROUP BY, HAVING clauses.
- ORDER BY. Prior to Impala 1.4.0, Impala required that queries using an ORDER BY clause also include a LIMIT clause. In Impala 1.4.0 and higher, this restriction is lifted; sort operations that would exceed the Impala memory limit automatically use a temporary disk work area to perform the sort.
- Impala supports a wide variety of JOIN clauses. Left, right, semi, full, and outer joins are supported in all Impala versions. The CROSS JOIN operator is available in Impala 1.2.2 and higher. During performance tuning, you can override the reordering of join clauses that Impala does internally by including the keyword STRAIGHT_JOIN immediately after the SELECT and any DISTINCT or ALL keywords.

  See Joins in Impala SELECT Statements on page 296 for details and examples of join queries.

- UNION ALL.
- LIMIT.
- External tables.
- Relational operators such as greater than, less than, or equal to.
- Arithmetic operators such as addition or subtraction.
- Logical/Boolean operators AND, OR, and NOT. Impala does not support the corresponding symbols &&, ||, and !.
- Common SQL built-in functions such as COUNT, SUM, CAST, LIKE, IN, BETWEEN, and COALESCE. Impala specifically supports built-ins described in Impala Built-In Functions on page 391.

- In CDH 5.12 / Impala 2.9 and higher, an optional TABLESAMPLE clause immediately after a table reference, to specify that the query only processes a specified percentage of the table data. See TABLESAMPLE Clause on page 315 for details.

Impala queries ignore files with extensions commonly used for temporary work files by Hadoop tools. Any files with extensions `.tmp` or `.copying` are not considered part of the Impala table. The suffix matching is case-insensitive, so for example Impala ignores both `.copying` and `.COPYING` suffixes.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts. See http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/sg_redaction.html for details.

Amazon S3 considerations:

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Cancellation: Can be cancelled. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the **Cancel** button from the **Watch** page in Hue, **Actions > Cancel** from the **Queries** list in Cloudera Manager, or **Cancel** from the list of in-flight queries (for a particular node) on the **Queries** tab in the Impala web UI (port 25000).

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read permissions for the files in all applicable directories in all source tables, and read and execute permissions for the relevant data directories. (A SELECT operation could read files from multiple different HDFS directories if the source table is partitioned.) If a query attempts to read a data file and is unable to because of an HDFS permission error, the query halts and does not return any further results.

Related information:

The SELECT syntax is so extensive that it forms its own category of statements: queries. The other major classifications of SQL statements are data definition language (see DDL Statements on page 203) and data manipulation language (see DML Statements on page 204).

Because the focus of Impala is on fast queries with interactive response times over huge data sets, query performance and scalability are important considerations. See Tuning Impala for Performance on page 565 and Scalability Considerations for Impala on page 605 for details.

Joins in Impala SELECT Statements

A join query is a SELECT statement that combines data from two or more tables, and returns a result set containing items from some or all of those tables. It is a way to cross-reference and correlate related data that is organized into multiple tables, typically using identifiers that are repeated in each of the joined tables.

Syntax:

Impala supports a wide variety of `JOIN` clauses. Left, right, semi, full, and outer joins are supported in all Impala versions. The `CROSS JOIN` operator is available in Impala 1.2.2 and higher. During performance tuning, you can override the reordering of join clauses that Impala does internally by including the keyword `STRAIGHT_JOIN` immediately after the SELECT and any `DISTINCT` or `ALL` keywords.

```
SELECT select_list FROM
  table_or_subquery1 [INNER] JOIN table_or_subquery2 |
```

```
  table_or_subquery1 {LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]} JOIN table_or_subquery2
 |
  table_or_subquery1 {LEFT | RIGHT} SEMI JOIN table_or_subquery2 |
  table_or_subquery1 {LEFT | RIGHT} ANTI JOIN table_or_subquery2 |
    [ ON col1 = col2 [AND col3 = col4 ...] |
      USING (col1 [, col2 ...]) ]
  [other_join_clause ...]
[ WHERE where_clauses ]

SELECT select_list FROM
  table_or_subquery1, table_or_subquery2 [, table_or_subquery3 ...]
  [other_join_clause ...]
WHERE
    col1 = col2 [AND col3 = col4 ...]

SELECT select_list FROM
  table_or_subquery1 CROSS JOIN table_or_subquery2
  [other_join_clause ...]
[ WHERE where_clauses ]
```

SQL-92 and SQL-89 Joins:

Queries with the explicit `JOIN` keywords are known as SQL-92 style joins, referring to the level of the SQL standard where they were introduced. The corresponding `ON` or `USING` clauses clearly show which columns are used as the join keys in each case:

```
SELECT t1.c1, t2.c2 FROM t1 JOIN t2
  ON t1.id = t2.id and t1.type_flag = t2.type_flag
  WHERE t1.c1 > 100;

SELECT t1.c1, t2.c2 FROM t1 JOIN t2
  USING (id, type_flag)
  WHERE t1.c1 > 100;
```

The `ON` clause is a general way to compare columns across the two tables, even if the column names are different. The `USING` clause is a shorthand notation for specifying the join columns, when the column names are the same in both tables. You can code equivalent `WHERE` clauses that compare the columns, instead of `ON` or `USING` clauses, but that practice is not recommended because mixing the join comparisons with other filtering clauses is typically less readable and harder to maintain.

Queries with a comma-separated list of tables and subqueries are known as SQL-89 style joins. In these queries, the equality comparisons between columns of the joined tables go in the `WHERE` clause alongside other kinds of comparisons. This syntax is easy to learn, but it is also easy to accidentally remove a `WHERE` clause needed for the join to work correctly.

```
SELECT t1.c1, t2.c2 FROM t1, t2
  WHERE
  t1.id = t2.id AND t1.type_flag = t2.type_flag
  AND t1.c1 > 100;
```

Self-joins:

Impala can do self-joins, for example to join on two different columns in the same table to represent parent-child relationships or other tree-structured data. There is no explicit syntax for this; just use the same table name for both the left-hand and right-hand table, and assign different table aliases to use when referring to the fully qualified column names:

```
-- Combine fields from both parent and child rows.
SELECT lhs.id, rhs.parent, lhs.c1, rhs.c2 FROM tree_data lhs, tree_data rhs WHERE lhs.id
 = rhs.parent;
```

Inner and outer joins:

An inner join is the most common and familiar type: rows in the result set contain the requested columns from the appropriate tables, for all combinations of rows where the join columns of the tables have identical values. If a column

with the same name occurs in both tables, use a fully qualified name or a column alias to refer to the column in the select list or other clauses. Impala performs inner joins by default for both SQL-89 and SQL-92 join syntax:

```
-- The following 3 forms are all equivalent.
SELECT t1.id, c1, c2 FROM t1, t2 WHERE t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 JOIN t2 ON t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

An outer join retrieves all rows from the left-hand table, or the right-hand table, or both; wherever there is no matching data in the table on the other side of the join, the corresponding columns in the result set are set to NULL. To perform an outer join, include the OUTER keyword in the join operator, along with either LEFT, RIGHT, or FULL:

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.id = t2.id;
```

For outer joins, Impala requires SQL-92 syntax; that is, the JOIN keyword instead of comma-separated table names. Impala does not support vendor extensions such as (+) or *= notation for doing outer joins with SQL-89 query syntax.

Equijoins and Non-Equijoins:

By default, Impala requires an equality comparison between the left-hand and right-hand tables, either through ON, USING, or WHERE clauses. These types of queries are classified broadly as equijoins. Inner, outer, full, and semi joins can all be equijoins based on the presence of equality tests between columns in the left-hand and right-hand tables.

In Impala 1.2.2 and higher, non-equijoin queries are also possible, with comparisons such as != or < between the join columns. These kinds of queries require care to avoid producing huge result sets that could exceed resource limits. Once you have planned a non-equijoin query that produces a result set of acceptable size, you can code the query using the CROSS JOIN operator, and add the extra comparisons in the WHERE clause:

```
SELECT * FROM t1 CROSS JOIN t2 WHERE t1.total > t2.maximum_price;
```

In CDH 5.5 / Impala 2.3 and higher, additional non-equijoin queries are possible due to the addition of nested loop joins. These queries typically involve SEMI JOIN, ANTI JOIN, or FULL OUTER JOIN clauses. Impala sometimes also uses nested loop joins internally when evaluating OUTER JOIN queries involving complex type columns. Query phases involving nested loop joins do not use the spill-to-disk mechanism if they exceed the memory limit. Impala decides internally when to use each join mechanism; you cannot specify any query hint to choose between the nested loop join or the original hash join algorithm.

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.int_col < t2.int_col;
```

Semi-joins:

Semi-joins are a relatively rarely used variation. With the left semi-join, only data from the left-hand table is returned, for rows where there is matching data in the right-hand table, based on comparisons between join columns in ON or WHERE clauses. Only one instance of each row from the left-hand table is returned, regardless of how many matching rows exist in the right-hand table. A right semi-join (available in Impala 2.0 and higher) reverses the comparison and returns data from the right-hand table.

```
SELECT t1.c1, t1.c2, t1.c2 FROM t1 LEFT SEMI JOIN t2 ON t1.id = t2.id;
```

Natural joins (not supported):

Impala does not support the NATURAL JOIN operator, again to avoid inconsistent or huge result sets. Natural joins do away with the ON and USING clauses, and instead automatically join on all columns with the same names in the left-hand and right-hand tables. This kind of query is not recommended for rapidly evolving data structures such as are typically used in Hadoop. Thus, Impala does not support the NATURAL JOIN syntax, which can produce different query results as columns are added to or removed from tables.

If you do have any queries that use NATURAL JOIN, make sure to rewrite them with explicit USING clauses, because Impala could interpret the NATURAL keyword as a table alias:

```
-- 'NATURAL' is interpreted as an alias for 't1' and Impala attempts an inner join,
-- resulting in an error because inner joins require explicit comparisons between columns.
SELECT t1.c1, t2.c2 FROM t1 NATURAL JOIN t2;
ERROR: NotImplementedException: Join with 't2' requires at least one conjunctive equality
 predicate.
  To perform a Cartesian product between two tables, use a CROSS JOIN.

-- If you expect the tables to have identically named columns with matching values,
-- list the corresponding column names in a USING clause.
SELECT t1.c1, t2.c2 FROM t1 JOIN t2 USING (id, type_flag, name, address);
```

Anti-joins (CDH 5.2 / Impala 2.0 and higher only):

Impala supports the LEFT ANTI JOIN and RIGHT ANTI JOIN clauses in CDH 5.2 and higher. The LEFT or RIGHT keyword is required for this kind of join. For LEFT ANTI JOIN, this clause returns those values from the left-hand table that have no matching value in the right-hand table. RIGHT ANTI JOIN reverses the comparison and returns values from the right-hand table. You can express this negative relationship either through the ANTI JOIN clause or through a NOT EXISTS operator with a subquery.

Complex type considerations:

When referring to a column with a complex type (STRUCT, ARRAY, or MAP) in a query, you use join notation to "unpack" the scalar fields of the struct, the elements of the array, or the key-value pairs of the map. (The join notation is not required for aggregation operations, such as COUNT( ) or SUM( ) for array elements.) Because Impala recognizes which complex type elements are associated with which row of the result set, you use the same syntax as for a cross or cartesian join, without an explicit join condition. See Complex Types (CDH 5.5 or higher only) on page 139 for details about Impala support for complex types.

Usage notes:

You typically use join queries in situations like these:

- When related data arrives from different sources, with each data set physically residing in a separate table. For example, you might have address data from business records that you cross-check against phone listings or census data.

  Note: Impala can join tables of different file formats, including Impala-managed tables and HBase tables. For example, you might keep small dimension tables in HBase, for convenience of single-row lookups and updates, and for the larger fact tables use Parquet or other binary file format optimized for scan operations. Then, you can issue a join query to cross-reference the fact tables with the dimension tables.

- When data is normalized, a technique for reducing data duplication by dividing it across multiple tables. This kind of organization is often found in data that comes from traditional relational database systems. For example, instead of repeating some long string such as a customer name in multiple tables, each table might contain a numeric customer ID. Queries that need to display the customer name could "join" the table that specifies which customer ID corresponds to which name.
- When certain columns are rarely needed for queries, so they are moved into separate tables to reduce overhead for common queries. For example, a biography field might be rarely needed in queries on employee data. Putting that field in a separate table reduces the amount of I/O for common queries on employee addresses or phone numbers. Queries that do need the biography column can retrieve it by performing a join with that separate table.
- In CDH 5.5 / Impala 2.3 or higher, when referring to complex type columns in queries. See Complex Types (CDH 5.5 or higher only) on page 139 for details.

When comparing columns with the same names in ON or WHERE clauses, use the fully qualified names such as *db_name*.*table_name*, or assign table aliases, column aliases, or both to make the code more compact and understandable:

```
select t1.c1 as first_id, t2.c2 as second_id from
  t1 join t2 on first_id = second_id;

select fact.custno, dimension.custno from
  customer_data as fact join customer_address as dimension
  using (custno)
```



Note:

Performance for join queries is a crucial aspect for Impala, because complex join queries are resource-intensive operations. An efficient join query produces much less network traffic and CPU overhead than an inefficient one. For best results:

- Make sure that both table and column statistics are available for all the tables involved in a join query, and especially for the columns referenced in any join conditions. Impala uses the statistics to automatically deduce an efficient join order. Use SHOW TABLE STATS *table_name* and SHOW COLUMN STATS *table_name* to check if statistics are already present. Issue the COMPUTE STATS *table_name* for a nonpartitioned table, or (in Impala 2.1.0 and higher) COMPUTE INCREMENTAL STATS *table_name* for a partitioned table, to collect the initial statistics at both the table and column levels, and to keep the statistics up to date after any substantial INSERT or LOAD DATA operations.
- If table or column statistics are not available, join the largest table first. You can check the existence of statistics with the SHOW TABLE STATS *table_name* and SHOW COLUMN STATS *table_name* statements.
- If table or column statistics are not available, join subsequent tables according to which table has the most selective filter, based on overall size and WHERE clauses. Joining the table with the most selective filter results in the fewest number of rows being returned.

For more information and examples of performance for join queries, see Performance Considerations for Join Queries on page 568.

To control the result set from a join query, include the names of corresponding column names in both tables in an ON or USING clause, or by coding equality comparisons for those columns in the WHERE clause.

```
[localhost:21000] > select c_last_name, ca_city from customer join customer_address
where c_customer_sk = ca_address_sk;
+-------------+-----------------+
| c_last_name | ca_city         |
+-------------+-----------------+
| Lewis       | Fairfield       |
| Moses       | Fairview        |
| Hamilton    | Pleasant Valley |
| White       | Oak Ridge       |
| Moran       | Glendale        |
...
| Richards    | Lakewood         |
| Day         | Lebanon          |
| Painter     | Oak Hill         |
| Bentley     | Greenfield       |
| Jones       | Stringtown       |
+-------------+-----------------+
Returned 50000 row(s) in 9.82s
```

One potential downside of joins is the possibility of excess resource usage in poorly constructed queries. Impala imposes restrictions on join queries to guard against such issues. To minimize the chance of runaway queries on large data sets, Impala requires every join query to contain at least one equality predicate between the columns of the various tables. For example, if T1 contains 1000 rows and T2 contains 1,000,000 rows, a query SELECT *columns* FROM t1 JOIN

t2 could return up to 1 billion rows (1000 * 1,000,000); Impala requires that the query include a clause such as ON t1.c1 = t2.c2 or WHERE t1.c1 = t2.c2.

Because even with equality clauses, the result set can still be large, as we saw in the previous example, you might use a LIMIT clause to return a subset of the results:

```
[localhost:21000] > select c_last_name, ca_city from customer, customer_address where
c_customer_sk = ca_address_sk limit 10;
+-------------+-----------------+
| c_last_name | ca_city         |
+-------------+-----------------+
| Lewis       | Fairfield       |
| Moses       | Fairview        |
| Hamilton    | Pleasant Valley |
| White       | Oak Ridge       |
| Moran       | Glendale        |
| Sharp       | Lakeview        |
| Wiles       | Farmington      |
| Shipman     | Union           |
| Gilbert     | New Hope        |
| Brunson     | Martinsville    |
+-------------+-----------------+
Returned 10 row(s) in 0.63s
```

Or you might use additional comparison operators or aggregation functions to condense a large result set into a smaller set of values:

```
[localhost:21000] > -- Find the names of customers who live in one particular town.
[localhost:21000] > select distinct c_last_name from customer, customer_address where
  c_customer_sk = ca_address_sk
  and ca_city = "Green Acres";
+---------------+
| c_last_name   |
+---------------+
| Hensley       |
| Pearson       |
| Mayer         |
| Montgomery    |
| Ricks         |
...
| Barrett       |
| Price         |
| Hill          |
| Hansen        |
| Meeks         |
+---------------+
Returned 332 row(s) in 0.97s

[localhost:21000] > -- See how many different customers in this town have names starting
 with "A".
[localhost:21000] > select count(distinct c_last_name) from customer, customer_address
 where
  c_customer_sk = ca_address_sk
  and ca_city = "Green Acres"
  and substr(c_last_name,1,1) = "A";
+----------------------------+
| count(distinct c_last_name) |
+----------------------------+
| 12                         |
+----------------------------+
Returned 1 row(s) in 1.00s
```

Because a join query can involve reading large amounts of data from disk, sending large amounts of data across the network, and loading large amounts of data into memory to do the comparisons and filtering, you might do benchmarking, performance analysis, and query tuning to find the most efficient join queries for your data set, hardware capacity, network configuration, and cluster workload.

The two categories of joins in Impala are known as **partitioned joins** and **broadcast joins**. If inaccurate table or column statistics, or some quirk of the data distribution, causes Impala to choose the wrong mechanism for a particular join, consider using query hints as a temporary workaround. For details, see Optimizer Hints in Impala on page 387.

Handling NULLs in Join Columns:

By default, join key columns do not match if either one contains a NULL value. To treat such columns as equal if both contain NULL, you can use an expression such as A = B OR (A IS NULL AND B IS NULL). In CDH 5.7 / Impala 2.5 and higher, the <=> operator (shorthand for IS NOT DISTINCT FROM) performs the same comparison in a concise and efficient form. The <=> operator is more efficient in for comparing join keys in a NULL-safe manner, because the operator can use a hash join while the OR expression cannot.

Examples:

The following examples refer to these simple tables containing small sets of integers:

```
[localhost:21000] > create table t1 (x int);
[localhost:21000] > insert into t1 values (1), (2), (3), (4), (5), (6);

[localhost:21000] > create table t2 (y int);
[localhost:21000] > insert into t2 values (2), (4), (6);

[localhost:21000] > create table t3 (z int);
[localhost:21000] > insert into t3 values (1), (3), (5);
```

The following example demonstrates an anti-join, returning the values from T1 that do not exist in T2 (in this case, the odd numbers 1, 3, and 5):

```
[localhost:21000] > select x from t1 left anti join t2 on (t1.x = t2.y);
+---+
| x |
+---+
| 1 |
| 3 |
| 5 |
+---+
```

Related information:

See these tutorials for examples of different kinds of joins:

- Cross Joins and Cartesian Products with the CROSS JOIN Operator on page 57

ORDER BY Clause

The ORDER BY clause of a SELECT statement sorts the result set based on the values from one or more columns.

First, data is sorted locally by each impalad daemon, then streamed to the coordinator daemon, which merges the sorted result sets. For distributed queries, this is a relatively expensive operation and can require more memory capacity than a query without ORDER BY. Even if the query takes approximately the same time to finish with or without the ORDER BY clause, subjectively it can appear slower because no results are available until all processing is finished, rather than results coming back gradually as rows matching the WHERE clause are found. Therefore, if you only need the first N results from the sorted result set, also include the LIMIT clause, which reduces network overhead and the memory requirement on the coordinator node.

Syntax:

The full syntax for the ORDER BY clause is:

```
ORDER BY col_ref [, col_ref ...] [ASC | DESC] [NULLS FIRST | NULLS LAST]

col_ref ::= column_name | integer_literal
```

Although the most common usage is ORDER BY *column_name*, you can also specify ORDER BY 1 to sort by the first column of the result set, ORDER BY 2 to sort by the second column, and so on. The number must be a numeric literal, not some other kind of constant expression. (If the argument is some other expression, even a STRING value, the query succeeds but the order of results is undefined.)

ORDER BY *column_number* can only be used when the query explicitly lists the columns in the SELECT list, not with SELECT * queries.

Ascending and descending sorts:

The default sort order (the same as using the ASC keyword) puts the smallest values at the start of the result set, and the largest values at the end. Specifying the DESC keyword reverses that order.

Sort order for NULL values:

See NULL on page 170 for details about how NULL values are positioned in the sorted result set, and how to use the NULLS FIRST and NULLS LAST clauses. (The sort position for NULL values in ORDER BY ... DESC queries is changed in Impala 1.2.1 and higher to be more standards-compliant, and the NULLS FIRST and NULLS LAST keywords are new in Impala 1.2.1.)

Prior to Impala 1.4.0, Impala required any query including an ORDER BY clause to also use a LIMIT clause. In Impala 1.4.0 and higher, the LIMIT clause is optional for ORDER BY queries. In cases where sorting a huge result set requires enough memory to exceed the Impala memory limit for a particular executor Impala daemon, Impala automatically uses a temporary disk work area to perform the sort operation.

Complex type considerations:

In CDH 5.5 / Impala 2.3 and higher, the complex data types STRUCT, ARRAY, and MAP are available. These columns cannot be referenced directly in the ORDER BY clause. When you query a complex type column, you use join notation to "unpack" the elements of the complex type, and within the join query you can include an ORDER BY clause to control the order in the result set of the scalar elements from the complex type. See Complex Types (CDH 5.5 or higher only) on page 139 for details about Impala support for complex types.

The following query shows how a complex type column cannot be directly used in an ORDER BY clause:

```
CREATE TABLE games (id BIGINT, score ARRAY <BIGINT>) STORED AS PARQUET;
...use LOAD DATA to load externally created Parquet files into the table...
SELECT id FROM games ORDER BY score DESC;
ERROR: AnalysisException: ORDER BY expression 'score' with complex type 'ARRAY<BIGINT>'
 is not supported.
```

Examples:

The following query retrieves the user ID and score, only for scores greater than one million, with the highest scores for each user listed first. Because the individual array elements are now represented as separate rows in the result set, they can be used in the ORDER BY clause, referenced using the ITEM pseudo-column that represents each array element.

```
SELECT id, item FROM games, games.score
  WHERE item > 1000000
ORDER BY id, item desc;
```

The following queries use similar ORDER BY techniques with variations of the GAMES table, where the complex type is an ARRAY containing STRUCT or MAP elements to represent additional details about each game that was played. For an array of structures, the fields of the structure are referenced as ITEM.*field_name*. For an array of maps, the keys and values within each array element are referenced as ITEM.KEY and ITEM.VALUE.

```
CREATE TABLE games2 (id BIGINT, play array < struct <game_name: string, score: BIGINT,
 high_score: boolean> >) STORED AS PARQUET
...use LOAD DATA to load externally created Parquet files into the table...
SELECT id, item.game_name, item.score FROM games2, games2.play
  WHERE item.score > 1000000
ORDER BY id, item.score DESC;

CREATE TABLE games3 (id BIGINT, play ARRAY < MAP <STRING, BIGINT> >) STORED AS PARQUET;
...use LOAD DATA to load externally created Parquet files into the table...
SELECT id, info.key AS k, info.value AS v from games3, games3.play AS plays,
games3.play.item AS info
  WHERE info.KEY = 'score' AND info.VALUE > 1000000
ORDER BY id, info.value desc;
```

Usage notes:

Although the LIMIT clause is now optional on ORDER BY queries, if your query only needs some number of rows that you can predict in advance, use the LIMIT clause to reduce unnecessary processing. For example, if the query has a clause LIMIT 10, each executor Impala daemon sorts its portion of the relevant result set and only returns 10 rows to the coordinator node. The coordinator node picks the 10 highest or lowest row values out of this small intermediate result set.

If an ORDER BY clause is applied to an early phase of query processing, such as a subquery or a view definition, Impala ignores the ORDER BY clause. To get ordered results from a subquery or view, apply an ORDER BY clause to the outermost or final SELECT level.

ORDER BY is often used in combination with LIMIT to perform "top-N" queries:

```
SELECT user_id AS "Top 10 Visitors", SUM(page_views) FROM web_stats
  GROUP BY page_views, user_id
  ORDER BY SUM(page_views) DESC LIMIT 10;
```

ORDER BY is sometimes used in combination with OFFSET and LIMIT to paginate query results, although it is relatively inefficient to issue multiple queries like this against the large tables typically used with Impala:

```
SELECT page_title AS "Page 1 of search results", page_url FROM search_content
  WHERE LOWER(page_title) LIKE '%game%')
  ORDER BY page_title LIMIT 10 OFFSET 0;
SELECT page_title AS "Page 2 of search results", page_url FROM search_content
  WHERE LOWER(page_title) LIKE '%game%')
  ORDER BY page_title LIMIT 10 OFFSET 10;
SELECT page_title AS "Page 3 of search results", page_url FROM search_content
  WHERE LOWER(page_title) LIKE '%game%')
  ORDER BY page_title LIMIT 10 OFFSET 20;
```

Internal details:

Impala sorts the intermediate results of an ORDER BY clause in memory whenever practical. In a cluster of N executor Impala daemons, each daemon sorts roughly 1/Nth of the result set, the exact proportion varying depending on how the data matching the query is distributed in HDFS.

If the size of the sorted intermediate result set on any executor Impala daemon would cause the query to exceed the Impala memory limit, Impala sorts as much as practical in memory, then writes partially sorted data to disk. (This technique is known in industry terminology as "external sorting" and "spilling to disk".) As each 8 MB batch of data is written to disk, Impala frees the corresponding memory to sort a new 8 MB batch of data. When all the data has been processed, a final merge sort operation is performed to correctly order the in-memory and on-disk results as the result set is transmitted back to the coordinator node. When external sorting becomes necessary, Impala requires approximately 60 MB of RAM at a minimum for the buffers needed to read, write, and sort the intermediate results. If more RAM is available on the Impala daemon, Impala will use the additional RAM to minimize the amount of disk I/O for sorting.

This external sort technique is used as appropriate on each Impala daemon (possibly including the coordinator node) to sort the portion of the result set that is processed on that node. When the sorted intermediate results are sent back to the coordinator node to produce the final result set, the coordinator node uses a merge sort technique to produce a final sorted result set without using any extra resources on the coordinator node.

Configuration for disk usage:

By default, intermediate files used during large sort, join, aggregation, or analytic function operations are stored in the directory /tmp/impala-scratch . These files are removed when the operation finishes. (Multiple concurrent queries can perform operations that use the "spill to disk" technique, without any name conflicts for these temporary files.) You can specify a different location by starting the impalad daemon with the --scratch_dirs="*path_to_directory*" configuration option or the equivalent configuration option in the Cloudera Manager user interface. You can specify a single directory, or a comma-separated list of directories. The scratch directories must be on the local filesystem, not in HDFS. You might specify different directory paths for different hosts, depending on the capacity and speed of the available storage devices. In CDH 5.5 / Impala 2.3 or higher, Impala successfully starts (with a warning written to the log) if it cannot create or read and write files in one of the scratch directories. If there is less than 1 GB free on the filesystem where that directory resides, Impala still runs, but writes a

warning message to its log. If Impala encounters an error reading or writing files in a scratch directory during a query, Impala logs the error and the query fails.

Sorting considerations: Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored and the results are not necessarily sorted. An INSERT ... SELECT operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

An ORDER BY clause without an additional LIMIT clause is ignored in any view definition. If you need to sort the entire result set from a view, use an ORDER BY clause in the SELECT statement that queries the view. You can still make a simple "top 10" report by combining the ORDER BY and LIMIT clauses in the same view definition:

```
[localhost:21000] > create table unsorted (x bigint);
[localhost:21000] > insert into unsorted values (1), (9), (3), (7), (5), (8), (4), (6),
 (2);
[localhost:21000] > create view sorted_view as select x from unsorted order by x;
[localhost:21000] > select x from sorted_view; -- ORDER BY clause in view has no effect.
+---+
| x |
+---+
| 1 |
| 9 |
| 3 |
| 7 |
| 5 |
| 8 |
| 4 |
| 6 |
| 2 |
+---+
[localhost:21000] > select x from sorted_view order by x; -- View query requires ORDER
 BY at outermost level.
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
+---+
[localhost:21000] > create view top_3_view as select x from unsorted order by x limit
3;
[localhost:21000] > select x from top_3_view; -- ORDER BY and LIMIT together in view
definition are preserved.
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

With the lifting of the requirement to include a LIMIT clause in every ORDER BY query (in Impala 1.4 and higher):

- Now the use of scratch disk space raises the possibility of an "out of disk space" error on a particular Impala daemon, as opposed to the previous possibility of an "out of memory" error. Make sure to keep at least 1 GB free on the filesystem used for temporary sorting work.

In Impala 1.2.1 and higher, all NULL values come at the end of the result set for ORDER BY ... ASC queries, and at the beginning of the result set for ORDER BY ... DESC queries. In effect, NULL is considered greater than all other values for sorting purposes. The original Impala behavior always put NULL values at the end, even for ORDER BY ... DESC queries. The new behavior in Impala 1.2.1 makes Impala more compatible with other popular database systems.

In Impala 1.2.1 and higher, you can override or specify the sorting behavior for NULL by adding the clause NULLS FIRST or NULLS LAST at the end of the ORDER BY clause.

```
[localhost:21000] > create table numbers (x int);
[localhost:21000] > insert into numbers values (1), (null), (2), (null), (3);
[localhost:21000] > select x from numbers order by x nulls first;
+------+
| x    |
+------+
| NULL |
| NULL |
| 1    |
| 2    |
| 3    |
+------+
[localhost:21000] > select x from numbers order by x desc nulls first;
+------+
| x    |
+------+
| NULL |
| NULL |
| 3    |
| 2    |
| 1    |
+------+
[localhost:21000] > select x from numbers order by x nulls last;
+------+
| x    |
+------+
| 1    |
| 2    |
| 3    |
| NULL |
| NULL |
+------+
[localhost:21000] > select x from numbers order by x desc nulls last;
+------+
| x    |
+------+
| 3    |
| 2    |
| 1    |
| NULL |
| NULL |
+------+
```

Related information:

See SELECT Statement on page 295 for further examples of queries with the ORDER BY clause.

Analytic functions use the ORDER BY clause in a different context to define the sequence in which rows are analyzed. See Impala Analytic Functions on page 506 for details.

GROUP BY Clause

Specify the GROUP BY clause in queries that use aggregation functions, such as COUNT(), SUM(), AVG(), MIN(), and MAX(). Specify in the GROUP BY clause the names of all the columns that do not participate in the aggregation operation.

Complex type considerations:

In CDH 5.5 / Impala 2.3 and higher, the complex data types STRUCT, ARRAY, and MAP are available. These columns cannot be referenced directly in the ORDER BY clause. When you query a complex type column, you use join notation to "unpack" the elements of the complex type, and within the join query you can include an ORDER BY clause to control the order in the result set of the scalar elements from the complex type. See Complex Types (CDH 5.5 or higher only) on page 139 for details about Impala support for complex types.

Zero-length strings: For purposes of clauses such as DISTINCT and GROUP BY, Impala considers zero-length strings (""), NULL, and space to all be different values.

Examples:

For example, the following query finds the 5 items that sold the highest total quantity (using the SUM( ) function, and also counts the number of sales transactions for those items (using the COUNT( ) function). Because the column representing the item IDs is not used in any aggregation functions, we specify that column in the GROUP BY clause.

```
select
  ss_item_sk as Item,
  count(ss_item_sk) as Times_Purchased,
  sum(ss_quantity) as Total_Quantity_Purchased
from store_sales
  group by ss_item_sk
  order by sum(ss_quantity) desc
  limit 5;
+-------+-----------------+--------------------------+
| item  | times_purchased | total_quantity_purchased |
+-------+-----------------+--------------------------+
| 9325  | 372             | 19072                    |
| 4279  | 357             | 18501                    |
| 7507  | 371             | 18475                    |
| 5953  | 369             | 18451                    |
| 16753 | 375             | 18446                    |
+-------+-----------------+--------------------------+
```

The HAVING clause lets you filter the results of aggregate functions, because you cannot refer to those expressions in the WHERE clause. For example, to find the 5 lowest-selling items that were included in at least 100 sales transactions, we could use this query:

```
select
  ss_item_sk as Item,
  count(ss_item_sk) as Times_Purchased,
  sum(ss_quantity) as Total_Quantity_Purchased
from store_sales
  group by ss_item_sk
  having count(ss_item_sk)>= 100
  order by sum(ss_quantity)
  limit 5;
+-------+-----------------+--------------------------+
| item  | times_purchased | total_quantity_purchased |
+-------+-----------------+--------------------------+
| 13943 | 105             | 4087                     |
| 2992  | 101             | 4176                     |
| 4773  | 107             | 4204                     |
| 14350 | 103             | 4260                     |
| 11956 | 102             | 4275                     |
+-------+-----------------+--------------------------+
```

When performing calculations involving scientific or financial data, remember that columns with type FLOAT or DOUBLE are stored as true floating-point numbers, which cannot precisely represent every possible fractional value. Thus, if you include a FLOAT or DOUBLE column in a GROUP BY clause, the results might not precisely match literal values in your query or from an original Text data file. Use rounding operations, the BETWEEN operator, or another arithmetic technique to match floating-point values that are "near" literal values you expect. For example, this query on the ss_wholesale_cost column returns cost values that are close but not identical to the original figures that were entered as decimal fractions.

```
select ss_wholesale_cost, avg(ss_quantity * ss_sales_price) as avg_revenue_per_sale
  from sales
  group by ss_wholesale_cost
  order by avg(ss_quantity * ss_sales_price) desc
  limit 5;
+-------------------+----------------------+
| ss_wholesale_cost | avg_revenue_per_sale |
+-------------------+----------------------+
| 96.94000244140625 | 4454.351539300434    |
| 95.93000030517578 | 4423.119941283189    |
| 98.37999725341797 | 4332.516490316291    |
| 97.97000122070312 | 4330.480601655014    |
| 98.52999877929688 | 4291.316953108634    |
+-------------------+----------------------+
```

Notice how wholesale cost values originally entered as decimal fractions such as 96.94 and 98.38 are slightly larger or smaller in the result set, due to precision limitations in the hardware floating-point types. The imprecise representation of FLOAT and DOUBLE values is why financial data processing systems often store currency using data types that are less space-efficient but avoid these types of rounding errors.

Related information:

SELECT Statement on page 295, Impala Aggregate Functions on page 479

HAVING Clause

Performs a filter operation on a SELECT query, by examining the results of aggregation functions rather than testing each individual table row. Therefore, it is always used in conjunction with a function such as COUNT(), SUM(), AVG(), MIN(), or MAX(), and typically with the GROUP BY clause also.

Restrictions:

The filter expression in the HAVING clause cannot include a scalar subquery.

Related information:

SELECT Statement on page 295, GROUP BY Clause on page 306, Impala Aggregate Functions on page 479

LIMIT Clause

The LIMIT clause in a SELECT query sets a maximum number of rows for the result set. Pre-selecting the maximum size of the result set helps Impala to optimize memory usage while processing a distributed query.

Syntax:

```
LIMIT constant_integer_expression
```

The argument to the LIMIT clause must evaluate to a constant value. It can be a numeric literal, or another kind of numeric expression involving operators, casts, and function return values. You cannot refer to a column or use a subquery.

Usage notes:

This clause is useful in contexts such as:

- To return exactly N items from a top-N query, such as the 10 highest-rated items in a shopping category or the 50 hostnames that refer the most traffic to a web site.
- To demonstrate some sample values from a table or a particular query. (To display some arbitrary items, use a query with no ORDER BY clause. An ORDER BY clause causes additional memory and/or disk usage during the query.)
- To keep queries from returning huge result sets by accident if a table is larger than expected, or a WHERE clause matches more rows than expected.

Originally, the value for the LIMIT clause had to be a numeric literal. In Impala 1.2.1 and higher, it can be a numeric expression.

Prior to Impala 1.4.0, Impala required any query including an ORDER BY clause to also use a LIMIT clause. In Impala 1.4.0 and higher, the LIMIT clause is optional for ORDER BY queries. In cases where sorting a huge result set requires enough memory to exceed the Impala memory limit for a particular executor Impala daemon, Impala automatically uses a temporary disk work area to perform the sort operation.

See ORDER BY Clause on page 302 for details.

In Impala 1.2.1 and higher, you can combine a LIMIT clause with an OFFSET clause to produce a small result set that is different from a top-N query, for example, to return items 11 through 20. This technique can be used to simulate "paged" results. Because Impala queries typically involve substantial amounts of I/O, use this technique only for compatibility in cases where you cannot rewrite the application logic. For best performance and scalability, wherever practical, query as many items as you expect to need, cache them on the application side, and display small groups of results to users using application logic.

Restrictions:

Correlated subqueries used in EXISTS and IN operators cannot include a LIMIT clause.

Examples:

The following example shows how the LIMIT clause caps the size of the result set, with the limit being applied after any other clauses such as WHERE.

```
[localhost:21000] > create database limits;
[localhost:21000] > use limits;
[localhost:21000] > create table numbers (x int);
[localhost:21000] > insert into numbers values (1), (3), (4), (5), (2);
Inserted 5 rows in 1.34s
[localhost:21000] > select x from numbers limit 100;
+---+
| x |
+---+
| 1 |
| 3 |
| 4 |
| 5 |
| 2 |
+---+
Returned 5 row(s) in 0.26s
[localhost:21000] > select x from numbers limit 3;
+---+
| x |
+---+
| 1 |
| 3 |
| 4 |
+---+
Returned 3 row(s) in 0.27s
[localhost:21000] > select x from numbers where x > 2 limit 2;
+---+
| x |
+---+
| 3 |
| 4 |
+---+
Returned 2 row(s) in 0.27s
```

For top-N and bottom-N queries, you use the ORDER BY and LIMIT clauses together:

```
[localhost:21000] > select x as "Top 3" from numbers order by x desc limit 3;
+-------+
| top 3 |
+-------+
| 5     |
| 4     |
| 3     |
+-------+
[localhost:21000] > select x as "Bottom 3" from numbers order by x limit 3;
+----------+
| bottom 3 |
+----------+
| 1        |
| 2        |
| 3        |
+----------+
```

You can use constant values besides integer literals as the LIMIT argument:

```
-- Other expressions that yield constant integer values work too.
SELECT x FROM t1 LIMIT 1e6;                        -- Limit is one million.
SELECT x FROM t1 LIMIT length('hello world');      -- Limit is 11.
SELECT x FROM t1 LIMIT 2+2;                        -- Limit is 4.
SELECT x FROM t1 LIMIT cast(truncate(9.9) AS INT); -- Limit is 9.
```

OFFSET Clause

The OFFSET clause in a SELECT query causes the result set to start some number of rows after the logical first item. The result set is numbered starting from zero, so OFFSET 0 produces the same result as leaving out the OFFSET clause. Always use this clause in combination with ORDER BY (so that it is clear which item should be first, second, and so on) and LIMIT (so that the result set covers a bounded range, such as items 0-9, 100-199, and so on).

In Impala 1.2.1 and higher, you can combine a LIMIT clause with an OFFSET clause to produce a small result set that is different from a top-N query, for example, to return items 11 through 20. This technique can be used to simulate "paged" results. Because Impala queries typically involve substantial amounts of I/O, use this technique only for compatibility in cases where you cannot rewrite the application logic. For best performance and scalability, wherever practical, query as many items as you expect to need, cache them on the application side, and display small groups of results to users using application logic.

Examples:

The following example shows how you could run a "paging" query originally written for a traditional database application. Because typical Impala queries process megabytes or gigabytes of data and read large data files from disk each time, it is inefficient to run a separate query to retrieve each small group of items. Use this technique only for compatibility while porting older applications, then rewrite the application code to use a single query with a large result set, and display pages of results from the cached result set.

```
[localhost:21000] > create table numbers (x int);
[localhost:21000] > insert into numbers select x from very_long_sequence;
Inserted 1000000 rows in 1.34s
[localhost:21000] > select x from numbers order by x limit 5 offset 0;
+----+
| x  |
+----+
| 1  |
| 2  |
| 3  |
| 4  |
| 5  |
+----+
[localhost:21000] > select x from numbers order by x limit 5 offset 5;
+----+
| x  |
+----+
| 6  |
| 7  |
| 8  |
| 9  |
| 10 |
+----+
```

UNION Clause

The UNION clause lets you combine the result sets of multiple queries. By default, the result sets are combined as if the DISTINCT operator was applied.

Syntax:

```
query_1 UNION [DISTINCT | ALL] query_2
```

Usage notes:

The UNION keyword by itself is the same as UNION DISTINCT. Because eliminating duplicates can be a memory-intensive process for a large result set, prefer UNION ALL where practical. (That is, when you know the different queries in the union will not produce any duplicates, or where the duplicate values are acceptable.)

When an ORDER BY clause applies to a UNION ALL or UNION query, in Impala 1.4 and higher, the LIMIT clause is no longer required. To make the ORDER BY and LIMIT clauses apply to the entire result set, turn the UNION query into a subquery, SELECT from the subquery, and put the ORDER BY clause at the end, outside the subquery.

Examples:

First, set up some sample data, including duplicate 1 values:

```
[localhost:21000] > create table few_ints (x int);
[localhost:21000] > insert into few_ints values (1), (1), (2), (3);
[localhost:21000] > set default_order_by_limit=1000;
```

This example shows how `UNION ALL` returns all rows from both queries, without any additional filtering to eliminate duplicates. For the large result sets common with Impala queries, this is the most memory-efficient technique.

```
[localhost:21000] > select x from few_ints order by x;
+---+
| x |
+---+
| 1 |
| 1 |
| 2 |
| 3 |
+---+
Returned 4 row(s) in 0.41s
[localhost:21000] > select x from few_ints union all select x from few_ints;
+---+
| x |
+---+
| 1 |
| 1 |
| 2 |
| 3 |
| 1 |
| 1 |
| 2 |
| 3 |
+---+
Returned 8 row(s) in 0.42s
[localhost:21000] > select * from (select x from few_ints union all select x from
few_ints) as t1 order by x;
+---+
| x |
+---+
| 1 |
| 1 |
| 1 |
| 1 |
| 2 |
| 2 |
| 3 |
| 3 |
+---+
Returned 8 row(s) in 0.53s
[localhost:21000] > select x from few_ints union all select 10;
+----+
| x  |
+----+
| 10 |
| 1  |
| 1  |
| 2  |
| 3  |
+----+
Returned 5 row(s) in 0.38s
```

This example shows how the `UNION` clause without the `ALL` keyword condenses the result set to eliminate all duplicate values, making the query take more time and potentially more memory. The extra processing typically makes this technique not recommended for queries that return result sets with millions or billions of values.

```
[localhost:21000] > select x from few_ints union select x+1 from few_ints;
+---+
| x |
+---+
| 3 |
| 4 |
```

```
| 1 |
| 2 |
+---+
Returned 4 row(s) in 0.51s
[localhost:21000] > select x from few_ints union select 10;
+----+
| x  |
+----+
| 2  |
| 10 |
| 1  |
| 3  |
+----+
Returned 4 row(s) in 0.49s
[localhost:21000] > select * from (select x from few_ints union select x from few_ints)
 as t1 order by x;
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.53s
```

Subqueries in Impala SELECT Statements

A ***subquery*** is a query that is nested within another query. Subqueries let queries on one table dynamically adapt based on the contents of another table. This technique provides great flexibility and expressive power for SQL queries.

A subquery can return a result set for use in the `FROM` or `WITH` clauses, or with operators such as `IN` or `EXISTS`.

A ***scalar subquery*** produces a result set with a single row containing a single column, typically produced by an aggregation function such as `MAX()` or `SUM()`. This single result value can be substituted in scalar contexts such as arguments to comparison operators. If the result set is empty, the value of the scalar subquery is `NULL`. For example, the following query finds the maximum value of `T2.Y` and then substitutes that value into the `WHERE` clause of the outer block that queries `T1`:

```
SELECT x FROM t1 WHERE x > (SELECT MAX(y) FROM t2);
```

Uncorrelated subqueries do not refer to any tables from the outer block of the query. The same value or set of values produced by the subquery is used when evaluating each row from the outer query block. In this example, the subquery returns an arbitrary number of values from `T2.Y`, and each value of `T1.X` is tested for membership in that same set of values:

```
SELECT x FROM t1 WHERE x IN (SELECT y FROM t2);
```

Correlated subqueries compare one or more values from the outer query block to values referenced in the `WHERE` clause of the subquery. Each row evaluated by the outer `WHERE` clause can be evaluated using a different set of values. These kinds of subqueries are restricted in the kinds of comparisons they can do between columns of the inner and outer tables. (See the following **Restrictions** item.)

For example, the following query finds all the employees with salaries that are higher than average for their department. The subquery potentially computes a different `AVG()` value for each employee.

```
SELECT employee_name, employee_id FROM employees one WHERE
  salary > (SELECT avg(salary) FROM employees two WHERE one.dept_id = two.dept_id);
```

Syntax:

Subquery in the `FROM` clause:

```
SELECT select_list FROM table_ref [, table_ref ...]

table_ref ::= table_name | (select_statement)
```

Subqueries in WHERE clause:

```
WHERE value comparison_operator (scalar_select_statement)
WHERE value [NOT] IN (select_statement)
WHERE [NOT] EXISTS (correlated_select_statement)
WHERE NOT EXISTS (correlated_select_statement)
```

comparison_operator is a numeric comparison such as =, <=, !=, and so on, or a string comparison operator such as LIKE or REGEXP.

Although you can use non-equality comparison operators such as < or >=, the subquery must include at least one equality comparison between the columns of the inner and outer query blocks.

All syntax is available for both correlated and uncorrelated queries, except that the NOT EXISTS clause cannot be used with an uncorrelated subquery.

Impala subqueries can be nested arbitrarily deep.

Standards compliance: Introduced in SQL:1999.

Examples:

This example illustrates how subqueries can be used in the FROM clause to organize the table names, column names, and column values by producing intermediate result sets, especially for join queries.

```
SELECT avg(t1.x), max(t2.y) FROM
  (SELECT id, cast(a AS DECIMAL(10,5)) AS x FROM raw_data WHERE a BETWEEN 0 AND 100) AS
 t1
  JOIN
  (SELECT id, length(s) AS y FROM raw_data WHERE s LIKE 'A%') AS t2;
  USING (id);
```

These examples show how a query can test for the existence of values in a separate table using the EXISTS() operator with a subquery.

The following examples show how a value can be compared against a set of values returned by a subquery.

```
SELECT count(x) FROM t1 WHERE EXISTS(SELECT 1 FROM t2 WHERE t1.x = t2.y * 10);

SELECT x FROM t1 WHERE x IN (SELECT y FROM t2 WHERE state = 'CA');
```

The following examples demonstrate scalar subqueries. When a subquery is known to return a single value, you can substitute it where you would normally put a constant value.

```
SELECT x FROM t1 WHERE y = (SELECT max(z) FROM t2);
SELECT x FROM t1 WHERE y > (SELECT count(z) FROM t2);
```

Usage notes:

If the same table is referenced in both the outer and inner query blocks, construct a table alias in the outer query block and use a fully qualified name to distinguish the inner and outer table references:

```
SELECT * FROM t1 one WHERE id IN (SELECT parent FROM t1 two WHERE t1.parent = t2.id);
```

The STRAIGHT_JOIN hint affects the join order of table references in the query block containing the hint. It does not affect the join order of nested queries, such as views, inline views, or WHERE-clause subqueries. To use this hint for performance tuning of complex queries, apply the hint to all query blocks that need a fixed join order.

Internal details:

Internally, subqueries involving IN, NOT IN, EXISTS, or NOT EXISTS clauses are rewritten into join queries. Depending on the syntax, the subquery might be rewritten to an outer join, semi join, cross join, or anti join.

A query is processed differently depending on whether the subquery calls any aggregation functions. There are correlated and uncorrelated forms, with and without calls to aggregation functions. Each of these four categories is rewritten differently.

Column statistics considerations:

Because queries that include correlated and uncorrelated subqueries in the WHERE clause are written into join queries, to achieve best performance, follow the same guidelines for running the COMPUTE STATS statement as you do for tables involved in regular join queries. Run the COMPUTE STATS statement for each associated tables after loading or substantially changing the data in that table. See Table and Column Statistics on page 575 for details.

Added in: Subqueries are substantially enhanced starting in Impala 2.0 for CDH 4, and CDH 5.2.0. Now, they can be used in the WHERE clause, in combination with clauses such as EXISTS and IN, rather than just in the FROM clause.

Restrictions:

The initial Impala support for nested subqueries addresses the most common use cases. Some restrictions remain:

- Although you can use subqueries in a query involving UNION or UNION ALL in Impala 2.1.0 and higher, currently you cannot construct a union of two subqueries (for example, in the argument of an IN or EXISTS operator).
- Subqueries returning scalar values cannot be used with the operators ANY or ALL. (Impala does not currently have a SOME operator, but if it did, the same restriction would apply.)
- For the EXISTS and NOT EXISTS clauses, any subquery comparing values from the outer query block to another table must use at least one equality comparison, not exclusively other kinds of comparisons such as less than, greater than, BETWEEN, or !=.
- Currently, a scalar subquery cannot be used as the first or second argument to the BETWEEN operator.
- A subquery cannot be used inside an OR conjunction. Expressions inside a subquery, for example in the WHERE clause, can use OR conjunctions; the restriction only applies to parts of the query “above” the subquery.
- Scalar subqueries are only supported in numeric contexts. You cannot use a scalar subquery as an argument to the LIKE, REGEXP, or RLIKE operators, or compare it to a value of a non-numeric type such as TIMESTAMP or BOOLEAN.
- You cannot use subqueries with the CASE function to generate the comparison value, the values to be compared against, or the return value.
- A subquery is not allowed in the filter condition for the HAVING clause. (Strictly speaking, a subquery cannot appear anywhere outside the WITH, FROM, and WHERE clauses.)
- You must use a fully qualified name (*table_name.column_name* or *database_name.table_name.column_name*) when referring to any column from the outer query block within a subquery.
- The TABLESAMPLE clause of the SELECT statement does not apply to a table reference derived from a view, a subquery, or anything other than a real base table. This clause only works for tables backed by HDFS or HDFS-like data files, therefore it does not apply to Kudu or HBase tables.

Complex type considerations:

For the complex types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher, the join queries that “unpack” complex type columns often use correlated subqueries in the FROM clause. For example, if the first table in the join clause is CUSTOMER, the second join clause might have a subquery that selects from the column CUSTOMER.C_ORDERS, which is an ARRAY. The subquery re-evaluates the ARRAY elements corresponding to each row from the CUSTOMER table. See Complex Types (CDH 5.5 or higher only) on page 139 for details and examples of using subqueries with complex types.

Related information:

EXISTS Operator on page 176, IN Operator on page 180

TABLESAMPLE Clause

Specify the TABLESAMPLE clause when you need to explore the data distribution within the table, the table is very large, and it is impractical or unnecessary to process all the data from the table or selected partitions.

The clause makes the query process a randomized set of data files from the table, so that the total volume of data is greater than or equal to the specified percentage of data bytes within that table. (Or the data bytes within the set of partitions that remain after partition pruning is performed.)

Syntax:

```
TABLESAMPLE SYSTEM(percentage) [REPEATABLE(seed)]
```

The TABLESAMPLE clause comes immediately after a table name or table alias.

The SYSTEM keyword represents the sampling method. Currently, Impala only supports a single sampling method named SYSTEM.

The *percentage* argument is an integer literal from 0 to 100. A percentage of 0 produces an empty result set for a particular table reference, while a percentage of 100 uses the entire contents. Because the sampling works by selecting a random set of data files, the proportion of sampled data from the table may be greater than the specified percentage, based on the number and sizes of the underlying data files. See the usage notes for details.

The optional REPEATABLE keyword lets you specify an arbitrary positive integer seed value that ensures that when the query is run again, the sampling selects the same set of data files each time. REPEATABLE does not have a default value. If you omit the REPEATABLE keyword, the random seed is derived from the current time.

Added in: CDH 5.12.0 / Impala 2.9.0

See COMPUTE STATS Statement for the TABLESAMPLE clause used in the COMPUTE STATS statement.

Usage notes:

You might use this clause with aggregation queries, such as finding the approximate average, minimum, or maximum where exact precision is not required. You can use these findings to plan the most effective strategy for constructing queries against the full table or designing a partitioning strategy for the data.

Some other database systems have a TABLESAMPLE clause. The Impala syntax for this clause is modeled on the syntax for popular relational databases, not the Hive TABLESAMPLE clause. For example, there is no BUCKETS keyword as in HiveQL.

The precision of the *percentage* threshold depends on the number and sizes of the underlying data files. Impala brings in additional data files, one at a time, until the number of bytes exceeds the specified percentage based on the total number of bytes for the entire set of table data. The precision of the percentage threshold is higher when the table contains many data files with consistent sizes. See the code listings later in this section for examples.

When you estimate characteristics of the data distribution based on sampling a percentage of the table data, be aware that the data might be unevenly distributed between different files. Do not assume that the percentage figure reflects the percentage of rows in the table. For example, one file might contain all blank values for a STRING column, while another file contains long strings in that column; therefore, one file could contain many more rows than another. Likewise, a table created with the SORT BY clause might contain narrow ranges of values for the sort columns, making it impractical to extrapolate the number of distinct values for those columns based on sampling only some of the data files.

Because a sample of the table data might not contain all values for a particular column, if the TABLESAMPLE is used in a join query, the key relationships between the tables might produce incomplete result sets compared to joins using all the table data. For example, if you join 50% of table A with 50% of table B, some values in the join columns might not match between the two tables, even though overall there is a 1:1 relationship between the tables.

The REPEATABLE keyword makes identical queries use a consistent set of data files when the query is repeated. You specify an arbitrary integer key that acts as a seed value when Impala randomly selects the set of data files to use in the query. This technique lets you verify correctness, examine performance, and so on for queries using the TABLESAMPLE clause without the sampled data being different each time. The repeatable aspect is reset (that is, the

set of selected data files may change) any time the contents of the table change. The statements or operations that can make sampling results non-repeatable are:

- `INSERT`.
- `TRUNCATE TABLE`.
- `LOAD DATA`.
- `REFRESH` or `INVALIDATE METADATA` after files are added or removed by a non-Impala mechanism.
-

This clause is similar in some ways to the `LIMIT` clause, because both serve to limit the size of the intermediate data and final result set. `LIMIT 0` is more efficient than `TABLESAMPLE SYSTEM(0)` for verifying that a query can execute without producing any results. `TABLESAMPLE SYSTEM(`*n*`)` often makes query processing more efficient than using a `LIMIT` clause by itself, because all phases of query execution use less data overall. If the intent is to retrieve some representative values from the table in an efficient way, you might combine `TABLESAMPLE`, `ORDER BY`, and `LIMIT` clauses within a single query.

Partitioning:

When you query a partitioned table, any partition pruning happens before Impala selects the data files to sample. For example, in a table partitioned by year, a query with `WHERE year = 2017` and a `TABLESAMPLE SYSTEM(10)` clause would sample data files representing at least 10% of the bytes present in the 2017 partition.

Amazon S3 considerations:

This clause applies to S3 tables the same way as tables with data files stored on HDFS.

ADLS considerations:

This clause applies to ADLS tables the same way as tables with data files stored on HDFS.

Kudu considerations:

This clause does not apply to Kudu tables.

HBase considerations:

This clause does not apply to HBase tables.

Performance considerations:

From a performance perspective, the `TABLESAMPLE` clause is especially valuable for exploratory queries on text, Avro, or other file formats other than Parquet. Text-based or row-oriented file formats must process substantial amounts of redundant data for queries that derive aggregate results such as `MAX()`, `MIN()`, or `AVG()` for a single column. Therefore, you might use `TABLESAMPLE` early in the ETL pipeline, when data is still in raw text format and has not been converted to Parquet or moved into a partitioned table.

Restrictions:

This clause applies only to tables that use a storage layer with underlying raw data files, such as HDFS, Amazon S3, or Microsoft ADLS.

This clause does not apply to table references that represent views. A query that applies the `TABLESAMPLE` clause to a view or a subquery fails with a semantic error.

Because the sampling works at the level of entire data files, it is by nature coarse-grained. It is possible to specify a small sample percentage but still process a substantial portion of the table data if the table contains relatively few data files, if each data file is very large, or if the data files vary substantially in size. Be sure that you understand the data distribution and physical file layout so that you can verify if the results are suitable for extrapolation. For example, if the table contains only a single data file, the “sample” will consist of all the table data regardless of the percentage you specify. If the table contains data files of 1 GiB, 1 GiB, and 1 KiB, when you specify a sampling percentage of 50 you would either process slightly more than 50% of the table (1 GiB + 1 KiB) or almost the entire table (1 GiB + 1 GiB), depending on which data files were selected for sampling.

If data files are added by a non-Impala mechanism, and the table metadata is not updated by a REFRESH or INVALIDATE METADATA statement, the TABLESAMPLE clause does not consider those new files when computing the number of bytes in the table or selecting which files to sample.

If data files are removed by a non-Impala mechanism, and the table metadata is not updated by a REFRESH or INVALIDATE METADATA statement, the query fails if the TABLESAMPLE clause attempts to reference any of the missing files.

Examples:

The following examples demonstrate the TABLESAMPLE clause. These examples intentionally use very small data sets to illustrate how the number of files, size of each file, and overall size of data in the table interact with the percentage specified in the clause.

These examples use an unpartitioned table, containing several files of roughly the same size:

```
create table sample_demo (x int, s string);

insert into sample_demo values (1, 'one');
insert into sample_demo values (2, 'two');
insert into sample_demo values (3, 'three');
insert into sample_demo values (4, 'four');
insert into sample_demo values (5, 'five');

show files in sample_demo;
+---------------------+------+-----------+
| Path                | Size | Partition |
+---------------------+------+-----------+
| 991213608_data.0.   | 7B   |           |
| 982196806_data.0.   | 6B   |           |
| _2122096884_data.0. | 8B   |           |
| _586325431_data.0.  | 6B   |           |
| 1894746258_data.0.  | 7B   |           |
+---------------------+------+-----------+

show table stats sample_demo;
+-------+--------+------+--------+-------------------------+
| #Rows | #Files | Size | Format | Location                |
+-------+--------+------+--------+-------------------------+
| -1    | 5      | 34B  | TEXT   | /tsample.db/sample_demo |
+-------+--------+------+--------+-------------------------+
```

A query that samples 50% of the table must process at least 17 bytes of data. Based on the sizes of the data files, we can predict that each such query uses 3 arbitrary files. Any 1 or 2 files are not enough to reach 50% of the total data in the table (34 bytes), so the query adds more files until it passes the 50% threshold:

```
select distinct x from sample_demo tablesample system(50);
+---+
| x |
+---+
| 4 |
| 1 |
| 5 |
+---+

select distinct x from sample_demo tablesample system(50);
+---+
| x |
+---+
| 5 |
| 4 |
| 2 |
+---+

select distinct x from sample_demo tablesample system(50);
+---+
| x |
+---+
| 5 |
```

```
| 3 |
| 2 |
+---+
```

To help run reproducible experiments, the REPEATABLE clause causes Impala to choose the same set of files for each query. Although the data set being considered is deterministic, the order of results varies (in the absence of an ORDER BY clause) because of the way distributed queries are processed:

```
select distinct x from sample_demo
  tablesample system(50) repeatable (12345);
+---+
| x |
+---+
| 3 |
| 2 |
| 1 |
+---+

select distinct x from sample_demo
  tablesample system(50) repeatable (12345);
+---+
| x |
+---+
| 2 |
| 1 |
| 3 |
+---+
```

The following examples show how uneven data distribution affects which data is sampled. Adding another data file containing a long string value changes the threshold for 50% of the total data in the table:

```
insert into sample_demo values
(1000, 'Boyhood is the longest time in life for a boy. The last term of the school-year
 is made of decades, not of weeks, and living through them is like waiting for the
millennium. Booth Tarkington');

show files in sample_demo;
+---------------------+------+-----------+
| Path                | Size | Partition |
+---------------------+------+-----------+
| 991213608_data.0.   | 7B   |           |
| 982196806_data.0.   | 6B   |           |
| _253317650_data.0.  | 196B |           |
| _2122096884_data.0. | 8B   |           |
| _586325431_data.0.  | 6B   |           |
| 1894746258_data.0.  | 7B   |           |
+---------------------+------+-----------+

show table stats sample_demo;
+-------+--------+------+--------+-------------------------+
| #Rows | #Files | Size | Format | Location                |
+-------+--------+------+--------+-------------------------+
| -1    | 6      | 230B | TEXT   | /tsample.db/sample_demo |
+-------+--------+------+--------+-------------------------+
```

Even though the queries do not refer to the S column containing the long value, all the sampling queries include the data file containing the column value X=1000, because the query cannot reach the 50% threshold (115 bytes) without including that file. The large file might be considered first, in which case it is the only file processed by the query. Or an arbitrary set of other files might be considered first.

```
select distinct x from sample_demo tablesample system(50);
+------+
| x    |
+------+
| 1000 |
| 3    |
| 1    |
+------+
```

```
select distinct x from sample_demo tablesample system(50);
+------+
| x    |
+------+
| 1000 |
+------+

select distinct x from sample_demo tablesample system(50);
+------+
| x    |
+------+
| 1000 |
| 4    |
| 2    |
| 1    |
+------+
```

The following examples demonstrate how the TABLESAMPLE clause interacts with other table aspects, such as partitioning and file format:

```
create table sample_demo_partitions (x int, s string) partitioned by (n int) stored as
 parquet;

insert into sample_demo_partitions partition (n = 1) select * from sample_demo;
insert into sample_demo_partitions partition (n = 2) select * from sample_demo;
insert into sample_demo_partitions partition (n = 3) select * from sample_demo;

show files in sample_demo_partitions;
+---------------------------------+--------+-----------+
| Path                            | Size   | Partition |
+---------------------------------+--------+-----------+
| 000000_364262785_data.0.parq    | 1.24KB | n=1       |
| 000001_973526736_data.0.parq    | 566B   | n=1       |
| 0000000_1300598134_data.0.parq  | 1.24KB | n=2       |
| 0000001_689099063_data.0.parq   | 568B   | n=2       |
| 0000000_1861371709_data.0.parq  | 1.24KB | n=3       |
| 0000001_1065507912_data.0.parq  | 566B   | n=3       |
+---------------------------------+--------+-----------+

show table stats tablesample_demo_partitioned;
+-------+-------+--------+--------+---------+-----------------------------------------------+
| n     | #Rows | #Files | Size   | Format  | Location
     |
+-------+-------+--------+--------+---------+-----------------------------------------------+
| 1     | -1    | 2      | 1.79KB | PARQUET | /tsample.db/tablesample_demo_partitioned/n=1
  |
| 2     | -1    | 2      | 1.80KB | PARQUET | /tsample.db/tablesample_demo_partitioned/n=2
  |
| 3     | -1    | 2      | 1.79KB | PARQUET | /tsample.db/tablesample_demo_partitioned/n=3
  |
| Total | -1    | 6      | 5.39KB |         |
     |
+-------+-------+--------+--------+---------+-----------------------------------------------+
```

If the query does not involve any partition pruning, the sampling applies to the data volume of the entire table:

```
-- 18 rows total.
select count(*) from sample_demo_partitions;
+----------+
| count(*) |
+----------+
| 18       |
+----------+

-- The number of rows per data file is not
-- perfectly balanced, therefore the count
-- is different depending on which set of files
-- is considered.
select count(*) from sample_demo_partitions
```

```
  tablesample system(75);
+----------+
| count(*) |
+----------+
| 14       |
+----------+

select count(*) from sample_demo_partitions
  tablesample system(75);
+----------+
| count(*) |
+----------+
| 16       |
+----------+
```

If the query only processes certain partitions, the query computes the sampling threshold based on the data size and set of files only from the relevant partitions:

```
select count(*) from sample_demo_partitions
  tablesample system(50) where n = 1;
+----------+
| count(*) |
+----------+
| 6        |
+----------+

select count(*) from sample_demo_partitions
  tablesample system(50) where n = 1;
+----------+
| count(*) |
+----------+
| 2        |
+----------+
```

Related information:

SELECT Statement on page 295

WITH Clause

A clause that can be added before a SELECT statement, to define aliases for complicated expressions that are referenced multiple times within the body of the SELECT. Similar to CREATE VIEW, except that the table and column names defined in the WITH clause do not persist after the query finishes, and do not conflict with names used in actual tables or views. Also known as “subquery factoring”.

You can rewrite a query using subqueries to work the same as with the WITH clause. The purposes of the WITH clause are:

- Convenience and ease of maintenance from less repetition with the body of the query. Typically used with queries involving UNION, joins, or aggregation functions where the similar complicated expressions are referenced multiple times.
- SQL code that is easier to read and understand by abstracting the most complex part of the query into a separate block.
- Improved compatibility with SQL from other database systems that support the same clause (primarily Oracle Database).

> **Note:**
>
> The Impala WITH clause does not support recursive queries in the WITH, which is supported in some other database systems.

Standards compliance: Introduced in SQL:1999.

Examples:

```
-- Define 2 subqueries that can be referenced from the body of a longer query.
with t1 as (select 1), t2 as (select 2) insert into tab select * from t1 union all select
 * from t2;

-- Define one subquery at the outer level, and another at the inner level as part of
the
-- initial stage of the UNION ALL query.
with t1 as (select 1) (with t2 as (select 2) select * from t2) union all select * from
 t1;
```

DISTINCT Operator

The DISTINCT operator in a SELECT statement filters the result set to remove duplicates.

```
-- Returns the unique values from one column.
-- NULL is included in the set of values if any rows have a NULL in this column.
SELECT DISTINCT c_birth_country FROM customer;

-- Returns the unique combinations of values from multiple columns.
SELECT DISTINCT c_salutation, c_last_name FROM customer;
```

You can use DISTINCT in combination with an aggregation function, typically COUNT( ), to find how many different values a column contains.

```
-- Counts the unique values from one column.
-- NULL is not included as a distinct value in the count.
SELECT COUNT(DISTINCT c_birth_country) FROM customer;

-- Counts the unique combinations of values from multiple columns.
SELECT COUNT(DISTINCT c_salutation, c_last_name) FROM customer;
```

Zero-length strings: For purposes of clauses such as DISTINCT and GROUP BY, Impala considers zero-length strings (""), NULL, and space to all be different values.



Note:

In contrast with some database systems that always return DISTINCT values in sorted order, Impala does not do any ordering of DISTINCT values. Always include an ORDER BY clause if you need the values in alphabetical or numeric sorted order.

SET Statement

The SET statement specifies values for query options that control the runtime behavior of other statements within the same session.

When issued in impala-shell, the SET command is interpreted as an impala-shell command that has differences from the SQL SET statement. See impala-shell Command Reference on page 721 for the information about the SET command in impala-shell.

Syntax:

```
SET
SET ALL
SET query_option=option_value
SET query_option=""
```

SET and SET ALL with no arguments return a result set consisting of all the applicable query options and their current values.

The *query_option* and *option_value* are case-insensitive.

Unlike the `impala-shell` command version of SET, when used as a SQL statement, the string values for *option_value* need to be quoted, e.g. `SET option="new_value"`.

The `SET query_option = ""` statement unsets the value of the *query_option* in the current session, reverting it to the default state. In `impala-shell`, use the `UNSET` command to set a query option to its default.

Each query option has a specific allowed notation for its arguments. See Query Options for the SET Statement on page 322 for the details of each query option.

Usage notes:

In CDH 5.14 / Impala 2.11 and higher, the outputs of the SET and `SET ALL` statements were reorganized as below:

- The options are divided into groups: `Regular Query Options`, `Advanced Query Options`, `Development Query Options`, and `Deprecated Query Options`.
  - The advanced options are intended for use in specific kinds of performance tuning and debugging scenarios.
  - The development options are related to internal development of Impala or features that are not yet finalized. These options might be changed or removed without notice.
  - The deprecated options are related to features that are removed or changed so that the options no longer have any purpose. These options might be removed in future versions.
- By default, only the first two groups, regular and advanced, are displayed by the SET command. Use `SET ALL` to see all groups of options.
- `impala-shell` options and user-specified variables are always displayed at the end of the list of query options, after all appropriate option groups.

Added in: CDH 5.2.0 / Impala 2.0.0

SET has always been available as an `impala-shell` command. Promoting it to a SQL statement lets you use this feature in client applications through the JDBC and ODBC APIs.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Cloudera Manager: You can set any of the query options globally in Cloudera Manager to affect all the queries in the cluster.

1. Navigate to ***Impala cluster*** > **Configuration** > **Impala Daemon Query Options Advanced Configuration Snippet (Safety Valve)**
2. In the field, type a key-value pair of a query option and the value, e.g. `EXPLAIN_LEVEL=2`.
3. Click **+** to enter an additional option.
4. Clive **Save Changes**.

Related information:

See Query Options for the SET Statement on page 322 for the query options you can adjust using this statement.

Query Options for the SET Statement

You can specify the following options using the SET statement, and those settings affect all queries issued from that session.

Some query options are useful in day-to-day operations for improving usability, performance, or flexibility.

Other query options control special-purpose aspects of Impala operation and are intended primarily for advanced debugging or troubleshooting.

Options with Boolean parameters can be set to 1 or `true` to enable, or 0 or `false` to turn off.



Note:

In Impala 2.0 and later, you can set query options directly through the JDBC and ODBC interfaces by using the SET statement. Formerly, SET was only available as a command within the `impala-shell` interpreter.

In CDH 5.14 / Impala 2.11 and later, you can set query options for an `impala-shell` session by specifying one or more command-line arguments of the form `--query_option=option=value`. See impala-shell Configuration Options on page 714 for details.

Related information:

SET Statement on page 321

ABORT_ON_ERROR Query Option

When this option is enabled, Impala cancels a query immediately when any of the nodes encounters an error, rather than continuing and possibly returning incomplete results. This option is disabled by default, to help gather maximum diagnostic information when an error occurs, for example, whether the same problem occurred on all nodes or only a single node. Currently, the errors that Impala can skip over involve data corruption, such as a column that contains a string value when expected to contain an integer value.

To control how much logging Impala does for non-fatal errors when ABORT_ON_ERROR is turned off, use the MAX_ERRORS option.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of SET statement)

Related information:

MAX_ERRORS Query Option on page 339, Using Impala Logging on page 709

ALLOW_ERASURE_CODED_FILES Query Option

Use the ALLOW_ERASURE_CODED_FILES query option to enable or disable the support of erasure coded files in Impala. Until Impala is fully tested and certified with erasure coded files, this query option is set to FALSE by default.

When the ALLOW_ERASURE_CODED_FILES query option is set to FALSE, Impala returns an error when a query requires scanning an erasure coded file.

Type: BOOLEAN

Default: FALSE

Added in: CDH 6.1

ALLOW_UNSUPPORTED_FORMATS Query Option



Note: This query option was removed in CDH 6.1 and no longer has any effect. Do not use.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of SET statement)

APPX_COUNT_DISTINCT Query Option (CDH 5.2 or higher only)

When the APPX_COUNT_DISTINCT query option is set to TRUE, Impala implicitly converts COUNT(DISTINCT) operations to the NDV( ) function calls. The resulting count is approximate rather than precise. Enable the query option when a tolerable amount of error is acceptable in order to obtain faster query results than with a COUNT (DISTINCT) queries.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of SET statement)

Related information:

COUNT Function on page 485, DISTINCT Operator on page 321, NDV Function on page 497

BATCH_SIZE Query Option

Number of rows evaluated at a time by SQL operators. Unspecified or a size of 0 uses a predefined default size. Using a large number improves responsiveness, especially for scan operations, at the cost of a higher memory footprint.

This option is primarily for testing during Impala development, or for use under the direction of Cloudera Support.

Type: numeric

Default: 0 (meaning the predefined default of 1024)

BUFFER_POOL_LIMIT Query Option

Defines a limit on the amount of memory that a query can allocate from the internal buffer pool. The value for this limit applies to the memory on each host, not the aggregate memory across the cluster. Typically not changed by users, except during diagnosis of out-of-memory errors during queries.

Type: integer

Default:

The default setting for this option is the lower of 80% of the `MEM_LIMIT` setting, or the `MEM_LIMIT` setting minus 32 MB.

Added in: CDH 5.13.0 / Impala 2.10.0

Usage notes:

If queries encounter out-of-memory errors, consider decreasing the `BUFFER_POOL_LIMIT` setting to less than 80% of the `MEM_LIMIT setting`.

Examples:

```
-- Set an absolute value.
set buffer_pool_limit=8GB;

-- Set a relative value based on the MEM_LIMIT setting.
set buffer_pool_limit=80%;
```

Related information:

DEFAULT_SPILLABLE_BUFFER_SIZE Query Option on page 327, MAX_ROW_SIZE Query Option on page 340, MIN_SPILLABLE_BUFFER_SIZE Query Option on page 345, Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher) on page 607

COMPRESSION_CODEC Query Option (CDH 5.2 or higher only)

When Impala writes Parquet data files using the `INSERT` statement, the underlying compression is controlled by the `COMPRESSION_CODEC` query option.

> **Note:** Prior to Impala 2.0, this option was named `PARQUET_COMPRESSION_CODEC`. In Impala 2.0 and later, the `PARQUET_COMPRESSION_CODEC` name is not recognized. Use the more general name `COMPRESSION_CODEC` for new code.

Syntax:

```
SET COMPRESSION_CODEC=codec_name; // Supported for all codecs.
SET COMPRESSION_CODEC=codec_name:compression_level; // Only supported for ZSTD.
```

The allowed values for this query option are `SNAPPY` (the default), `GZIP`, `ZSTD`, and `NONE`.

ZSTD also supports setting a compression level. The lower the level, the faster the speed at the cost of compression ratio. Compression levels from 1 up to 22 are supported for ZSTD. The default compression level 3 is used, if one is not passed using the compression_codec query option.



Note: A Parquet file created with COMPRESSION_CODEC=NONE is still typically smaller than the original data, due to encoding schemes such as run-length encoding and dictionary encoding that are applied separately from compression.

The option value is not case-sensitive.

If the option is set to an unrecognized value, all kinds of queries will fail due to the invalid option setting, not just queries involving Parquet tables. (The value BZIP2 is also recognized, but is not compatible with Parquet tables.)

Type: STRING

Default: SNAPPY

Examples:

```
set compression_codec=zstd; // Default compression level 3.
insert into parquet_table_zstd_default_compressed select * from t1;

set compression_codec=zstd:12; // Compression level 12.
insert into parquet_table_zstd_highly_compressed select * from t1;

set compression_codec=gzip;
insert into parquet_table_highly_compressed select * from t1;

set compression_codec=snappy;
insert into parquet_table_compression_plus_fast_queries select * from t1;

set compression_codec=none;
insert into parquet_table_no_compression select * from t1;

set compression_codec=foo;
select * from t1 limit 5;
ERROR: Invalid compression codec: foo
```

Related information:

For information about how compressing Parquet data files affects query performance, see Compressions for Parquet Data Files on page 647.

COMPUTE_STATS_MIN_SAMPLE_SIZE Query Option

The COMPUTE_STATS_MIN_SAMPLE_SIZE query option specifies the minimum number of bytes that will be scanned in COMPUTE STATS TABLESAMPLE, regardless of the user-supplied sampling percent. This query option prevents sampling for very small tables where accurate stats can be obtained cheaply without sampling because the minimum sample size is required to get meaningful stats.

Type: integer

Default: 1GB

Added in: CDH 5.15 / Impala 2.12

Usage notes:

DEBUG_ACTION Query Option

Introduces artificial problem conditions within queries. For internal Cloudera debugging and troubleshooting.

Type: STRING

Default: empty string

DECIMAL_V2 Query Option

A query option that changes behavior related to the DECIMAL data type. Set this option to FALSE for backward compatibility to Impala 2.x.

Type: Boolean

Default: TRUE

DEFAULT_JOIN_DISTRIBUTION_MODE Query Option

This option determines the join distribution that Impala uses when any of the tables involved in a join query is missing statistics.

Impala optimizes join queries based on the presence of table statistics, which are produced by the Impala COMPUTE STATS statement. By default, when a table involved in the join query does not have statistics, Impala uses the "broadcast" technique that transmits the entire contents of the table to all executor nodes participating in the query. If one table involved in a join has statistics and the other does not, the table without statistics is broadcast. If both tables are missing statistics, the table on the right-hand side of the join is broadcast. This behavior is appropriate when the table involved is relatively small, but can lead to excessive network, memory, and CPU overhead if the table being broadcast is large.

Because Impala queries frequently involve very large tables, and suboptimal joins for such tables could result in spilling or out-of-memory errors, the setting DEFAULT_JOIN_DISTRIBUTION_MODE=SHUFFLE lets you override the default behavior. The shuffle join mechanism divides the corresponding rows of each table involved in a join query using a hashing algorithm, and transmits subsets of the rows to other nodes for processing. Typically, this kind of join is more efficient for joins between large tables of similar size.

The setting DEFAULT_JOIN_DISTRIBUTION_MODE=SHUFFLE is recommended when setting up and deploying new clusters, because it is less likely to result in serious consequences such as spilling or out-of-memory errors if the query plan is based on incomplete information. This setting is not the default, to avoid changing the performance characteristics of join queries for clusters that are already tuned for their existing workloads.

Type: integer

The allowed values are BROADCAST (equivalent to 0) or SHUFFLE (equivalent to 1).

Examples:

The following examples demonstrate appropriate scenarios for each setting of this query option.

```
-- Create a billion-row table.
create table big_table stored as parquet
  as select * from huge_table limit 1e9;

-- For a big table with no statistics, the
-- shuffle join mechanism is appropriate.
set default_join_distribution_mode=shuffle;

...join queries involving the big table...
```

```
-- Create a hundred-row table.
create table tiny_table stored as parquet
  as select * from huge_table limit 100;

-- For a tiny table with no statistics, the
-- broadcast join mechanism is appropriate.
set default_join_distribution_mode=broadcast;

...join queries involving the tiny table...
```

```
compute stats tiny_table;
compute stats big_table;

-- Once the stats are computed, the query option has
```

```
-- no effect on join queries involving these tables.
-- Impala can determine the absolute and relative sizes
-- of each side of the join query by examining the
-- row size, cardinality, and so on of each table.

...join queries involving both of these tables...
```

Related information:

COMPUTE STATS Statement on page 219, Joins in Impala SELECT Statements on page 296, Performance Considerations for Join Queries on page 568

DEFAULT_SPILLABLE_BUFFER_SIZE Query Option

Specifies the default size for a memory buffer used when the spill-to-disk mechanism is activated, for example for queries against a large table with no statistics, or large join operations.

Type: integer

Default:

2097152 (2 MB)

Units: A numeric argument represents a size in bytes; you can also use a suffix of m or mb for megabytes, or g or gb for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Added in: CDH 5.13.0 / Impala 2.10.0

Usage notes:

This query option sets an upper bound on the size of the internal buffer size that can be used during spill-to-disk operations. The actual size of the buffer is chosen by the query planner.

If overall query performance is limited by the time needed for spilling, consider increasing the DEFAULT_SPILLABLE_BUFFER_SIZE setting. Larger buffer sizes result in Impala issuing larger I/O requests to storage devices, which might result in higher throughput, particularly on rotational disks.

The tradeoff with a large value for this setting is increased memory usage during spill-to-disk operations. Reducing this value may reduce memory consumption.

To determine if the value for this setting is having an effect by capping the spillable buffer size, you can see the buffer size chosen by the query planner for a particular query. EXPLAIN the query while the setting EXPLAIN_LEVEL=2 is in effect.

Examples:

```
set default_spillable_buffer_size=4MB;
```

Related information:

BUFFER_POOL_LIMIT Query Option on page 324, MAX_ROW_SIZE Query Option on page 340, MIN_SPILLABLE_BUFFER_SIZE Query Option on page 345, Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher) on page 607

DISABLE_CODEGEN Query Option

The DISABLE_CODEGEN is a debug option, and it's used to work around any issues with Impala's runtime code generation. If a query fails with an “illegal instruction” or other hardware-specific message, try setting DISABLE_CODEGEN=true and running the query again. If the query succeeds only when the DISABLE_CODEGEN option is turned on, submit the problem to Cloudera Support and include that detail in the problem report.

Most queries will run significantly slower with DISABLE_CODEGEN=true.

In Impala 2.10 and higher, the DISABLE_CODEGEN_ROWS_THRESHOLD optimisation automatically disables codegen for small queries because short-running queries may run faster without the overhead of codegen.

The following values are supported:

- TRUE or 1: Disables codegen.
- FALSE or 0: Enables codegen.

Type: Boolean

Default: `false` (shown as 0 in output of SET statement)

DISABLE_CODEGEN_ROWS_THRESHOLD Query Option (CDH 5.13 / Impala 2.10 or higher only)

This setting controls the cutoff point (in terms of number of rows processed per Impala daemon) below which Impala disables native code generation for the whole query. Native code generation is very beneficial for queries that process many rows because it reduces the time taken to process of each row. However, generating the native code adds latency to query startup. Therefore, automatically disabling codegen for queries that process relatively small amounts of data can improve query response time.

Syntax:

```
SET DISABLE_CODEGEN_ROWS_THRESHOLD=number_of_rows
```

Type: numeric

Default: 50000

Usage notes: Typically, you increase the default value to make this optimization apply to more queries. If incorrect or corrupted table and column statistics cause Impala to apply this optimization incorrectly to queries that actually involve substantial work, you might see the queries being slower as a result of codegen being disabled. In that case, recompute statistics with the `COMPUTE STATS` or `COMPUTE INCREMENTAL STATS` statement. If there is a problem collecting accurate statistics, you can turn this feature off by setting the value to 0.

Internal details:

This setting applies to queries where the number of rows processed can be accurately determined, either through table and column statistics, or by the presence of a `LIMIT` clause. If Impala cannot accurately estimate the number of rows, then this setting does not apply.

If a query uses the complex data types `STRUCT`, `ARRAY`, or `MAP`, then codegen is never automatically disabled regardless of the `DISABLE_CODEGEN_ROWS_THRESHOLD` setting.

Added in: CDH 5.13.0 / Impala 2.10.0

DISABLE_ROW_RUNTIME_FILTERING Query Option (CDH 5.7 or higher only)

The `DISABLE_ROW_RUNTIME_FILTERING` query option reduces the scope of the runtime filtering feature. Queries still dynamically prune partitions, but do not apply the filtering logic to individual rows within partitions.

Only applies to queries against Parquet tables. For other file formats, Impala only prunes at the level of partitions, not individual rows.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false`

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

Impala automatically evaluates whether the per-row filters are being effective at reducing the amount of intermediate data. Therefore, this option is typically only needed for the rare case where Impala cannot accurately determine how effective the per-row filtering is for a query.

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Because this setting only improves query performance in very specific circumstances, depending on the query characteristics and data distribution, only use it when you determine through benchmarking that it improves performance of specific expensive queries. Consider setting this query option immediately before the expensive query and unsetting it immediately afterward.

File format considerations:

This query option only applies to queries against HDFS-based tables using the Parquet file format.

Kudu considerations:

When applied to a query involving a Kudu table, this option turns off all runtime filtering for the Kudu table.

Related information:

Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588, RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358

DISABLE_STREAMING_PREAGGREGATIONS Query Option (CDH 5.7 or higher only)

Turns off the "streaming preaggregation" optimization that is available in CDH 5.7 / Impala 2.5 and higher. This optimization reduces unnecessary work performed by queries that perform aggregation operations on columns with few or no duplicate values, for example `DISTINCT id_column` or `GROUP BY unique_column`. If the optimization causes regressions in existing queries that use aggregation functions, you can turn it off as needed by setting this query option.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of SET statement)



Note: In CDH 5.7.0 / Impala 2.5.0, only the value 1 enables the option, and the value `true` is not recognized. This limitation is tracked by the issue IMPALA-3334, which shows the releases where the problem is fixed.

Usage notes:

Typically, queries that would require enabling this option involve very large numbers of aggregated values, such as a billion or more distinct keys being processed on each worker node.

Added in: CDH 5.7.0 / Impala 2.5.0

DISABLE_UNSAFE_SPILLS Query Option (CDH 5.2 or higher only)

Enable this option if you prefer to have queries fail when they exceed the Impala memory limit, rather than write temporary data to disk.

Queries that "spill" to disk typically complete successfully, when in earlier Impala releases they would have failed. However, queries with exorbitant memory requirements due to missing statistics or inefficient join clauses could become so slow as a result that you would rather have them cancelled automatically and reduce the memory usage through standard Impala tuning techniques.

This option prevents only "unsafe" spill operations, meaning that one or more tables are missing statistics or the query does not include a hint to set the most efficient mechanism for a join or `INSERT ... SELECT` into a partitioned table. These are the tables most likely to result in suboptimal execution plans that could cause unnecessary spilling. Therefore, leaving this option enabled is a good way to find tables on which to run the `COMPUTE STATS` statement.

See SQL Operations that Spill to Disk on page 607 for information about the "spill to disk" feature for queries processing large result sets with joins, `ORDER BY`, `GROUP BY`, `DISTINCT`, aggregation functions, or analytic functions.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of SET statement)

Added in: CDH 5.2.0 / Impala 2.0.0

ENABLE_EXPR_REWRITES Query Option

The ENABLE_EXPR_REWRITES query option controls whether to enable or disable the query compile time optimizations. These optimizations rewrite the expression trees to a more compact and optimized form that helps avoid redundant expression evaluation at run time. Performance optimizations controlled by this query option include:

- Constant folding (added in)
- Extracting common conjuncts from disjunctions (added in)
- Simplify conditionals with constant conditions (added in)

Set the option to `false` or `0` to disable the performance optimizations.

Type: `boolean`

Default: `true` (1)

Added in: CDH 5.10

EXEC_SINGLE_NODE_ROWS_THRESHOLD Query Option (CDH 5.3 or higher only)

This setting controls the cutoff point (in terms of number of rows scanned) below which Impala treats a query as a "small" query, turning off optimizations such as parallel execution and native code generation. The overhead for these optimizations is applicable for queries involving substantial amounts of data, but it makes sense to skip them for queries involving tiny amounts of data. Reducing the overhead for small queries allows Impala to complete them more quickly, keeping admission control slots, CPU, memory, and so on available for resource-intensive queries.

Syntax:

```
SET EXEC_SINGLE_NODE_ROWS_THRESHOLD=number_of_rows
```

Type: numeric

Default: 100

Usage notes: Typically, you increase the default value to make this optimization apply to more queries. If incorrect or corrupted table and column statistics cause Impala to apply this optimization incorrectly to queries that actually involve substantial work, you might see the queries being slower as a result of remote reads. In that case, recompute statistics with the `COMPUTE STATS` or `COMPUTE INCREMENTAL STATS` statement. If there is a problem collecting accurate statistics, you can turn this feature off by setting the value to -1.

Internal details:

This setting applies to queries where the number of rows processed can be accurately determined, either through table and column statistics, or by the presence of a `LIMIT` clause. If Impala cannot accurately estimate the number of rows, then this setting does not apply.

In CDH 5.5 / Impala 2.3 and higher, where Impala supports the complex data types `STRUCT`, `ARRAY`, and `MAP`, if a query refers to any column of those types, the small-query optimization is turned off for that query regardless of the `EXEC_SINGLE_NODE_ROWS_THRESHOLD` setting.

For a query that is determined to be "small", all work is performed on the coordinator node. This might result in some I/O being performed by remote reads. The savings from not distributing the query work and not generating native code are expected to outweigh any overhead from the remote reads.

Added in: CDH 5.13

Examples:

A common use case is to query just a few rows from a table to inspect typical data values. In this example, Impala does not parallelize the query or perform native code generation because the result set is guaranteed to be smaller than the threshold value from this query option:

```
SET EXEC_SINGLE_NODE_ROWS_THRESHOLD=500;
SELECT * FROM enormous_table LIMIT 300;
```

EXEC_TIME_LIMIT_S Query Option (CDH 5.15 / Impala 2.12 or higher only)

The EXEC_TIME_LIMIT_S query option sets a time limit on query execution. If a query is still executing when time limit expires, it is automatically canceled. The option is intended to prevent runaway queries that execute for much longer than intended.

For example, an Impala administrator could set a default value of EXEC_TIME_LIMIT_S=3600 for a resource pool to automatically kill queries that execute for longer than one hour (see Admission Control and Query Queuing on page 549 for information about default query options). Then, if a user accidentally runs a large query that executes for more than one hour, it will be automatically killed after the time limit expires to free up resources. Users can override the default value per query or per session if they do not want the default EXEC_TIME_LIMIT_S value to apply to a specific query or a session.



Note:

The time limit only starts once the query is executing. Time spent planning the query, scheduling the query, or in admission control is not counted towards the execution time limit. SELECT statements are eligible for automatic cancellation until the client has fetched all result rows. DML queries are eligible for automatic cancellation until the DML statement has finished.

Syntax:

```
SET EXEC_TIME_LIMIT_S=seconds;
```

Type: numeric

Default: 0 (no time limit)

Added in: CDH 5.15 / Impala 2.12

Related information:

Setting Timeout Periods for Daemons, Queries, and Sessions on page 70

EXPLAIN_LEVEL Query Option

Controls the amount of detail provided in the output of the EXPLAIN statement. The basic output can help you identify high-level performance issues such as scanning a higher volume of data or more partitions than you expect. The higher levels of detail show how intermediate results flow between nodes and how different SQL operations such as ORDER BY, GROUP BY, joins, and WHERE clauses are implemented within a distributed query.

Type: STRING or INT

Default: 1

Arguments:

The allowed range of numeric values for this option is 0 to 3:

- 0 or MINIMAL: A barebones list, one line per operation. Primarily useful for checking the join order in very long queries where the regular EXPLAIN output is too long to read easily.
- 1 or STANDARD: The default level of detail, showing the logical way that work is split up for the distributed query.
- 2 or EXTENDED: Includes additional detail about how the query planner uses statistics in its decision-making process, to understand how a query could be tuned by gathering statistics, using query hints, adding or removing predicates, and so on. In CDH 6.2 / Impala 3.2 and higher, the output also includes the analyzed query with the cast information in the output header, and the implicit cast info in the Predicate section.
- 3 or VERBOSE: The maximum level of detail, showing how work is split up within each node into “query fragments” that are connected in a pipeline. This extra detail is primarily useful for low-level performance testing and tuning within Impala itself, rather than for rewriting the SQL code at the user level.



Note: Prior to Impala 1.3, the allowed argument range for EXPLAIN_LEVEL was 0 to 1: level 0 had the mnemonic NORMAL, and level 1 was VERBOSE. In Impala 1.3 and higher, NORMAL is not a valid mnemonic value, and VERBOSE still applies to the highest level of detail but now corresponds to level 3. You might need to adjust the values if you have any older impala-shell script files that set the EXPLAIN_LEVEL query option.

Changing the value of this option controls the amount of detail in the output of the EXPLAIN statement. The extended information from level 2 or 3 is especially useful during performance tuning, when you need to confirm whether the work for the query is distributed the way you expect, particularly for the most resource-intensive operations such as join queries against large tables, queries against tables with large numbers of partitions, and insert operations for Parquet tables. The extended information also helps to check estimated resource usage when you use the admission control or resource management features explained in Resource Management on page 549. See EXPLAIN Statement on page 271 for the syntax of the EXPLAIN statement, and Using the EXPLAIN Plan for Performance Tuning on page 602 for details about how to use the extended information.

Usage notes:

As always, read the EXPLAIN output from bottom to top. The lowest lines represent the initial work of the query (scanning data files), the lines in the middle represent calculations done on each node and how intermediate results are transmitted from one node to another, and the topmost lines represent the final results being sent back to the coordinator node.

The numbers in the left column are generated internally during the initial planning phase and do not represent the actual order of operations, so it is not significant if they appear out of order in the EXPLAIN output.

At all EXPLAIN levels, the plan contains a warning if any tables in the query are missing statistics. Use the COMPUTE STATS statement to gather statistics for each table and suppress this warning. See Table and Column Statistics on page 575 for details about how the statistics help query performance.

The PROFILE command in impala-shell always starts with an explain plan showing full detail, the same as with EXPLAIN_LEVEL=3. After the explain plan comes the executive summary, the same output as produced by the SUMMARY command in impala-shell.

Examples:

These examples use a trivial, empty table to illustrate how the essential aspects of query planning are shown in EXPLAIN output:

```
[localhost:21000] > create table t1 (x int, s string);
[localhost:21000] > set explain_level=1;
[localhost:21000] > explain select count(*) from t1;
+------------------------------------------------------------------------+
| Explain String                                                         |
+------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=10.00MB VCores=1               |
| WARNING: The following tables are missing relevant table and/or column |
|   statistics.                                                          |
| explain_plan.t1                                                        |
|                                                                        |
| 03:AGGREGATE [MERGE FINALIZE]                                          |
| |  output: sum(count(*))                                               |
| |                                                                      |
| 02:EXCHANGE [PARTITION=UNPARTITIONED]                                  |
| |                                                                      |
| 01:AGGREGATE                                                           |
| |  output: count(*)                                                    |
| |                                                                      |
| 00:SCAN HDFS [explain_plan.t1]                                         |
|    partitions=1/1 size=0B                                              |
+------------------------------------------------------------------------+
[localhost:21000] > explain select * from t1;
+------------------------------------------------------------------------+
| Explain String                                                         |
+------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0 |
```

```
| WARNING: The following tables are missing relevant table and/or column |
|   statistics.                                                          |
| explain_plan.t1                                                        |
|                                                                        |
| 01:EXCHANGE [PARTITION=UNPARTITIONED]                                  |
| |                                                                      |
| 00:SCAN HDFS [explain_plan.t1]                                         |
|    partitions=1/1 size=0B                                              |
+------------------------------------------------------------------------+
[localhost:21000] > set explain_level=2;
[localhost:21000] > explain select * from t1;
+------------------------------------------------------------------------+
| Explain String                                                         |
+------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0 |
| WARNING: The following tables are missing relevant table and/or column |
|   statistics.                                                          |
| explain_plan.t1                                                        |
|                                                                        |
| 01:EXCHANGE [PARTITION=UNPARTITIONED]                                  |
| |  hosts=0 per-host-mem=unavailable                                    |
| |  tuple-ids=0 row-size=19B cardinality=unavailable                    |
| |                                                                      |
| 00:SCAN HDFS [explain_plan.t1, PARTITION=RANDOM]                       |
|    partitions=1/1 size=0B                                              |
|    table stats: unavailable                                            |
|    column stats: unavailable                                           |
|    hosts=0 per-host-mem=0B                                             |
|    tuple-ids=0 row-size=19B cardinality=unavailable                    |
+------------------------------------------------------------------------+
[localhost:21000] > set explain_level=3;
[localhost:21000] > explain select * from t1;
+------------------------------------------------------------------------+
| Explain String                                                         |
+------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0 |
| WARNING: The following tables are missing relevant table and/or column |
|   statistics.                                                          |
| explain_plan.t1                                                        |
|                                                                        |
| F01:PLAN FRAGMENT [PARTITION=UNPARTITIONED]                            |
|   01:EXCHANGE [PARTITION=UNPARTITIONED]                                |
|      hosts=0 per-host-mem=unavailable                                  |
|      tuple-ids=0 row-size=19B cardinality=unavailable                  |
|                                                                        |
| F00:PLAN FRAGMENT [PARTITION=RANDOM]                                   |
|   DATASTREAM SINK [FRAGMENT=F01, EXCHANGE=01, PARTITION=UNPARTITIONED] |
|   00:SCAN HDFS [explain_plan.t1, PARTITION=RANDOM]                     |
|      partitions=1/1 size=0B                                            |
|      table stats: unavailable                                          |
|      column stats: unavailable                                         |
|      hosts=0 per-host-mem=0B                                           |
|      tuple-ids=0 row-size=19B cardinality=unavailable                  |
+------------------------------------------------------------------------+
```

As the warning message demonstrates, most of the information needed for Impala to do efficient query planning, and for you to understand the performance characteristics of the query, requires running the `COMPUTE STATS` statement for the table:

```
[localhost:21000] > compute stats t1;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+-----------------------------------------+
[localhost:21000] > explain select * from t1;
+------------------------------------------------------------------------+
| Explain String                                                         |
+------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=-9223372036854775808B VCores=0 |
|                                                                        |
| F01:PLAN FRAGMENT [PARTITION=UNPARTITIONED]                            |
```

```
|   01:EXCHANGE [PARTITION=UNPARTITIONED]                                      |
|      hosts=0 per-host-mem=unavailable                                        |
|      tuple-ids=0 row-size=20B cardinality=0                                  |
|                                                                              |
| F00:PLAN FRAGMENT [PARTITION=RANDOM]                                         |
|   DATASTREAM SINK [FRAGMENT=F01, EXCHANGE=01, PARTITION=UNPARTITIONED]       |
|   00:SCAN HDFS [explain_plan.t1, PARTITION=RANDOM]                           |
|      partitions=1/1 size=0B                                                  |
|      table stats: 0 rows total                                               |
|      column stats: all                                                       |
|      hosts=0 per-host-mem=0B                                                 |
|      tuple-ids=0 row-size=20B cardinality=0                                  |
+------------------------------------------------------------------------------+
```

Joins and other complicated, multi-part queries are the ones where you most commonly need to examine the EXPLAIN output and customize the amount of detail in the output. This example shows the default EXPLAIN output for a three-way join query, then the equivalent output with a [SHUFFLE] hint to change the join mechanism between the first two tables from a broadcast join to a shuffle join.

```
[localhost:21000] > set explain_level=1;
[localhost:21000] > explain select one.*, two.*, three.* from t1 one, t1 two, t1 three
 where one.x = two.x and two.x = three.x;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=4.00GB VCores=3  |
|                                                          |
| 07:EXCHANGE [PARTITION=UNPARTITIONED]                    |
| |                                                        |
| 04:HASH JOIN [INNER JOIN, BROADCAST]                     |
| |  hash predicates: two.x = three.x                      |
| |                                                        |
| |--06:EXCHANGE [BROADCAST]                               |
| |  |                                                     |
| |  02:SCAN HDFS [explain_plan.t1 three]                  |
| |     partitions=1/1 size=0B                             |
| |                                                        |
| 03:HASH JOIN [INNER JOIN, BROADCAST]                     |
| |  hash predicates: one.x = two.x                        |
| |                                                        |
| |--05:EXCHANGE [BROADCAST]                               |
| |  |                                                     |
| |  01:SCAN HDFS [explain_plan.t1 two]                    |
| |     partitions=1/1 size=0B                             |
| |                                                        |
| 00:SCAN HDFS [explain_plan.t1 one]                       |
|    partitions=1/1 size=0B                                |
+----------------------------------------------------------+
[localhost:21000] > explain select one.*, two.*, three.*
                  > from t1 one join [shuffle] t1 two join t1 three
                  > where one.x = two.x and two.x = three.x;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=4.00GB VCores=3  |
|                                                          |
| 08:EXCHANGE [PARTITION=UNPARTITIONED]                    |
| |                                                        |
| 04:HASH JOIN [INNER JOIN, BROADCAST]                     |
| |  hash predicates: two.x = three.x                      |
| |                                                        |
| |--07:EXCHANGE [BROADCAST]                               |
| |  |                                                     |
| |  02:SCAN HDFS [explain_plan.t1 three]                  |
| |     partitions=1/1 size=0B                             |
| |                                                        |
| 03:HASH JOIN [INNER JOIN, PARTITIONED]                   |
| |  hash predicates: one.x = two.x                        |
| |                                                        |
| |--06:EXCHANGE [PARTITION=HASH(two.x)]                   |
| |  |                                                     |
```

```
| |  01:SCAN HDFS [explain_plan.t1 two]                  |
| |     partitions=1/1 size=0B                           |
| |                                                      |
| 05:EXCHANGE [PARTITION=HASH(one.x)]                    |
| |                                                      |
| 00:SCAN HDFS [explain_plan.t1 one]                     |
|    partitions=1/1 size=0B                              |
+--------------------------------------------------------+
```

For a join involving many different tables, the default EXPLAIN output might stretch over several pages, and the only details you care about might be the join order and the mechanism (broadcast or shuffle) for joining each pair of tables. In that case, you might set EXPLAIN_LEVEL to its lowest value of 0, to focus on just the join order and join mechanism for each stage. The following example shows how the rows from the first and second joined tables are hashed and divided among the nodes of the cluster for further filtering; then the entire contents of the third table are broadcast to all nodes for the final stage of join processing.

```
[localhost:21000] > set explain_level=0;
[localhost:21000] > explain select one.*, two.*, three.*
                  > from t1 one join [shuffle] t1 two join t1 three
                  > where one.x = two.x and two.x = three.x;
+--------------------------------------------------------+
| Explain String                                         |
+--------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=4.00GB VCores=3 |
|                                                        |
| 08:EXCHANGE [PARTITION=UNPARTITIONED]                  |
| 04:HASH JOIN [INNER JOIN, BROADCAST]                   |
| |--07:EXCHANGE [BROADCAST]                             |
| |  02:SCAN HDFS [explain_plan.t1 three]                |
| 03:HASH JOIN [INNER JOIN, PARTITIONED]                 |
| |--06:EXCHANGE [PARTITION=HASH(two.x)]                 |
| |  01:SCAN HDFS [explain_plan.t1 two]                  |
| 05:EXCHANGE [PARTITION=HASH(one.x)]                    |
| 00:SCAN HDFS [explain_plan.t1 one]                     |
+--------------------------------------------------------+
```

HBASE_CACHE_BLOCKS Query Option

Setting this option is equivalent to calling the `setCacheBlocks` method of the class org.apache.hadoop.hbase.client.Scan, in an HBase Java application. Helps to control the memory pressure on the HBase RegionServer, in conjunction with the HBASE_CACHING query option.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of SET statement)

Related information:

Using Impala to Query HBase Tables on page 684, HBASE_CACHING Query Option on page 335

HBASE_CACHING Query Option

Setting this option is equivalent to calling the `setCaching` method of the class org.apache.hadoop.hbase.client.Scan, in an HBase Java application. Helps to control the memory pressure on the HBase RegionServer, in conjunction with the HBASE_CACHE_BLOCKS query option.

Type: BOOLEAN

Default: 0

Related information:

Using Impala to Query HBase Tables on page 684, HBASE_CACHE_BLOCKS Query Option on page 335

IDLE_SESSION_TIMEOUT Query Option (CDH 5.15 / Impala 2.12 or higher only)

The IDLE_SESSION_TIMEOUT query option sets the time in seconds after which an idle session is cancelled. A session is idle when no activity is occurring for any of the queries in that session, and the session has not started any new

queries. Once a session is expired, you cannot issue any new query requests to it. The session remains open, but the only operation you can perform is to close it.

The IDLE_SESSION_TIMEOUT query option overrides the --idle_session_timeout startup option. See Setting Timeout Periods for Daemons, Queries, and Sessions on page 70 for the --idle_session_timeout startup option.

The IDLE_SESSION_TIMEOUT query option allows JDBC/ODBC connections to set the session timeout as a query option with the SET statement.

Syntax:

```
SET IDLE_SESSION_TIMEOUT=seconds;
```

Type: numeric

Default: 0

- If --idle_session_timeout is not set, the session never expires.
- If --idle_session_timeout is set, use that timeout value.

Added in: CDH 5.15 / Impala 2.12

Related information:

Setting Timeout Periods for Daemons, Queries, and Sessions on page 70

KUDU_READ_MODE Query Option (CDH 6.1 or higher only)

The KUDU_READ_MODE query option allows you to set a desired consistency level for scans of Kudu tables.

Type: String

Default: "DEFAULT"

Added in: CDH 6.1

Usage notes:

The following values are supported for the query option:

- "DEFAULT": The value of the startup flag, --kudu_read_mode, is used.
- "READ_LATEST": Commonly known as the Read Committed isolation mode, in this mode, Kudu provides no consistency guarantees for this mode, except that all returned rows were committed at some point.
- "READ_AT_SNAPSHOT": Kudu will take a snapshot of the current state of the data and perform the scan over the snapshot, possibly after briefly waiting for ongoing writes to complete. This provides "Read Your Writes" consistency within a single Impala session, except in the case of a Kudu leader change. See the Kudu documentation for more details.

LIVE_PROGRESS Query Option (CDH 5.5 or higher only)

When the LIVE_PROGRESS query option is set to TRUE, Impala displays an interactive progress bar showing roughly what percentage of processing has been completed for queries submitted through the impala-shell command. When the query finishes, the progress bar is erased from the impala-shell console output.

Starting in CDH 6.1, the summary output also includes the queuing status consisting of whether the query was queued and what was the latest queuing reason.

Type: Boolean

Default: FALSE (0)

Command-line equivalent:

You can enable this query option within impala-shell by starting the shell with the --live_progress command-line option. You can still turn this setting off and on again within the shell through the SET command.

Usage notes:

The output from this query option is printed to standard error. The output is only displayed in interactive mode, that is, not when the `-q` or `-f` options are used.

For a more detailed way of tracking the progress of an interactive query through all phases of processing, see LIVE_SUMMARY Query Option (CDH 5.5 or higher only) on page 337.

Restrictions:

Because the percentage complete figure is calculated using the number of issued and completed "scan ranges", which occur while reading the table data, the progress bar might reach 100% before the query is entirely finished. For example, the query might do work to perform aggregations after all the table data has been read. If many of your queries fall into this category, consider using the `LIVE_SUMMARY` option instead for more granular progress reporting.

The `LIVE_PROGRESS` and `LIVE_SUMMARY` query options currently do not produce any output during `COMPUTE STATS` operations.

Because the `LIVE_PROGRESS` and `LIVE_SUMMARY` query options are available only within the `impala-shell` interpreter:

- You cannot change these query options through the SQL `SET` statement using the JDBC or ODBC interfaces. The `SET` command in `impala-shell` recognizes these names as shell-only options.
- Be careful when using `impala-shell` on a pre-CDH 5.5 system to connect to Impala running on a CDH 5.5 or higher system. The older `impala-shell` does not recognize these query option names. Upgrade `impala-shell` on the systems where you intend to use these query options.
- Likewise, the `impala-shell` command relies on some information only available in CDH 5.5 / Impala 2.3 and higher to prepare live progress reports and query summaries. The `LIVE_PROGRESS` and `LIVE_SUMMARY` query options have no effect when `impala-shell` connects to a cluster running an older version of Impala.

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
[localhost:21000] > set live_progress=true;
LIVE_PROGRESS set to true
[localhost:21000] > select count(*) from customer;
+----------+
| count(*) |
+----------+
| 150000   |
+----------+
[localhost:21000] > select count(*) from customer t1 cross join customer t2;
[####################################                                  ] 50%
[######################################################################] 100%
```

To see how the `LIVE_PROGRESS` and `LIVE_SUMMARY` query options work in real time, see this animated demo.

LIVE_SUMMARY Query Option (CDH 5.5 or higher only)

When the `LIVE_SUMMARY` query option is set to `TRUE`, Impala displays the same output as the `SUMMARY` command for queries submitted through the `impala-shell` command, with the measurements updated in real time as the query progresses. When the query finishes, the final `SUMMARY` output remains visible in the `impala-shell` console output.

Starting in CDH 6.1, the summary output also includes the queuing status consisting of whether the query was queued and what was the latest queuing reason.

Type: `Boolean`

Default: `FALSE (0)`

Command-line equivalent:

You can enable this query option within `impala-shell` by starting the shell with the `--live_summary` command-line option. You can still turn this setting off and on again within the shell through the SET command.

Usage notes:

The live summary output can be useful for evaluating long-running queries, to evaluate which phase of execution takes up the most time, or if some hosts take much longer than others for certain operations, dragging overall performance down. By making the information available in real time, this feature lets you decide what action to take even before you cancel a query that is taking much longer than normal.

For example, you might see the HDFS scan phase taking a long time, and therefore revisit performance-related aspects of your schema design such as constructing a partitioned table, switching to the Parquet file format, running the `COMPUTE STATS` statement for the table, and so on. Or you might see a wide variation between the average and maximum times for all hosts to perform some phase of the query, and therefore investigate if one particular host needed more memory or was experiencing a network problem.

The output from this query option is printed to standard error. The output is only displayed in interactive mode, that is, not when the `-q` or `-f` options are used.

For a simple and concise way of tracking the progress of an interactive query, see LIVE_PROGRESS Query Option (CDH 5.5 or higher only) on page 336.

Restrictions:

The `LIVE_PROGRESS` and `LIVE_SUMMARY` query options currently do not produce any output during `COMPUTE STATS` operations.

Because the `LIVE_PROGRESS` and `LIVE_SUMMARY` query options are available only within the `impala-shell` interpreter:

- You cannot change these query options through the SQL SET statement using the JDBC or ODBC interfaces. The SET command in `impala-shell` recognizes these names as shell-only options.
- Be careful when using `impala-shell` on a pre-CDH 5.5 system to connect to Impala running on a CDH 5.5 or higher system. The older `impala-shell` does not recognize these query option names. Upgrade `impala-shell` on the systems where you intend to use these query options.
- Likewise, the `impala-shell` command relies on some information only available in CDH 5.5 / Impala 2.3 and higher to prepare live progress reports and query summaries. The `LIVE_PROGRESS` and `LIVE_SUMMARY` query options have no effect when `impala-shell` connects to a cluster running an older version of Impala.

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following example shows a series of `LIVE_SUMMARY` reports that are displayed during the course of a query, showing how the numbers increase to show the progress of different phases of the distributed query. When you do the same in `impala-shell`, only a single report is displayed at any one time, with each update overwriting the previous numbers.

```
[localhost:21000] > set live_summary=true;
LIVE_SUMMARY set to true
[localhost:21000] > select count(*) from customer t1 cross join customer t2;
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
| Operator            | #Hosts | Avg Time | Max Time | #Rows   | Est. #Rows | Peak Mem | Est. Peak Mem | Detail                |
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
| 06:AGGREGATE        | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | -1 B          | FINALIZE              |
| 05:EXCHANGE         | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | -1 B          | UNPARTITIONED         |
| 03:AGGREGATE        | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | 10.00 MB      |                       |
| 02:NESTED LOOP JOIN | 0      | 0ns      | 0ns      | 0       | 22.50B     | 0 B      | 0 B           | CROSS JOIN, BROADCAST |
| |--04:EXCHANGE      | 0      | 0ns      | 0ns      | 0       | 150.00K    | 0 B      | 0 B           | BROADCAST             |
| |  01:SCAN HDFS     | 1      | 503.57ms | 503.57ms | 150.00K | 150.00K    | 24.09 MB | 64.00 MB      | tpch.customer t2      |
| 00:SCAN HDFS        | 0      | 0ns      | 0ns      | 0       | 150.00K    | 0 B      | 64.00 MB      | tpch.customer t1      |
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+

+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
| Operator            | #Hosts | Avg Time | Max Time | #Rows   | Est. #Rows | Peak Mem | Est. Peak Mem | Detail                |
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
| 06:AGGREGATE        | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | -1 B          | FINALIZE              |
| 05:EXCHANGE         | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | -1 B          | UNPARTITIONED         |
| 03:AGGREGATE        | 1      | 0ns      | 0ns      | 0       | 1          | 20.00 KB | 10.00 MB      |                       |
| 02:NESTED LOOP JOIN | 1      | 17.62s   | 17.62s   | 81.14M  | 22.50B     | 3.23 MB  | 0 B           | CROSS JOIN, BROADCAST |
| |--04:EXCHANGE      | 1      | 26.29ms  | 26.29ms  | 150.00K | 150.00K    | 0 B      | 0 B           | BROADCAST             |
| |  01:SCAN HDFS     | 1      | 503.57ms | 503.57ms | 150.00K | 150.00K    | 24.09 MB | 64.00 MB      | tpch.customer t2      |
```

```
| 00:SCAN HDFS        | 1      | 247.53ms | 247.53ms | 1.02K   | 150.00K    | 24.39 MB | 64.00 MB      | tpch.customer t1      |
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+

+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
| Operator            | #Hosts | Avg Time | Max Time | #Rows   | Est. #Rows | Peak Mem | Est. Peak Mem | Detail                |
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
| 06:AGGREGATE        | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | -1 B          | FINALIZE              |
| 05:EXCHANGE         | 0      | 0ns      | 0ns      | 0       | 1          | 0 B      | -1 B          | UNPARTITIONED         |
| 03:AGGREGATE        | 1      | 0ns      | 0ns      | 0       | 1          | 20.00 KB | 10.00 MB      |                       |
| 02:NESTED LOOP JOIN | 1      | 61.85s   | 61.85s   | 283.43M | 22.50B     | 3.23 MB  | 0 B           | CROSS JOIN, BROADCAST |
| |--04:EXCHANGE      | 1      | 26.29ms  | 26.29ms  | 150.00K | 150.00K    | 0 B      | 0 B           | BROADCAST             |
| |  01:SCAN HDFS     | 1      | 503.57ms | 503.57ms | 150.00K | 150.00K    | 24.09 MB | 64.00 MB      | tpch.customer t2      |
| 00:SCAN HDFS        | 1      | 247.59ms | 247.59ms | 2.05K   | 150.00K    | 24.39 MB | 64.00 MB      | tpch.customer t1      |
+---------------------+--------+----------+----------+---------+------------+----------+---------------+-----------------------+
```

To see how the LIVE_PROGRESS and LIVE_SUMMARY query options work in real time, see this animated demo.

MAX_ERRORS Query Option

Maximum number of non-fatal errors for any particular query that are recorded in the Impala log file. For example, if a billion-row table had a non-fatal data error in every row, you could diagnose the problem without all billion errors being logged. Unspecified or 0 indicates the built-in default value of 1000.

This option only controls how many errors are reported. To specify whether Impala continues or halts when it encounters such errors, use the ABORT_ON_ERROR option.

Type: numeric

Default: 0 (meaning 1000 errors)

Related information:

ABORT_ON_ERROR Query Option on page 323, Using Impala Logging on page 709

MAX_MEM_ESTIMATE_FOR_ADMISSION Query Option

Use the MAX_MEM_ESTIMATE_FOR_ADMISSION query option to set an upper limit on the memory estimates of a query as a workaround for over-estimates precluding a query from being admitted.

The query option takes effect when all of the below conditions are met:

- Memory-based admission control is enabled for the pool.
- The MEM_LIMIT query option is not set at the query, session, resource pool, or global level.

When the above conditions are met, MIN(MAX_MEM_ESTIMATE_FOR_ADMISSION, mem_estimate) is used for admission control.

Setting the MEM_LIMIT query option is usually a better option. Use the MAX_MEM_ESTIMATE_FOR_ADMISSION query option when it is not feasible to set MEM_LIMIT for each individual query.

Type: integer

Added in: CDH 6.1

MAX_NUM_RUNTIME_FILTERS Query Option (CDH 5.7 or higher only)

The MAX_NUM_RUNTIME_FILTERS query option sets an upper limit on the number of runtime filters that can be produced for each query.

Type: integer

Default: 10

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

Each runtime filter imposes some memory overhead on the query. Depending on the setting of the RUNTIME_BLOOM_FILTER_SIZE query option, each filter might consume between 1 and 16 megabytes per plan fragment. There are typically 5 or fewer filters per plan fragment.

Impala evaluates the effectiveness of each filter, and keeps the ones that eliminate the largest number of partitions or rows. Therefore, this setting can protect against potential problems due to excessive memory overhead for filter production, while still allowing a high level of optimization for suitable queries.

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu considerations:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

Related information:

Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588, RUNTIME_BLOOM_FILTER_SIZE Query Option (CDH 5.7 or higher only) on page 356, RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358

MAX_ROW_SIZE Query Option

Ensures that Impala can process rows of at least the specified size. (Larger rows might be successfully processed, but that is not guaranteed.) Applies when constructing intermediate or final rows in the result set. This setting prevents out-of-control memory use when accessing columns containing huge strings.

Type: integer

Default:

524288 (512 KB)

Units: A numeric argument represents a size in bytes; you can also use a suffix of `m` or `mb` for megabytes, or `g` or `gb` for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Added in: CDH 5.13.0 / Impala 2.10.0

Usage notes:

If a query fails because it involves rows with long strings and/or many columns, causing the total row size to exceed `MAX_ROW_SIZE` bytes, increase the `MAX_ROW_SIZE` setting to accommodate the total bytes stored in the largest row. Examine the error messages for any failed queries to see the size of the row that caused the problem.

Impala attempts to handle rows that exceed the `MAX_ROW_SIZE` value where practical, so in many cases, queries succeed despite having rows that are larger than this setting.

Specifying a value that is substantially higher than actually needed can cause Impala to reserve more memory than is necessary to execute the query.

In a Hadoop cluster with highly concurrent workloads and queries that process high volumes of data, traditional SQL tuning advice about minimizing wasted memory is worth remembering. For example, if a table has `STRING` columns where a single value might be multiple megabytes, make sure that the `SELECT` lists in queries only refer to columns that are actually needed in the result set, instead of using the `SELECT *` shorthand.

If you are upgrading Impala to CDH 5.13 / Impala 2.10 or higher from CDH 5.12 / Impala 2.9 or lower, follow the instructions in Handling Large Rows During Upgrade to CDH 5.13 / Impala 2.10 or Higher on page 38 to check if your queries are affected by these changes and to modify your configuration settings if so. This advice is especially important for any users who have increased the `--read_size` configuration setting from its default value of 8 MB.

Examples:

The following examples show the kinds of situations where it is necessary to adjust the `MAX_ROW_SIZE` setting. First, we create a table containing some very long values in `STRING` columns:

```
create table big_strings (s1 string, s2 string, s3 string) stored as parquet;

-- Turn off compression to more easily reason about data volume by doing SHOW TABLE
STATS.
```

```
-- Does not actually affect query success or failure, because MAX_ROW_SIZE applies when
-- column values are materialized in memory.
set compression_codec=none;
set;
...
  MAX_ROW_SIZE: [524288]
...

-- A very small row.
insert into big_strings values ('one', 'two', 'three');
-- A row right around the default MAX_ROW_SIZE limit: a 500 KiB string and a 30 KiB
string.
insert into big_strings values (repeat('12345',100000), 'short', repeat('123',10000));
-- A row that is too big if the query has to materialize both S1 and S3.
insert into big_strings values (repeat('12345',100000), 'short', repeat('12345',100000));
```

With the default MAX_ROW_SIZE setting, different queries succeed or fail based on which column values have to be materialized during query processing:

```
-- All the S1 values can be materialized within the 512 KB MAX_ROW_SIZE buffer.
select count(distinct s1) from big_strings;
+--------------------+
| count(distinct s1) |
+--------------------+
| 2                  |
+--------------------+

-- A row where even the S1 value is too large to materialize within MAX_ROW_SIZE.
insert into big_strings values (repeat('12345',1000000), 'short',
repeat('12345',1000000));

-- The 5 MiB string is too large to materialize. The message explains the size of the
result
-- set row the query is attempting to materialize.
select count(distinct(s1)) from big_strings;
WARNINGS: Row of size 4.77 MB could not be materialized in plan node with id 1.
  Increase the max_row_size query option (currently 512.00 KB) to process larger rows.

-- If more columns are involved, the result set row being materialized is bigger.
select count(distinct s1, s2, s3) from big_strings;
WARNINGS: Row of size 9.54 MB could not be materialized in plan node with id 1.
  Increase the max_row_size query option (currently 512.00 KB) to process larger rows.

-- Column S2, containing only short strings, can still be examined.
select count(distinct(s2)) from big_strings;
+----------------------+
| count(distinct (s2)) |
+----------------------+
| 2                    |
+----------------------+

-- Queries that do not materialize the big column values are OK.
select count(*) from big_strings;
+----------+
| count(*) |
+----------+
| 4        |
+----------+
```

The following examples show how adjusting MAX_ROW_SIZE upward allows queries involving the long string columns to succeed:

```
-- Boosting MAX_ROW_SIZE moderately allows all S1 values to be materialized.
set max_row_size=7mb;

select count(distinct s1) from big_strings;
```

```
+--------------------+
| count(distinct s1) |
+--------------------+
| 3                  |
+--------------------+

-- But the combination of S1 + S3 strings is still too large.
select count(distinct s1, s2, s3) from big_strings;
WARNINGS: Row of size 9.54 MB could not be materialized in plan node with id 1. Increase
 the max_row_size query option (currently 7.00 MB) to process larger rows.

-- Boosting MAX_ROW_SIZE to larger than the largest row in the table allows
-- all queries to complete successfully.
set max_row_size=12mb;

select count(distinct s1, s2, s3) from big_strings;
+----------------------------+
| count(distinct s1, s2, s3) |
+----------------------------+
| 4                          |
+----------------------------+
```

The following examples show how to reason about appropriate values for MAX_ROW_SIZE, based on the characteristics of the columns containing the long values:

```
-- With a large MAX_ROW_SIZE in place, we can examine the columns to
-- understand the practical lower limit for MAX_ROW_SIZE based on the
-- table structure and column values.
select max(length(s1) + length(s2) + length(s3)) / 1e6 as megabytes from big_strings;
+-----------+
| megabytes |
+-----------+
| 10.000005 |
+-----------+

-- We can also examine the 'Max Size' for each column after computing stats.
compute stats big_strings;
show column stats big_strings;
+--------+--------+------------------+--------+----------+-----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size  |
+--------+--------+------------------+--------+----------+-----------+
| s1     | STRING | 2                | -1     | 5000000  | 2500002.5 |
| s2     | STRING | 2                | -1     | 10       | 7.5       |
| s3     | STRING | 2                | -1     | 5000000  | 2500005   |
+--------+--------+------------------+--------+----------+-----------+
```

Related information:

BUFFER_POOL_LIMIT Query Option on page 324, DEFAULT_SPILLABLE_BUFFER_SIZE Query Option on page 327, MIN_SPILLABLE_BUFFER_SIZE Query Option on page 345, Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher) on page 607

MAX_SCAN_RANGE_LENGTH Query Option

Maximum length of the scan range. Interacts with the number of HDFS blocks in the table to determine how many CPU cores across the cluster are involved with the processing for a query. (Each core processes one scan range.)

Lowering the value can sometimes increase parallelism if you have unused CPU capacity, but a too-small value can limit query performance because each scan range involves extra overhead.

Only applicable to HDFS tables. Has no effect on Parquet tables. Unspecified or 0 indicates backend default, which is the same as the HDFS block size for each table.

Although the scan range can be arbitrarily long, Impala internally uses an 8 MB read buffer so that it can query tables with huge block sizes without allocating equivalent blocks of memory.

Type: numeric

In CDH 5.9 / Impala 2.7 and higher, the argument value can include unit specifiers, such as `100m` or `100mb`. In previous versions, Impala interpreted such formatted values as 0, leading to query failures.

Default: 0

MEM_LIMIT Query Option

The `MEM_LIMIT` query option defines the maximum amount of memory a query can allocate on each node. The total memory that can be used by a query is the `MEM_LIMIT` times the number of nodes.

There are two levels of memory limit for Impala. The `--mem_limit` startup option sets an overall limit for the `impalad` process (which handles multiple queries concurrently). That process memory limit can be expressed either as a percentage of RAM available to the process such as `-mem_limit=70%` or as a fixed amount of memory, such as `--mem_limit=100gb`. The memory available to the process is based on the host's physical memory and, since CDH 6.1 / Impala 3.2, memory limits from Linux Control Groups. For example, if an `impalad` process is running in a Docker container on a host with 100 GB of memory, the memory available is 100 GB or the Docker container's memory limit, whichever is less.

The `MEM_LIMIT` query option, which you set through `impala-shell` or the `SET` statement in a JDBC or ODBC application, applies to each individual query. The `MEM_LIMIT` query option is usually expressed as a fixed size such as `10gb`, and must always be less than the `impalad` memory limit.

If query processing exceeds the specified memory limit on any node, either the per-query limit or the `impalad` limit, Impala cancels the query automatically. Memory limits are checked periodically during query processing, so the actual memory in use might briefly exceed the limit without the query being cancelled.

Type: numeric

Units: A numeric argument represents memory size in bytes; you can also use a suffix of `m` or `mb` for megabytes, or more commonly `g` or `gb` for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Default: 0 (unlimited)

Usage notes:

The `MEM_LIMIT` setting is primarily useful in a high-concurrency setting, or on a cluster with a workload shared between Impala and other data processing components. You can prevent any query from accidentally using much more memory than expected, which could negatively impact other Impala queries.

Use the output of the `SUMMARY` command in `impala-shell` to get a report of memory used for each phase of your most heavyweight queries on each node, and then set a `MEM_LIMIT` somewhat higher than that. See Using the SUMMARY Report for Performance Tuning on page 603 for usage information about the `SUMMARY` command.

Examples:

The following examples show how to set the `MEM_LIMIT` query option using a fixed number of bytes, or suffixes representing gigabytes or megabytes.

```
[localhost:21000] > set mem_limit=3000000000;
MEM_LIMIT set to 3000000000
[localhost:21000] > select 5;
Query: select 5
+---+
| 5 |
+---+
| 5 |
+---+

[localhost:21000] > set mem_limit=3g;
MEM_LIMIT set to 3g
[localhost:21000] > select 5;
Query: select 5
+---+
| 5 |
+---+
```

```
| 5 |
+---+

[localhost:21000] > set mem_limit=3gb;
MEM_LIMIT set to 3gb
[localhost:21000] > select 5;
+---+
| 5 |
+---+
| 5 |
+---+

[localhost:21000] > set mem_limit=3m;
MEM_LIMIT set to 3m
[localhost:21000] > select 5;
+---+
| 5 |
+---+
| 5 |
+---+
[localhost:21000] > set mem_limit=3mb;
MEM_LIMIT set to 3mb
[localhost:21000] > select 5;
+---+
| 5 |
+---+
```

The following examples show how unrecognized `MEM_LIMIT` values lead to errors for subsequent queries.

```
[localhost:21000] > set mem_limit=3tb;
MEM_LIMIT set to 3tb
[localhost:21000] > select 5;
ERROR: Failed to parse query memory limit from '3tb'.

[localhost:21000] > set mem_limit=xyz;
MEM_LIMIT set to xyz
[localhost:21000] > select 5;
Query: select 5
ERROR: Failed to parse query memory limit from 'xyz'.
```

The following examples shows the automatic query cancellation when the `MEM_LIMIT` value is exceeded on any host involved in the Impala query. First it runs a successful query and checks the largest amount of memory used on any node for any stage of the query. Then it sets an artificially low `MEM_LIMIT` setting so that the same query cannot run.

```
[localhost:21000] > select count(*) from customer;
Query: select count(*) from customer
+----------+
| count(*) |
+----------+
| 150000   |
+----------+

[localhost:21000] > select count(distinct c_name) from customer;
Query: select count(distinct c_name) from customer
+------------------------+
| count(distinct c_name) |
+------------------------+
| 150000                 |
+------------------------+

[localhost:21000] > summary;
+--------------+--------+----------+----------+---------+------------+----------+---------------+---------------+
| Operator     | #Hosts | Avg Time | Max Time | #Rows   | Est. #Rows | Peak Mem | Est. Peak Mem | Detail        |
+--------------+--------+----------+----------+---------+------------+----------+---------------+---------------+
| 06:AGGREGATE | 1      | 230.00ms | 230.00ms | 1       | 1          | 16.00 KB | -1 B          | FINALIZE      |
| 05:EXCHANGE  | 1      | 43.44us  | 43.44us  | 1       | 1          | 0 B      | -1 B          | UNPARTITIONED |
| 02:AGGREGATE | 1      | 227.14ms | 227.14ms | 1       | 1          | 12.00 KB | 10.00 MB      |               |
| 04:AGGREGATE | 1      | 126.27ms | 126.27ms | 150.00K | 150.00K    | 15.17 MB | 10.00 MB      |               |
| 03:EXCHANGE  | 1      | 44.07ms  | 44.07ms  | 150.00K | 150.00K    | 0 B      | 0 B           | HASH(c_name)  |
| 01:AGGREGATE | 1      | 361.94ms | 361.94ms | 150.00K | 150.00K    | 23.04 MB | 10.00 MB      |               |
| 00:SCAN HDFS | 1      | 43.64ms  | 43.64ms  | 150.00K | 150.00K    | 24.19 MB | 64.00 MB      | tpch.customer |
+--------------+--------+----------+----------+---------+------------+----------+---------------+---------------+

[localhost:21000] > set mem_limit=15mb;
MEM_LIMIT set to 15mb
[localhost:21000] > select count(distinct c_name) from customer;
Query: select count(distinct c_name) from customer
ERROR:
Memory limit exceeded
Query did not have enough memory to get the minimum required buffers in the block manager.
```

MIN_SPILLABLE_BUFFER_SIZE Query Option

Specifies the minimum size for a memory buffer used when the spill-to-disk mechanism is activated, for example for queries against a large table with no statistics, or large join operations.

Type: integer

Default:

65536 (64 KB)

Units: A numeric argument represents a size in bytes; you can also use a suffix of `m` or `mb` for megabytes, or `g` or `gb` for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Added in: CDH 5.13.0 / Impala 2.10.0

Usage notes:

This query option sets a lower bound on the size of the internal buffer size that can be used during spill-to-disk operations. The actual size of the buffer is chosen by the query planner.

If overall query performance is limited by the time needed for spilling, consider increasing the `MIN_SPILLABLE_BUFFER_SIZE` setting. Larger buffer sizes result in Impala issuing larger I/O requests to storage devices, which might result in higher throughput, particularly on rotational disks.

The tradeoff with a large value for this setting is increased memory usage during spill-to-disk operations. Reducing this value may reduce memory consumption.

To determine if the value for this setting is having an effect by capping the spillable buffer size, you can see the buffer size chosen by the query planner for a particular query. `EXPLAIN` the query while the setting `EXPLAIN_LEVEL=2` is in effect.

Examples:

```
set min_spillable_buffer_size=128KB;
```

Related information:

BUFFER_POOL_LIMIT Query Option on page 324, DEFAULT_SPILLABLE_BUFFER_SIZE Query Option on page 327, MAX_ROW_SIZE Query Option on page 340, Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher) on page 607

MT_DOP Query Option

Sets the degree of parallelism used for certain operations that can benefit from multithreaded execution. You can specify values higher than zero to find the ideal balance of response time, memory usage, and CPU usage during statement processing.



Note:

The Impala execution engine is being revamped incrementally to add additional parallelism within a single host for certain statements and kinds of operations. The setting `MT_DOP=0` uses the “old” code path with limited intra-node parallelism.

Currently, the operations affected by the `MT_DOP` query option are:

- `COMPUTE [INCREMENTAL] STATS`. Impala automatically sets `MT_DOP=4` for `COMPUTE STATS` and `COMPUTE INCREMENTAL STATS` statements on Parquet tables.
- Queries with execution plans containing only scan and aggregation operators. Other queries produce an error if `MT_DOP` is set to a non-zero value. Therefore, this query option is typically only set for the duration of specific long-running, CPU-intensive queries.

Type: integer

Default: `0`

Because `COMPUTE STATS` and `COMPUTE INCREMENTAL STATS` statements for Parquet tables benefit substantially from extra intra-node parallelism, Impala automatically sets `MT_DOP=4` when computing stats for Parquet tables.

Range: 0 to 64

Examples:



Note:

Any timing figures in the following examples are on a small, lightly loaded development cluster. Your mileage may vary. Speedups depend on many factors, including the number of rows, columns, and partitions within each table.

The following example shows how to run a `COMPUTE STATS` statement against a Parquet table with or without an explicit `MT_DOP` setting:

```
-- Explicitly setting MT_DOP to 0 selects the old code path.
set mt_dop = 0;
MT_DOP set to 0

-- The analysis for the billion rows is distributed among hosts,
-- but uses only a single core on each host.
compute stats billion_rows_parquet;
+------------------------------------------+
| summary                                  |
+------------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+------------------------------------------+

drop stats billion_rows_parquet;

-- Using 4 logical processors per host is faster.
set mt_dop = 4;
MT_DOP set to 4

compute stats billion_rows_parquet;
+------------------------------------------+
| summary                                  |
+------------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+------------------------------------------+

drop stats billion_rows_parquet;

-- Unsetting the option reverts back to its default.
-- Which for COMPUTE STATS and a Parquet table is 4,
-- so again it uses the fast path.
unset MT_DOP;
Unsetting option MT_DOP

compute stats billion_rows_parquet;
+------------------------------------------+
| summary                                  |
+------------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+------------------------------------------+
```

The following example shows the effects of setting `MT_DOP` for a query involving only scan and aggregation operations for a Parquet table:

```
set mt_dop = 0;
MT_DOP set to 0
```

```
-- COUNT(DISTINCT) for a unique column is CPU-intensive.
select count(distinct id) from billion_rows_parquet;
+--------------------+
| count(distinct id) |
+--------------------+
| 1000000000         |
+--------------------+
Fetched 1 row(s) in 67.20s

set mt_dop = 16;
MT_DOP set to 16

-- Introducing more intra-node parallelism for the aggregation
-- speeds things up, and potentially reduces memory overhead by
-- reducing the number of scanner threads.
select count(distinct id) from billion_rows_parquet;
+--------------------+
| count(distinct id) |
+--------------------+
| 1000000000         |
+--------------------+
Fetched 1 row(s) in 17.19s
```

The following example shows how queries that are not compatible with non-zero MT_DOP settings produce an error when MT_DOP is set:

```
set mt_dop=1;
MT_DOP set to 1

select * from a1 inner join a2
  on a1.id = a2.id limit 4;
ERROR: NotImplementedException: MT_DOP not supported for plans with
  base table joins or table sinks.
```

Related information:

COMPUTE STATS Statement on page 219, Impala Aggregate Functions on page 479

NUM_NODES Query Option

Limit the number of nodes that process a query, typically during debugging.

Type: numeric

Allowed values: Only accepts the values 0 (meaning all nodes) or 1 (meaning all work is done on the coordinator node).

Default: 0

Usage notes:

If you are diagnosing a problem that you suspect is due to a timing issue due to distributed query processing, you can set NUM_NODES=1 to verify if the problem still occurs when all the work is done on a single node.

You might set the NUM_NODES option to 1 briefly, during INSERT or CREATE TABLE AS SELECT statements. Normally, those statements produce one or more data files per data node. If the write operation involves small amounts of data, a Parquet table, and/or a partitioned table, the default behavior could produce many small files when intuitively you might expect only a single output file. SET NUM_NODES=1 turns off the “distributed” aspect of the write operation, making it more likely to produce only one or a few data files.



Warning:

Because this option results in increased resource utilization on a single host, it could cause problems due to contention with other Impala statements or high resource usage. Symptoms could include queries running slowly, exceeding the memory limit, or appearing to hang. Use it only in a single-user development/test environment; **do not** use it in a production environment or in a cluster with a high-concurrency or high-volume or performance-critical workload.

NUM_ROWS_PRODUCED_LIMIT Query Option

The `NUM_ROWS_PRODUCED_LIMIT` query option limits the number of rows produced by a query. A query is canceled when its execution produces more rows than the specified limit set by the `NUM_ROWS_PRODUCED_LIMIT` option.

This limit only applies when the results are returned to a client, for example, to a `SELECT` query, but not to an `INSERT` query.

The default value of `0` specifies that there is no limit on the number of rows produced.

Type: `INT`

Allowed values: 0 or positive numbers

Default: 0

NUM_SCANNER_THREADS Query Option

Maximum number of scanner threads (on each node) used for each query. By default, Impala uses as many cores as are available (one thread per core). You might lower this value if queries are using excessive resources on a busy cluster. Impala imposes a maximum value automatically, so a high value has no practical effect.

Type: numeric

Default: 0

OPTIMIZE_PARTITION_KEY_SCANS Query Option (CDH 5.7 or higher only)

Enables a fast code path for queries that apply simple aggregate functions to partition key columns: `MIN(`*`key_column`*`)`, `MAX(`*`key_column`*`)`, or `COUNT(DISTINCT` *`key_column`*`)`.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of `SET` statement)



Note: In CDH 5.7.0 / Impala 2.5.0, only the value 1 enables the option, and the value `true` is not recognized. This limitation is tracked by the issue IMPALA-3334, which shows the releases where the problem is fixed.

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

This optimization speeds up common “introspection” operations over partition key columns, for example determining the distinct values of partition keys.

This optimization does not apply to `SELECT` statements that reference columns that are not partition keys. It also only applies when all the partition key columns in the `SELECT` statement are referenced in one of the following contexts:

- Within a `MAX()` or `MAX()` aggregate function or as the argument of any aggregate function with the `DISTINCT` keyword applied.
- Within a `WHERE`, `GROUP BY` or `HAVING` clause.

This optimization is enabled by a query option because it skips some consistency checks and therefore can return slightly different partition values if partitions are in the process of being added, dropped, or loaded outside of Impala. Queries might exhibit different behavior depending on the setting of this option in the following cases:

- If files are removed from a partition using HDFS or other non-Impala operations, there is a period until the next REFRESH of the table where regular queries fail at run time because they detect the missing files. With this optimization enabled, queries that evaluate only the partition key column values (not the contents of the partition itself) succeed, and treat the partition as if it still exists.
- If a partition contains any data files, but the data files do not contain any rows, a regular query considers that the partition does not exist. With this optimization enabled, the partition is treated as if it exists.

  If the partition includes no files at all, this optimization does not change the query behavior: the partition is considered to not exist whether or not this optimization is enabled.

Examples:

The following example shows initial schema setup and the default behavior of queries that return just the partition key column for a table:

```
-- Make a partitioned table with 3 partitions.
create table t1 (s string) partitioned by (year int);
insert into t1 partition (year=2015) values ('last year');
insert into t1 partition (year=2016) values ('this year');
insert into t1 partition (year=2017) values ('next year');

-- Regardless of the option setting, this query must read the
-- data files to know how many rows to return for each year value.
explain select year from t1;
+--------------------------------------------------+
| Explain String                                   |
+--------------------------------------------------+
| Estimated Per-Host Requirements: Memory=0B VCores=0 |
|                                                  |
| F00:PLAN FRAGMENT [UNPARTITIONED]                |
|   00:SCAN HDFS [key_cols.t1]                     |
|      partitions=3/3 files=4 size=40B             |
|      table stats: 3 rows total                   |
|      column stats: all                           |
|      hosts=3 per-host-mem=unavailable            |
|      tuple-ids=0 row-size=4B cardinality=3       |
+--------------------------------------------------+

-- The aggregation operation means the query does not need to read
-- the data within each partition: the result set contains exactly 1 row
-- per partition, derived from the partition key column value.
-- By default, Impala still includes a 'scan' operation in the query.
explain select distinct year from t1;
+--------------------------------------------------------------------------------------+
| Explain String                                                                       |
+--------------------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=0B VCores=0                                  |
|                                                                                      |
| 01:AGGREGATE [FINALIZE]                                                              |
| |  group by: year                                                                    |
| |                                                                                    |
| 00:SCAN HDFS [key_cols.t1]                                                           |
|    partitions=0/0 files=0 size=0B                                                    |
+--------------------------------------------------------------------------------------+
```

The following examples show how the plan is made more efficient when the OPTIMIZE_PARTITION_KEY_SCANS option is enabled:

```
set optimize_partition_key_scans=1;
OPTIMIZE_PARTITION_KEY_SCANS set to 1

-- The aggregation operation is turned into a UNION internally,
```

```
-- with constant values known in advance based on the metadata
-- for the partitioned table.
explain select distinct year from t1;
+-----------------------------------------------------+
| Explain String                                      |
+-----------------------------------------------------+
| Estimated Per-Host Requirements: Memory=0B VCores=0 |
|                                                     |
| F00:PLAN FRAGMENT [UNPARTITIONED]                   |
|   01:AGGREGATE [FINALIZE]                           |
|   |  group by: year                                 |
|   |  hosts=1 per-host-mem=unavailable               |
|   |  tuple-ids=1 row-size=4B cardinality=3          |
|   |                                                 |
|   00:UNION                                          |
|      constant-operands=3                            |
|      hosts=1 per-host-mem=unavailable               |
|      tuple-ids=0 row-size=4B cardinality=3          |
+-----------------------------------------------------+

-- The same optimization applies to other aggregation queries
-- that only return values based on partition key columns:
-- MIN, MAX, COUNT(DISTINCT), and so on.
explain select min(year) from t1;
+-----------------------------------------------------+
| Explain String                                      |
+-----------------------------------------------------+
| Estimated Per-Host Requirements: Memory=0B VCores=0 |
|                                                     |
| F00:PLAN FRAGMENT [UNPARTITIONED]                   |
|   01:AGGREGATE [FINALIZE]                           |
|   |  output: min(year)                              |
|   |  hosts=1 per-host-mem=unavailable               |
|   |  tuple-ids=1 row-size=4B cardinality=1          |
|   |                                                 |
|   00:UNION                                          |
|      constant-operands=3                            |
|      hosts=1 per-host-mem=unavailable               |
|      tuple-ids=0 row-size=4B cardinality=3          |
+-----------------------------------------------------+
```

PARQUET_COMPRESSION_CODEC Query Option

Deprecated. Use `COMPRESSION_CODEC` in Impala 2.0 and later. See COMPRESSION_CODEC Query Option (CDH 5.2 or higher only) on page 324 for details.

PARQUET_ANNOTATE_STRINGS_UTF8 Query Option (CDH 5.8 or higher only)

Causes Impala `INSERT` and `CREATE TABLE AS SELECT` statements to write Parquet files that use the UTF-8 annotation for `STRING` columns.

Usage notes:

By default, Impala represents a `STRING` column in Parquet as an unannotated binary field.

Impala always uses the UTF-8 annotation when writing `CHAR` and `VARCHAR` columns to Parquet files. An alternative to using the query option is to cast `STRING` values to `VARCHAR`.

This option is to help make Impala-written data more interoperable with other data processing engines. Impala itself currently does not support all operations on UTF-8 data. Although data processed by Impala is typically represented in ASCII, it is valid to designate the data as UTF-8 when storing on disk, because ASCII is a subset of UTF-8.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of `SET` statement)

Added in: CDH 5.8.0 / Impala 2.6.0

Related information:

Using the Parquet File Format with Impala Tables on page 643

PARQUET_ARRAY_RESOLUTION Query Option (CDH 5.12 or higher only)

The PARQUET_ARRAY_RESOLUTION query option controls the behavior of the indexed-based resolution for nested arrays in Parquet.

In Parquet, you can represent an array using a 2-level or 3-level representation. The modern, standard representation is 3-level. The legacy 2-level scheme is supported for compatibility with older Parquet files. However, there is no reliable metadata within Parquet files to indicate which encoding was used. It is even possible to have mixed encodings within the same file if there are multiple arrays. The PARQUET_ARRAY_RESOLUTION option controls the process of resolution that is to match every column/field reference from a query to a column in the Parquet file.

The supported values for the query option are:

- THREE_LEVEL: Assumes arrays are encoded with the 3-level representation, and does not attempt the 2-level resolution.
- TWO_LEVEL: Assumes arrays are encoded with the 2-level representation, and does not attempt the 3-level resolution.
- TWO_LEVEL_THEN_THREE_LEVEL: First tries to resolve assuming a 2-level representation, and if unsuccessful, tries a 3-level representation.

All of the above options resolve arrays encoded with a single level.

A failure to resolve a column/field reference in a query with a given array resolution policy does not necessarily result in a warning or error returned by the query. A mismatch might be treated like a missing column (returns NULL values), and it is not possible to reliably distinguish the 'bad resolution' and 'legitimately missing column' cases.

The name-based policy generally does not have the problem of ambiguous array representations. You specify to use the name-based policy by setting the PARQUET_FALLBACK_SCHEMA_RESOLUTION query option to NAME.

Type: Enum of TWO_LEVEL, TWO_LEVEL_THEN_THREE_LEVEL, and THREE_LEVEL

Default: THREE_LEVEL

Added in: CDH 5.12.0 / Impala 2.9.0

Examples:

EXAMPLE A: The following Parquet schema of a file can be interpreted as a 2-level or 3-level:

```
ParquetSchemaExampleA {
  optional group single_element_groups (LIST) {
    repeated group single_element_group {
      required int64 count;
    }
  }
}
```

The following table schema corresponds to a 2-level interpretation:

```
CREATE TABLE t (col1 array<struct<f1: bigint>>) STORED AS PARQUET;
```

Successful query with a 2-level interpretation:

```
SET PARQUET_ARRAY_RESOLUTION=TWO_LEVEL;
SELECT ITEM.f1 FROM t.col1;
```

The following table schema corresponds to a 3-level interpretation:

```
CREATE TABLE t (col1 array<bigint>) STORED AS PARQUET;
```

Successful query with a 3-level interpretation:

```
SET PARQUET_ARRAY_RESOLUTION=THREE_LEVEL;
SELECT ITEM FROM t.col1
```

EXAMPLE B: The following Parquet schema of a file can be only be successfully interpreted as a 2-level:

```
ParquetSchemaExampleB {
  required group list_of_ints (LIST) {
    repeated int32 list_of_ints_tuple;
  }
}
```

The following table schema corresponds to a 2-level interpretation:

```
CREATE TABLE t (col1 array<int>) STORED AS PARQUET;
```

Successful query with a 2-level interpretation:

```
SET PARQUET_ARRAY_RESOLUTION=TWO_LEVEL;
SELECT ITEM FROM t.col1
```

Unsuccessful query with a 3-level interpretation. The query returns `NULL`s as if the column was missing in the file:

```
SET PARQUET_ARRAY_RESOLUTION=THREE_LEVEL;
SELECT ITEM FROM t.col1
```

PARQUET_DICTIONARY_FILTERING Query Option (CDH 5.12 or higher only)

The `PARQUET_DICTIONARY_FILTERING` query option controls whether Impala uses dictionary filtering for Parquet files.

To efficiently process a highly selective scan query, when this option is enabled, Impala checks the values in the Parquet dictionary page and determines if the whole row group can be thrown out.

A column chunk is purely dictionary encoded and can be used by dictionary filtering if any of the following conditions are met:

1. If the `encoding_stats` is in the Parquet file, dictionary filtering uses it to determine if there are only dictionary encoded pages (i.e. there are no data pages with an encoding other than PLAIN_DICTIONARY).
2. If the encoding stats are not present, dictionary filtering looks at the encodings. The column is purely dictionary encoded if both of the conditions satisfy:
   - PLAIN_DICTIONARY is present.
   - Only PLAIN_DICTIONARY, RLE, or BIT_PACKED encodings are listed.
3. Dictionary filtering works for the Parquet dictionaries with less than 40000 values if the file was written by or lower.

In the query runtime profile output for each Impalad instance, the `NumDictFilteredRowGroups` field in the SCAN node section shows the number of row groups that were skipped based on dictionary filtering.

Note that row groups can be filtered out by Parquet statistics, and in such cases, dictionary filtering will not be considered.

The supported values for the query option are:

- `true` (`1`): Use dictionary filtering.
- `false` (`0`): Do not use dictionary filtering
- Any other values are treated as `false`.

Type: Boolean

Default: `true` (1)

Added in: CDH 5.12.0 / Impala 2.9.0

PARQUET_FALLBACK_SCHEMA_RESOLUTION Query Option (CDH 5.8 or higher only)

The `PARQUET_FALLBACK_SCHEMA_RESOLUTION` query option allows Impala to look up columns within Parquet files by column name, rather than column order, when necessary. The allowed values are:

- POSITION (0)
- NAME (1)

Usage notes:

By default, Impala looks up columns within a Parquet file based on the order of columns in the table. The `name` setting for this option enables behavior for Impala queries similar to the Hive setting `parquet.column.index access=false`. It also allows Impala to query Parquet files created by Hive with the `parquet.column.index.access=false` setting in effect.

Type: integer or string

Added in: CDH 5.8.0 / Impala 2.6.0

Related information:

Schema Evolution for Parquet Tables on page 654

PARQUET_FILE_SIZE Query Option

Specifies the maximum size of each Parquet data file produced by Impala `INSERT` statements.

Syntax:

Specify the size in bytes, or with a trailing `m` or `g` character to indicate megabytes or gigabytes. For example:

```
-- 128 megabytes.
set PARQUET_FILE_SIZE=134217728
INSERT OVERWRITE parquet_table SELECT * FROM text_table;

-- 512 megabytes.
set PARQUET_FILE_SIZE=512m;
INSERT OVERWRITE parquet_table SELECT * FROM text_table;

-- 1 gigabyte.
set PARQUET_FILE_SIZE=1g;
INSERT OVERWRITE parquet_table SELECT * FROM text_table;
```

Usage notes:

With tables that are small or finely partitioned, the default Parquet block size (formerly 1 GB, now 256 MB in Impala 2.0 and later) could be much larger than needed for each data file. For `INSERT` operations into such tables, you can increase parallelism by specifying a smaller `PARQUET_FILE_SIZE` value, resulting in more HDFS blocks that can be processed by different nodes.

Type: numeric, with optional unit specifier



Important:

Currently, the maximum value for this setting is 1 gigabyte (`1g`). Setting a value higher than 1 gigabyte could result in errors during an `INSERT` operation.

Default: 0 (produces files with a target size of 256 MB; files might be larger for very wide tables)

Isilon considerations:

Because the EMC Isilon storage devices use a global value for the block size rather than a configurable value for each file, the PARQUET_FILE_SIZE query option has no effect when Impala inserts data into a table or partition residing on Isilon storage. Use the `isi` command to set the default block size globally on the Isilon device. For example, to set the Isilon default block size to 256 MB, the recommended size for Parquet data files for Impala, issue the following command:

```
isi hdfs settings modify --default-block-size=256MB
```

Related information:

For information about the Parquet file format, and how the number and size of data files affects query performance, see Using the Parquet File Format with Impala Tables on page 643.

PARQUET_READ_STATISTICS Query Option (CDH 5.12 or higher only)

The PARQUET_READ_STATISTICS query option controls whether to read statistics from Parquet files and use them during query processing.

Parquet stores min/max stats which can be used to skip reading row groups if they don't qualify a certain predicate. When this query option is set to `true`, Impala reads the Parquet statistics and skips reading row groups that do not match the conditions in the WHERE clause.

Impala supports filtering based on Parquet statistics:

- Of the numerical types for the old version of the statistics: Boolean, Integer, Float
- Of the types for the new version of the statistics (starting in IMPALA 2.8): Boolean, Integer, Float, Decimal, String, Timestamp
- For simple predicates of the forms: `<slot> <op> <constant>` or `<constant> <op> <slot>`, where `<op>` is LT, LE, GE, GT, and EQ

The PARQUET_READ_STATISTICS option provides a workaround when dealing with files that have corrupt Parquet statistics and unknown errors.

In the query runtime profile output for each Impalad instance, the `NumStatsFilteredRowGroups` field in the SCAN node section shows the number of row groups that were skipped based on Parquet statistics.

The supported values for the query option are:

- `true` (`1`): Read statistics from Parquet files and use them in query processing.
- `false` (`0`): Do not use Parquet read statistics.
- Any other values are treated as `false`.

Type: Boolean

Default: `true`

Added in: CDH 5.12.0 / Impala 2.9.0

PREFETCH_MODE Query Option (CDH 5.8 or higher only)

Determines whether the prefetching optimization is applied during join query processing.

Type: numeric (0, 1) or corresponding mnemonic strings (NONE, HT_BUCKET).

Default: 1 (equivalent to HT_BUCKET)

Added in: CDH 5.8.0 / Impala 2.6.0

Usage notes:

The default mode is 1, which means that hash table buckets are prefetched during join query processing.

Related information:

Joins in Impala SELECT Statements on page 296, Performance Considerations for Join Queries on page 568.

QUERY_TIMEOUT_S Query Option (CDH 5.2 or higher only)

Sets the idle query timeout value for the session, in seconds. Queries that sit idle for longer than the timeout value are automatically cancelled. If the system administrator specified the `--idle_query_timeout` startup option, `QUERY_TIMEOUT_S` must be smaller than or equal to the `--idle_query_timeout` value.



Note:

The timeout clock for queries and sessions only starts ticking when the query or session is idle.

For queries, this means the query has results ready but is waiting for a client to fetch the data. A query can run for an arbitrary time without triggering a timeout, because the query is computing results rather than sitting idle waiting for the results to be fetched. The timeout period is intended to prevent unclosed queries from consuming resources and taking up slots in the admission count of running queries, potentially preventing other queries from starting.

For sessions, this means that no query has been submitted for some period of time.

Syntax:

```
SET QUERY_TIMEOUT_S=seconds;
```

Type: numeric

Default: 0 (no timeout if `--idle_query_timeout` not in effect; otherwise, use `--idle_query_timeout` value)

Added in: CDH 5.2.0 / Impala 2.0.0

Related information:

Setting Timeout Periods for Daemons, Queries, and Sessions on page 70

REPLICA_PREFERENCE Query Option (CDH 5.9 or higher only)

The `REPLICA_PREFERENCE` query option lets you distribute the work more evenly if hotspots and bottlenecks persist. It causes the access cost of all replicas of a data block to be considered equal to or worse than the configured value. This allows Impala to schedule reads to suboptimal replicas (e.g. local in the presence of cached ones) in order to distribute the work across more executor nodes.

Allowed values are: `CACHE_LOCAL` (`0`), `DISK_LOCAL` (`2`), `REMOTE` (`4`)

Type: Enum

Default: `CACHE_LOCAL (0)`

Added in: CDH 5.9.0 / Impala 2.7.0

Usage Notes:

By default Impala selects the best replica it can find in terms of access cost. The preferred order is cached, local, and remote. With `REPLICA_PREFERENCE`, the preference of all replicas are capped at the selected value. For example, when `REPLICA_PREFERENCE` is set to `DISK_LOCAL`, cached and local replicas are treated with the equal preference. When set to `REMOTE`, all three types of replicas, cached, local, remote, are treated with equal preference.

Related information:

Using HDFS Caching with Impala (CDH 5.3 or higher only) on page 593, SCHEDULE_RANDOM_REPLICA Query Option (CDH 5.7 or higher only) on page 360

REQUEST_POOL Query Option

The pool or queue name that queries should be submitted to. Only applies when you enable the Impala admission control feature. Specifies the name of the pool used by requests from Impala to the resource manager.

Type: `STRING`

Default: empty (use the user-to-pool mapping defined by an `impalad` startup option in the Impala configuration file)

Related information:

Admission Control and Query Queuing on page 549

RESOURCE_TRACE_RATIO Query Option (CDH 6.2 / Impala 3.2 or higher only)

The `RESOURCE_TRACE_RATIO` query option specifies the ratio of queries where the CPU usage info will be included in the profiles. Collecting CPU usage and sending it around adds a slight overhead during query execution. This query option lets you control whether to collect additional information to diagnose the resource usage.

For example, setting `RESOURCE_TRACE_RATIO=1` adds a trace of the CPU usage to the profile of each query.

Setting `RESOURCE_TRACE_RATIO=0.5` means that randomly selected half of all queries will have that information collected by the coordinator and included in the profiles.

Setting `RESOURCE_TRACE_RATIO=0` means that CPU usage will not be tracked and included in the profiles.

Values from 0 to 1 are allowed.

Type: Number

Default: 0

Added in: CDH 6.2

RUNTIME_BLOOM_FILTER_SIZE Query Option (CDH 5.7 or higher only)

Size (in bytes) of Bloom filter data structure used by the runtime filtering feature.



Important:

In CDH 5.8 / Impala 2.6 and higher, this query option only applies as a fallback, when statistics are not available. By default, Impala estimates the optimal size of the Bloom filter structure regardless of the setting for this option. (This is a change from the original behavior in CDH 5.7 / Impala 2.5.)

In CDH 5.8 / Impala 2.6 and higher, when the value of this query option is used for query planning, it is constrained by the minimum and maximum sizes specified by the `RUNTIME_FILTER_MIN_SIZE` and `RUNTIME_FILTER_MAX_SIZE` query options. The filter size is adjusted upward or downward if necessary to fit within the minimum/maximum range.

Type: integer

Default: 1048576 (1 MB)

Maximum: 16 MB

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

This setting affects optimizations for large and complex queries, such as dynamic partition pruning for partitioned tables, and join optimization for queries that join large tables. Larger filters are more effective at handling higher cardinality input sets, but consume more memory per filter.

If your query filters on high-cardinality columns (for example, millions of different values) and you do not get the expected speedup from the runtime filtering mechanism, consider doing some benchmarks with a higher value for `RUNTIME_BLOOM_FILTER_SIZE`. The extra memory devoted to the Bloom filter data structures can help make the filtering more accurate.

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Because the effectiveness of this setting depends so much on query characteristics and data distribution, you typically only use it for specific queries that need some extra tuning, and the ideal value depends on the query. Consider setting this query option immediately before the expensive query and unsetting it immediately afterward.

Kudu considerations:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

Related information:

Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588, RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358, RUNTIME_FILTER_MIN_SIZE Query Option (CDH 5.8 or higher only) on page 357, RUNTIME_FILTER_MAX_SIZE Query Option (CDH 5.8 or higher only) on page 357

RUNTIME_FILTER_MAX_SIZE Query Option (CDH 5.8 or higher only)

The `RUNTIME_FILTER_MAX_SIZE` query option adjusts the settings for the runtime filtering feature. This option defines the maximum size for a filter, no matter what the estimates produced by the planner are. This value also overrides any lower number specified for the `RUNTIME_BLOOM_FILTER_SIZE` query option. Filter sizes are rounded up to the nearest power of two.

Type: integer

Default: 0 (meaning use the value from the corresponding `impalad` startup option)

Added in: CDH 5.8.0 / Impala 2.6.0

Usage notes:

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu considerations:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

Related information:

Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588, RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358, RUNTIME_FILTER_MIN_SIZE Query Option (CDH 5.8 or higher only) on page 357, RUNTIME_BLOOM_FILTER_SIZE Query Option (CDH 5.7 or higher only) on page 356

RUNTIME_FILTER_MIN_SIZE Query Option (CDH 5.8 or higher only)

The `RUNTIME_FILTER_MIN_SIZE` query option adjusts the settings for the runtime filtering feature. This option defines the minimum size for a filter, no matter what the estimates produced by the planner are. This value also overrides any lower number specified for the `RUNTIME_BLOOM_FILTER_SIZE` query option. Filter sizes are rounded up to the nearest power of two.

Type: integer

Default: 0 (meaning use the value from the corresponding `impalad` startup option)

Added in: CDH 5.8.0 / Impala 2.6.0

Usage notes:

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu considerations:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

Related information:

Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588, RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358, RUNTIME_FILTER_MAX_SIZE Query Option (CDH 5.8 or higher only) on page 357, RUNTIME_BLOOM_FILTER_SIZE Query Option (CDH 5.7 or higher only) on page 356

RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only)

The RUNTIME_FILTER_MODE query option adjusts the settings for the runtime filtering feature. It turns this feature on and off, and controls how extensively the filters are transmitted between hosts.

Type: numeric (0, 1, 2) or corresponding mnemonic strings (OFF, LOCAL, GLOBAL).

Default: 2 (equivalent to GLOBAL); formerly was 1 / LOCAL, in CDH 5.7 / Impala 2.5

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

In CDH 5.8 / Impala 2.6 and higher, the default is GLOBAL. This setting is recommended for a wide variety of workloads, to provide best performance with "out of the box" settings.

The lowest setting of LOCAL does a similar level of optimization (such as partition pruning) as in earlier Impala releases. This setting was the default in CDH 5.7 / Impala 2.5, to allow for a period of post-upgrade testing for existing workloads. This setting is suitable for workloads with non-performance-critical queries, or if the coordinator node is under heavy CPU or memory pressure.

You might change the setting to OFF if your workload contains many queries involving partitioned tables or joins that do not experience a performance increase from the runtime filters feature. If the overhead of producing the runtime filters outweighs the performance benefit for queries, you can turn the feature off entirely.

Related information:

Partitioning for Impala Tables on page 625 for details about runtime filtering. DISABLE_ROW_RUNTIME_FILTERING Query Option (CDH 5.7 or higher only) on page 328, RUNTIME_BLOOM_FILTER_SIZE Query Option (CDH 5.7 or higher only) on page 356, RUNTIME_FILTER_WAIT_TIME_MS Query Option (CDH 5.7 or higher only) on page 358, and MAX_NUM_RUNTIME_FILTERS Query Option (CDH 5.7 or higher only) on page 339 for tuning options for runtime filtering.

RUNTIME_FILTER_WAIT_TIME_MS Query Option (CDH 5.7 or higher only)

The RUNTIME_FILTER_WAIT_TIME_MS query option adjusts the settings for the runtime filtering feature. It specifies a time in milliseconds that each scan node waits for runtime filters to be produced by other plan fragments.

Type: integer

Default: 0 (meaning use the value from the corresponding impalad startup option)

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Related information:

Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588, RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358

S3_SKIP_INSERT_STAGING Query Option (CDH 5.8 or higher only)

Speeds up INSERT operations on tables or partitions residing on the Amazon S3 filesystem. The tradeoff is the possibility of inconsistent data left behind if an error occurs partway through the operation.

By default, Impala write operations to S3 tables and partitions involve a two-stage process. Impala writes intermediate files to S3, then (because S3 does not provide a "rename" operation) those intermediate files are copied to their final

location, making the process more expensive as on a filesystem that supports renaming or moving files. This query option makes Impala skip the intermediate files, and instead write the new data directly to the final destination.

Usage notes:



Important:

If a host that is participating in the `INSERT` operation fails partway through the query, you might be left with a table or partition that contains some but not all of the expected data files. Therefore, this option is most appropriate for a development or test environment where you have the ability to reconstruct the table if a problem during `INSERT` leaves the data in an inconsistent state.

The timing of file deletion during an `INSERT OVERWRITE` operation makes it impractical to write new files to S3 and delete the old files in a single operation. Therefore, this query option only affects regular `INSERT` statements that add to the existing data in a table, not `INSERT OVERWRITE` statements. Use `TRUNCATE TABLE` if you need to remove all contents from an S3 table before performing a fast `INSERT` with this option enabled.

Performance improvements with this option enabled can be substantial. The speed increase might be more noticeable for non-partitioned tables than for partitioned tables.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `true` (shown as 1 in output of `SET` statement)

Added in: CDH 5.8.0 / Impala 2.6.0

Related information:

Using Impala with the Amazon S3 Filesystem on page 692

SCAN_BYTES_LIMIT Query Option (CDH 6.1 or higher only)

The `SCAN_BYTES_LIMIT` query option sets a limit on the bytes scanned by HDFS and HBase SCAN operations. If a query is still executing when the query's coordinator detects that it has exceeded the limit, the query is terminated with an error. The option is intended to prevent runaway queries that scan more data than is intended.

For example, an Impala administrator could set a default value of `SCAN_BYTES_LIMIT=100GB` for a resource pool to automatically kill queries that scan more than 100 GB of data (see Impala Admission Control and Query Queuing for information about default query options). If a user accidentally omits a partition filter in a `WHERE` clause and runs a large query that scans a lot of data, the query will be automatically terminated after it scans more data than the `SCAN_BYTES_LIMIT`.

You can override the default value per-query or per-session, in the same way as other query options, if you do not want the default `SCAN_BYTES_LIMIT` value to apply to a specific query or session.



Note:

- Only data actually read from the underlying storage layer is counted towards the limit. E.g. Impala's Parquet scanner employs several techniques to skip over data in a file that is not relevant to a specific query, so often only a fraction of the file size is counted towards `SCAN_BYTES_LIMIT`.
- As of Impala 3.1, bytes scanned by Kudu tablet servers are not counted towards the limit.

Because the checks are done periodically, the query may scan over the limit at times.

Syntax: `SET SCAN_BYTES_LIMIT=bytes;`

Type: numeric

Units:

- A numeric argument represents memory size in bytes.
- Specify a suffix of `m` or `mb` for megabytes.

- Specify a suffix of g or gb for gigabytes.
- If you specify a suffix with unrecognized formats, subsequent queries fail with an error.

Default: `0` (no limit)

Added in: CDH 6.1

SCHEDULE_RANDOM_REPLICA Query Option (CDH 5.7 or higher only)

The `SCHEDULE_RANDOM_REPLICA` query option fine-tunes the scheduling algorithm for deciding which host processes each HDFS data block or Kudu tablet to reduce the chance of CPU hotspots.

By default, Impala estimates how much work each host has done for the query, and selects the host that has the lowest workload. This algorithm is intended to reduce CPU hotspots arising when the same host is selected to process multiple data blocks / tablets. Use the `SCHEDULE_RANDOM_REPLICA` query option if hotspots still arise for some combinations of queries and data layout.

The `SCHEDULE_RANDOM_REPLICA` query option only applies to tables and partitions that are not enabled for the HDFS caching.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false`

Added in: CDH 5.7.0 / Impala 2.5.0

Related information:

Using HDFS Caching with Impala (CDH 5.3 or higher only) on page 593, Avoiding CPU Hotspots for HDFS Cached Data on page 612 , REPLICA_PREFERENCE Query Option (CDH 5.9 or higher only) on page 355

SCRATCH_LIMIT Query Option

Specifies the maximum amount of disk storage, in bytes, that any Impala query can consume on any host using the "spill to disk" mechanism that handles queries that exceed the memory limit.

Syntax:

Specify the size in bytes, or with a trailing `m` or `g` character to indicate megabytes or gigabytes. For example:

```
-- 128 megabytes.
set SCRATCH_LIMIT=134217728

-- 512 megabytes.
set SCRATCH_LIMIT=512m;

-- 1 gigabyte.
set SCRATCH_LIMIT=1g;
```

Usage notes:

A value of zero turns off the spill to disk feature for queries in the current session, causing them to fail immediately if they exceed the memory limit.

The amount of memory used per host for a query is limited by the `MEM_LIMIT` query option.

The more DataNodes in the cluster, the less memory is used on each host, and therefore also less scratch space is required for queries that exceed the memory limit.

Type: numeric, with optional unit specifier

Default: -1 (amount of spill space is unlimited)

Related information:

SQL Operations that Spill to Disk on page 607, MEM_LIMIT Query Option on page 343

SHUFFLE_DISTINCT_EXPRS Query Option

The SHUFFLE_DISTINCT_EXPRS query option controls the shuffling behavior when a query has both grouping and distinct expressions. Impala can optionally include the distinct expressions in the hash exchange to spread the data among more nodes. However, this plan requires one more hash exchange phase.

It is recommended that you turn off this option if the NDVs of the grouping expressions are high.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false`

SUPPORT_START_OVER Query Option

Leave this setting at its default value. It is a read-only setting, tested by some client applications such as Hue.

If you accidentally change it through `impala-shell`, subsequent queries encounter errors until you undo the change by issuing `UNSET support_start_over`.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false`

SYNC_DDL Query Option

When enabled, causes any DDL operation such as `CREATE TABLE` or `ALTER TABLE` to return only when the changes have been propagated to all other Impala nodes in the cluster by the Impala catalog service. That way, if you issue a subsequent `CONNECT` statement in `impala-shell` to connect to a different node in the cluster, you can be sure that other node will already recognize any added or changed tables. (The catalog service automatically broadcasts the DDL changes to all nodes automatically, but without this option there could be a period of inconsistency if you quickly switched to another node, such as by issuing a subsequent query through a load-balancing proxy.)

Although `INSERT` is classified as a DML statement, when the `SYNC_DDL` option is enabled, `INSERT` statements also delay their completion until all the underlying data and metadata changes are propagated to all Impala nodes. Internally, Impala inserts have similarities with DDL statements in traditional database systems, because they create metadata needed to track HDFS block locations for new files and they potentially add new partitions to partitioned tables.



Note: Because this option can introduce a delay after each write operation, if you are running a sequence of `CREATE DATABASE`, `CREATE TABLE`, `ALTER TABLE`, `INSERT`, and similar statements within a setup script, to minimize the overall delay you can enable the `SYNC_DDL` query option only near the end, before the final DDL statement.

Type: Boolean; recognized values are 1 and 0, or `true` and `false`; any other value interpreted as `false`

Default: `false` (shown as 0 in output of `SET` statement)

Related information:

DDL Statements on page 203

THREAD_RESERVATION_AGGREGATE_LIMIT Query Option (CDH 6.1 or higher only)

The `THREAD_RESERVATION_AGGREGATE_LIMIT` query option limits the number of reserved threads for a query across all nodes on which it is executing. The option is intended to prevent execution of complex queries that can consume excessive CPU or operating system resources on a cluster. Queries that have more threads than this threshold are rejected by Impala's admission controller before they start executing.

For example, an Impala administrator could set a default value of `THREAD_RESERVATION_AGGREGATE_LIMIT=2000` for a resource pool on a 100 node where they expect only relatively simple queries with less than 20 threads per node to run. This will reject queries that require more than 2000 reserved threads across all nodes, for example a query with 21 fragments running on all 100 nodes of the cluster.

You can override the default value per-query or per-session, in the same way as other query options, if you do not want the default `THREAD_RESERVATION_AGGREGATE_LIMIT` value to apply to a specific query or session.

Syntax: `SET THREAD_RESERVATION_AGGREGATE_LIMIT=number;`

Type: numeric

Default: 0 (no limit)

Added in: CDH 6.1

THREAD_RESERVATION_LIMIT Query Option (CDH 6.1 or higher only)

The `THREAD_RESERVATION_LIMIT` query option limits the number of reserved threads for a query on each node. The option is intended to prevent execution of complex queries that can consume excessive CPU or operating system resources on a single node. Queries that have more threads per node than this threshold are rejected by Impala's admission controller before they start executing. You can see the number of reserved threads for a query in its explain plan in the "Per-Host Resource Reservation" line.

For example, an Impala administrator could set a default value of `THREAD_RESERVATION_LIMIT=100` for a resource pool where they expect only relatively simple queries to run. This will reject queries that require more than 100 reserved threads on a node, for example, queries with more than 100 fragments.

You can override the default value per-query or per-session, in the same way as other query options, if you do not want the default `THREAD_RESERVATION_LIMIT` value to apply to a specific query or session.

> **Note:** The number of reserved threads on a node may be lower than the maximum value in the explain plan if not all fragments of that query are scheduled on every node.

Syntax: `SET THREAD_RESERVATION_LIMIT=number;`

Type: numeric

Default: 3000

Added in: CDH 6.1

TIMEZONE Query Option (CDH 6.1 / Impala 3.1 or higher only)

The `TIMEZONE` query option defines the timezone used for conversions between UTC and the local time. If not set, Impala uses the system time zone where the Coordinator Impalad runs. As query options are not sent to the Coordinator immediately, the timezones are validated only when the query runs.

Impala takes the timezone into a consideration in the following cases:

- When calling the `NOW()` function
- When converting between Unix time and timestamp if the `use_local_tz_for_unix_timestamp_conversions` flag is `TRUE`
- When reading Parquet timestamps written by Hive if the `convert_legacy_hive_parquet_utc_timestamps` flag is `TRUE`

Syntax:

```
SET TIMEZONE=time zone
```

time zone can be a canonical code or a time zone name defined in IANA Time Zone Database. The value is case-sensitive.

Leading/trailing quotes (') and double quotes (") are stripped.

If *time zone* is an empty string, the time zone for the query is set to the default time zone of the Impalad Coordinator.

If *time zone* is `NULL` or a space character, Impala returns an error when the query is executed.

Type: String

Default: The system time zone where the Coordinator Impalad runs

Examples:

```
SET TIMEZONE=UTC;
SET TIMEZONE="Europe/Budapest";
```

Added in: CDH 6.1

TOPN_BYTES_LIMIT Query Option (CDH 6.1 / Impala 3.1 or higher only)

The `TOPN_BYTES_LIMIT` query option places a limit on the amount of estimated memory that Impala can process for *top-N* queries.

Top-N queries are the queries that include both `ORDER BY` and `LIMIT` clauses. *Top-N* queries don't spill to disk so have to keep all rows they process in memory, and those queries can cause out-of-memory issues when running with a large limit and an offset. If the Impala planner estimates that a *top-N* operator will process more bytes than the `TOPN_BYTES_LIMIT` value, it will replace the *top-N* operator with the *sort* operator. Switching to the *sort* operator allows Impala to spill to disk, thus requiring less memory than *top-N*, but potentially with performance penalties.

The option has no effect when set to 0 or -1.

Syntax:

```
SET TOPN_BYTES_LIMIT=limit
```

Type: Number

Default: 536870912 (512 MB)

Added in: CDH 6.1

SHOW Statement

The `SHOW` statement is a flexible way to get information about different types of Impala objects.

Syntax:

```
SHOW DATABASES [[LIKE] 'pattern']
SHOW SCHEMAS [[LIKE] 'pattern'] - an alias for SHOW DATABASES
SHOW TABLES [IN database_name] [[LIKE] 'pattern']
SHOW [AGGREGATE | ANALYTIC] FUNCTIONS [IN database_name] [[LIKE] 'pattern']
SHOW CREATE TABLE [database_name].table_name
SHOW CREATE VIEW [database_name].view_name
SHOW TABLE STATS [database_name.]table_name
SHOW COLUMN STATS [database_name.]table_name
SHOW [RANGE] PARTITIONS [database_name.]table_name
SHOW FILES IN [database_name.]table_name [PARTITION (key_col_expression [,
key_col_expression]]

SHOW ROLES
SHOW CURRENT ROLES
SHOW ROLE GRANT GROUP group_name
SHOW GRANT ROLE role_name

SHOW GRANT USER user_name
SHOW GRANT USER user_name ON SERVER
SHOW GRANT USER user_name ON DATABASE database_name
SHOW GRANT USER user_name ON TABLE table_name
SHOW GRANT USER user_name ON URI uri
```

Issue a `SHOW object_type` statement to see the appropriate objects in the current database, or `SHOW object_type IN database_name` to see objects in a specific database.

The optional *pattern* argument is a quoted string literal, using Unix-style * wildcards and allowing | for alternation. The preceding LIKE keyword is also optional. All object names are stored in lowercase, so use all lowercase letters in the pattern string. For example:

```
show databases 'a*';
show databases like 'a*';
show tables in some_db like '*fact*';
use some_db;
show tables '*dim*|*fact*';
```

Cancellation: Cannot be cancelled.

SHOW FILES Statement

The SHOW FILES statement displays the files that constitute a specified table, or a partition within a partitioned table. This syntax is available in CDH 5.4 / Impala 2.2 and higher only. The output includes the names of the files, the size of each file, and the applicable partition for a partitioned table. The size includes a suffix of B for bytes, MB for megabytes, and GB for gigabytes.

In CDH 5.10 / Impala 2.8 and higher, you can use general expressions with operators such as <, IN, LIKE, and BETWEEN in the PARTITION clause, instead of only equality operators. For example:

```
show files in sample_table partition (j < 5);
show files in sample_table partition (k = 3, l between 1 and 10);
show files in sample_table partition (month like 'J%');
```



Note: This statement applies to tables and partitions stored on HDFS, or in the Amazon Simple Storage System (S3). It does not apply to views. It does not apply to tables mapped onto HBase or Kudu, because those data management systems do not use the same file-based storage layout.

Usage notes:

You can use this statement to verify the results of your ETL process: that is, that the expected files are present, with the expected sizes. You can examine the file information to detect conditions such as empty files, missing files, or inefficient layouts due to a large number of small files. When you use INSERT statements to copy from one table to another, you can see how the file layout changes due to file format conversions, compaction of small input files into large data blocks, and multiple output files from parallel queries and partitioned inserts.

The output from this statement does not include files that Impala considers to be hidden or invisible, such as those whose names start with a dot or an underscore, or that end with the suffixes .copying or .tmp.

The information for partitioned tables complements the output of the SHOW PARTITIONS statement, which summarizes information about each partition. SHOW PARTITIONS produces some output for each partition, while SHOW FILES does not produce any output for empty partitions because they do not include any data files.

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have read permission for all the table files, read and execute permission for all the directories that make up the table, and execute permission for the database directory and all its parent directories.

Examples:

The following example shows a SHOW FILES statement for an unpartitioned table using text format:

```
[localhost:21000] > create table unpart_text (x bigint, s string);
[localhost:21000] > insert into unpart_text (x, s) select id, name
                  > from oreilly.sample_data limit 20e6;
[localhost:21000] > show files in unpart_text;
+-----------------------------------------------------------------------+----------+-----------+
| path                                                                  | size     | partition |
+-----------------------------------------------------------------------+----------+-----------+
| hdfs://impala_data_dir/d.db/unpart_text/35665776ef85cfaf_1012432410_data.0. | 448.31MB |           |
+-----------------------------------------------------------------------+----------+-----------+
[localhost:21000] > insert into unpart_text (x, s) select id, name from oreilly.sample_data limit 100e6;
```

```
[localhost:21000] > show files in unpart_text;
+-------------------------------------------------------------------------+----------+-----------+
| path                                                                    | size     | partition |
+-------------------------------------------------------------------------+----------+-----------+
| hdfs://impala_data_dir/d.db/unpart_text/35665776ef85cfaf_1012432410_data.0. | 448.31MB |           |
| hdfs://impala_data_dir/d.db/unpart_text/ac3dba252a8952b8_1663177415_data.0. | 2.19GB   |           |
+-------------------------------------------------------------------------+----------+-----------+
```

This example illustrates how, after issuing some `INSERT ... VALUES` statements, the table now contains some tiny files of just a few bytes. Such small files could cause inefficient processing of parallel queries that are expecting multi-megabyte input files. The example shows how you might compact the small files by doing an `INSERT ... SELECT` into a different table, possibly converting the data to Parquet in the process:

```
[localhost:21000] > insert into unpart_text values (10,'hello'), (20, 'world');
[localhost:21000] > insert into unpart_text values (-1,'foo'), (-1000, 'bar');
[localhost:21000] > show files in unpart_text;
+-------------------------------------------------------------------------+----------+
| path                                                                    | size     |
+-------------------------------------------------------------------------+----------+
| hdfs://impala_data_dir/d.db/unpart_text/4f11b8bdf8b6aa92_238145083_data.0.  | 18B      |
| hdfs://impala_data_dir/d.db/unpart_text/35665776ef85cfaf_1012432410_data.0. | 448.31MB |
| hdfs://impala_data_dir/d.db/unpart_text/ac3dba252a8952b8_1663177415_data.0. | 2.19GB   |
| hdfs://impala_data_dir/d.db/unpart_text/cfb8252452445682_1868457216_data.0. | 17B      |
+-------------------------------------------------------------------------+----------+
[localhost:21000] > create table unpart_parq stored as parquet as select * from unpart_text;
+---------------------------+
| summary                   |
+---------------------------+
| Inserted 120000002 row(s) |
+---------------------------+
[localhost:21000] > show files in unpart_parq;
+-------------------------------------------------------------------------+----------+
| path                                                                    | size     |
+-------------------------------------------------------------------------+----------+
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630184_549959007_data.0.parq  | 255.36MB |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630184_549959007_data.1.parq  | 178.52MB |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630185_549959007_data.0.parq  | 255.37MB |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630185_549959007_data.1.parq  | 57.71MB  |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630186_2141167244_data.0.parq | 255.40MB |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630186_2141167244_data.1.parq | 175.52MB |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630187_1006832086_data.0.parq | 255.40MB |
| hdfs://impala_data_dir/d.db/unpart_parq/60798d96ba630187_1006832086_data.1.parq | 214.61MB |
+-------------------------------------------------------------------------+----------+
```

The following example shows a `SHOW FILES` statement for a partitioned text table with data in two different partitions, and two empty partitions. The partitions with no data are not represented in the `SHOW FILES` output.

```
[localhost:21000] > create table part_text (x bigint, y int, s string)
                                > partitioned by (year bigint, month bigint, day bigint);
[localhost:21000] > insert overwrite part_text (x, y, s) partition (year=2014,month=1,day=1)
                  > select id, val, name from oreilly.normalized_parquet
where id between 1 and 1000000;
[localhost:21000] > insert overwrite part_text (x, y, s) partition (year=2014,month=1,day=2)
                  > select id, val, name from oreilly.normalized_parquet
                  > where id between 1000001 and 2000000;
[localhost:21000] > alter table part_text add partition (year=2014,month=1,day=3);
[localhost:21000] > alter table part_text add partition (year=2014,month=1,day=4);
[localhost:21000] > show partitions part_text;
+-------+-------+-----+-------+--------+---------+--------------+-------------------+--------+-------------------+
| year  | month | day | #Rows | #Files | Size    | Bytes Cached | Cache Replication | Format | Incremental stats |
+-------+-------+-----+-------+--------+---------+--------------+-------------------+--------+-------------------+
| 2014  | 1     | 1   | -1    | 4      | 25.16MB | NOT CACHED   | NOT CACHED        | TEXT   | false             |
| 2014  | 1     | 2   | -1    | 4      | 26.22MB | NOT CACHED   | NOT CACHED        | TEXT   | false             |
| 2014  | 1     | 3   | -1    | 0      | 0B      | NOT CACHED   | NOT CACHED        | TEXT   | false             |
| 2014  | 1     | 4   | -1    | 0      | 0B      | NOT CACHED   | NOT CACHED        | TEXT   | false             |
| Total |       |     | -1    | 8      | 51.38MB | 0B           |                   |        |                   |
+-------+-------+-----+-------+--------+---------+--------------+-------------------+--------+-------------------+
[localhost:21000] > show files in part_text;
+-------------------------------------------------------------------------------------------------------+--------+-------------------------+
| path                                                                                                  | size   | partition               |
+-------------------------------------------------------------------------------------------------------+--------+-------------------------+
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=1/80732d9dc80689f_1418645991_data.0.  | 5.77MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=1/80732d9dc8068a0_1418645991_data.0.  | 6.25MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=1/80732d9dc8068a1_147082319_data.0.   | 7.16MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=1/80732d9dc8068a2_2111411753_data.0.  | 5.98MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=2/21a828cf494b5bbb_501271652_data.0.  | 6.42MB | year=2014/month=1/day=2 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=2/21a828cf494b5bbc_501271652_data.0.  | 6.62MB | year=2014/month=1/day=2 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=2/21a828cf494b5bbd_1393490200_data.0. | 6.98MB | year=2014/month=1/day=2 |
| hdfs://impala_data_dir/d.db/part_text/year=2014/month=1/day=2/21a828cf494b5bbe_1393490200_data.0. | 6.20MB | year=2014/month=1/day=2 |
+-------------------------------------------------------------------------------------------------------+--------+-------------------------+
```

The following example shows a `SHOW FILES` statement for a partitioned Parquet table. The number and sizes of files are different from the equivalent partitioned text table used in the previous example, because `INSERT` operations for

Parquet tables are parallelized differently than for text tables. (Also, the amount of data is so small that it can be written to Parquet without involving all the hosts in this 4-node cluster.)

```
[localhost:21000] > create table part_parq (x bigint, y int, s string)
                  > partitioned by (year bigint, month bigint, day bigint) stored as parquet;
[localhost:21000] > insert into part_parq partition (year,month,day) select x, y, s, year, month, day from partitioned_text;
[localhost:21000] > show partitions part_parq;
+-------+-------+-----+-------+--------+---------+--------------+-------------------+---------+-------------------+
| year  | month | day | #Rows | #Files | Size    | Bytes Cached | Cache Replication | Format  | Incremental stats |
+-------+-------+-----+-------+--------+---------+--------------+-------------------+---------+-------------------+
| 2014  | 1     | 1   | -1    | 3      | 17.89MB | NOT CACHED   | NOT CACHED        | PARQUET | false             |
| 2014  | 1     | 2   | -1    | 3      | 17.89MB | NOT CACHED   | NOT CACHED        | PARQUET | false             |
| Total |       |     | -1    | 6      | 35.79MB | 0B           |                   |         |                   |
+-------+-------+-----+-------+--------+---------+--------------+-------------------+---------+-------------------+
[localhost:21000] > show files in part_parq;
+-----------------------------------------------------------------------------------------+--------+------------------------+
| path                                                                                    | size   | partition              |
+-----------------------------------------------------------------------------------------+--------+------------------------+
| hdfs://impala_data_dir/d.db/part_parq/year=2014/month=1/day=1/1134113650_data.0.parq | 4.49MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_parq/year=2014/month=1/day=1/617567880_data.0.parq  | 5.14MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_parq/year=2014/month=1/day=1/2099499416_data.0.parq | 8.27MB | year=2014/month=1/day=1 |
| hdfs://impala_data_dir/d.db/part_parq/year=2014/month=1/day=2/945567189_data.0.parq  | 8.80MB | year=2014/month=1/day=2 |
| hdfs://impala_data_dir/d.db/part_parq/year=2014/month=1/day=2/2145850112_data.0.parq | 4.80MB | year=2014/month=1/day=2 |
| hdfs://impala_data_dir/d.db/part_parq/year=2014/month=1/day=2/665613448_data.0.parq  | 4.29MB | year=2014/month=1/day=2 |
+-----------------------------------------------------------------------------------------+--------+------------------------+
```

The following example shows output from the SHOW FILES statement for a table where the data files are stored in Amazon S3:

```
[localhost:21000] > show files in s3_testing.sample_data_s3;
+-----------------------------------------------------------------------------+---------+
| path                                                                        | size    |
+-----------------------------------------------------------------------------+---------+
| s3a://impala-demo/sample_data/e065453cba1988a6_1733868553_data.0.parq | 24.84MB |
+-----------------------------------------------------------------------------+---------+
```

SHOW ROLES Statement

The SHOW ROLES statement displays roles. This syntax is available in CDH 5.2 / Impala 2.0 and later only, when you are using the Sentry authorization framework along with the Sentry service, as described in Using Impala with the Sentry Service (CDH 5.1 or higher only) on page 91.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

Kudu considerations:

Examples:

Depending on the roles set up within your organization by the CREATE ROLE statement, the output might look something like this:

```
show roles;
+-----------+
| role_name |
+-----------+
| analyst   |
| role1     |
| sales     |
| superuser |
| test_role |
+-----------+
```

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Enabling Sentry Authorization for Impala on page 87

SHOW CURRENT ROLES

The SHOW CURRENT ROLES statement displays roles assigned to the current user. This syntax is available in CDH 5.2 / Impala 2.0 and later only, when you are using the Sentry authorization framework along with the Sentry service, as described in Using Impala with the Sentry Service (CDH 5.1 or higher only) on page 91.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

Kudu considerations:

Examples:

Depending on the roles set up within your organization by the CREATE ROLE statement, the output might look something like this:

```
show current roles;
+-----------+
| role_name |
+-----------+
| role1     |
| superuser |
+-----------+
```

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Enabling Sentry Authorization for Impala on page 87

SHOW ROLE GRANT GROUP Statement

The SHOW ROLE GRANT GROUP statement lists all the roles assigned to the specified group. This statement is only allowed for Sentry administrative users and others users that are part of the specified group. This syntax is available in CDH 5.2 / Impala 2.0 and later only, when you are using the Sentry authorization framework along with the Sentry service, as described in Using Impala with the Sentry Service (CDH 5.1 or higher only) on page 91.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Kudu considerations:

Related information:

Enabling Sentry Authorization for Impala on page 87

SHOW GRANT ROLE Statement

The SHOW GRANT ROLE statement list all the grants for the given role name. This statement is only allowed for Sentry administrative users and other users that have been granted the specified role. This syntax is available in CDH 5.2 / Impala 2.0 and later only, when you are using the Sentry authorization framework along with the Sentry service, as described in Using Impala with the Sentry Service (CDH 5.1 or higher only) on page 91.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Kudu considerations:

Related information:

Enabling Sentry Authorization for Impala on page 87

SHOW GRANT USER Statement

The SHOW GRANT USER statement shows the list of privileges for a given user. This statement is only allowed for Sentry administrative users. However, the current user can run SHOW GRANT USER for themselves.

This syntax is available in CDH 6.1 / Impala 3.1 and later only, when you are using the Sentry authorization framework along with the Sentry service, as described in Using Impala with the Sentry Service (CDH 5.1 or higher only) on page 91.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Enabling Sentry Authorization for Impala on page 87

SHOW DATABASES

The SHOW DATABASES statement is often the first one you issue when connecting to an instance for the first time. You typically issue SHOW DATABASES to see the names you can specify in a USE *db_name* statement, then after switching to a database you issue SHOW TABLES to see the names you can specify in SELECT and INSERT statements.

In CDH 5.7 / Impala 2.5 and higher, the output includes a second column showing any associated comment for each database.

The output of SHOW DATABASES includes the special _impala_builtins database, which lets you view definitions of built-in functions, as described under SHOW FUNCTIONS.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

Examples:

This example shows how you might locate a particular table on an unfamiliar system. The DEFAULT database is the one you initially connect to; a database with that name is present on every system. You can issue SHOW TABLES IN *db_name* without going into a database, or SHOW TABLES once you are inside a particular database.

```
[localhost:21000] > show databases;
+------------------+----------------------------------------------+
| name             | comment                                      |
+------------------+----------------------------------------------+
```

```
| _impala_builtins | System database for Impala builtin functions |
| default          | Default Hive database                        |
| file_formats     |                                              |
+------------------+----------------------------------------------+
Returned 3 row(s) in 0.02s
[localhost:21000] > show tables in file_formats;
+--------------------+
| name               |
+--------------------+
| parquet_table      |
| rcfile_table       |
| sequencefile_table |
| textfile_table     |
+--------------------+
Returned 4 row(s) in 0.01s
[localhost:21000] > use file_formats;
[localhost:21000] > show tables like '*parq*';
+--------------------+
| name               |
+--------------------+
| parquet_table      |
+--------------------+
Returned 1 row(s) in 0.01s
```

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Overview of Impala Databases on page 193, CREATE DATABASE Statement on page 226, DROP DATABASE Statement on page 262, USE Statement on page 385 SHOW TABLES Statement on page 369, SHOW FUNCTIONS Statement on page 379

SHOW TABLES Statement

Displays the names of tables. By default, lists tables in the current database, or with the `IN` clause, in a specified database. By default, lists all tables, or with the `LIKE` clause, only those whose name match a pattern with * wildcards.

Security considerations:

When authorization is enabled, the output of the `SHOW` statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the `SHOW` output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its `LOCATION` attribute.)

Examples:

The following examples demonstrate the `SHOW TABLES` statement. If the database contains no tables, the result set is empty. If the database does contain tables, `SHOW TABLES IN` *db_name* lists all the table names. `SHOW TABLES` with no qualifiers lists all the table names in the current database.

```
create database empty_db;
show tables in empty_db;
Fetched 0 row(s) in 0.11s

create database full_db;
create table full_db.t1 (x int);
create table full_db.t2 like full_db.t1;

show tables in full_db;
+------+
| name |
+------+
| t1   |
```

```
| t2   |
+------+

use full_db;
show tables;
+------+
| name |
+------+
| t1   |
| t2   |
+------+
```

This example demonstrates how SHOW TABLES LIKE '*wildcard_pattern*' lists table names that match a pattern, or multiple alternative patterns. The ability to do wildcard matches for table names makes it helpful to establish naming conventions for tables to conveniently locate a group of related tables.

```
create table fact_tbl (x int);
create table dim_tbl_1 (s string);
create table dim_tbl_2 (s string);

/* Asterisk is the wildcard character. Only 2 out of the 3 just-created tables are
returned. */
show tables like 'dim*';
+-----------+
| name      |
+-----------+
| dim_tbl_1 |
| dim_tbl_2 |
+-----------+

/* We are already in the FULL_DB database, but just to be sure we can specify the database
 name also. */
show tables in full_db like 'dim*';
+-----------+
| name      |
+-----------+
| dim_tbl_1 |
| dim_tbl_2 |
+-----------+

/* The pipe character separates multiple wildcard patterns. */
show tables like '*dim*|t*';
+-----------+
| name      |
+-----------+
| dim_tbl_1 |
| dim_tbl_2 |
| t1        |
| t2        |
+-----------+
```

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

Overview of Impala Tables on page 196, CREATE TABLE Statement on page 234, ALTER TABLE Statement on page 205, DROP TABLE Statement on page 268, DESCRIBE Statement on page 251, SHOW CREATE TABLE Statement on page 370, SHOW TABLE STATS Statement on page 373, SHOW DATABASES on page 368, SHOW FUNCTIONS Statement on page 379

SHOW CREATE TABLE Statement

As a schema changes over time, you might run a CREATE TABLE statement followed by several ALTER TABLE statements. To capture the cumulative effect of all those statements, SHOW CREATE TABLE displays a CREATE TABLE statement that would reproduce the current structure of a table. You can use this output in scripts that set up or clone a group of tables, rather than trying to reproduce the original sequence of CREATE TABLE and ALTER TABLE statements. When creating variations on the original table, or cloning the original table on a different system, you

might need to edit the SHOW CREATE TABLE output to change things such as the database name, LOCATION field, and so on that might be different on the destination system.

If you specify a view name in the SHOW CREATE TABLE, it returns a CREATE VIEW statement with column names and the original SQL statement to reproduce the view. You need the VIEW_METADATA privilege on the view and SELECT privilege on all underlying views and tables to successfully run the SHOW CREATE VIEW statement for a view. The SHOW CREATE VIEW is available as an alias for SHOW CREATE TABLE.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Kudu considerations:

For Kudu tables:

- The column specifications include attributes such as NULL, NOT NULL, ENCODING, and COMPRESSION. If you do not specify those attributes in the original CREATE TABLE statement, the SHOW CREATE TABLE output displays the defaults that were used.
- The specifications of any RANGE clauses are not displayed in full. To see the definition of the range clauses for a Kudu table, use the SHOW RANGE PARTITIONS statement.
- The TBLPROPERTIES output reflects the Kudu master address and the internal Kudu name associated with the Impala table.

```
show CREATE TABLE numeric_grades_default_letter;
+-------------------------------------------------------------------------------------------------+
| result                                                                                          |
+-------------------------------------------------------------------------------------------------+
| CREATE TABLE user.numeric_grades_default_letter (                                               |
|   score TINYINT NOT NULL ENCODING AUTO_ENCODING COMPRESSION DEFAULT_COMPRESSION,                |
|   letter_grade STRING NULL ENCODING AUTO_ENCODING COMPRESSION DEFAULT_COMPRESSION               |
DEFAULT '-', |
|   student STRING NULL ENCODING AUTO_ENCODING COMPRESSION DEFAULT_COMPRESSION,                   |
|   PRIMARY KEY (score)                                                                           |
| )                                                                                               |
| PARTITION BY RANGE (score) (...)                                                                |
| STORED AS KUDU                                                                                  |
| TBLPROPERTIES ('kudu.master_addresses'='vd0342.example.com:7051')                               |
+-------------------------------------------------------------------------------------------------+

show range partitions numeric_grades_default_letter;
+--------------------+
| RANGE (score)      |
+--------------------+
| 0 <= VALUES < 50   |
| 50 <= VALUES < 65  |
| 65 <= VALUES < 80  |
| 80 <= VALUES < 100 |
+--------------------+
```

Examples:

The following example shows how various clauses from the CREATE TABLE statement are represented in the output of SHOW CREATE TABLE.

```
create table show_create_table_demo (id int comment "Unique ID", y double, s string)
  partitioned by (year smallint)
  stored as parquet;

show create table show_create_table_demo;
+-------------------------------------------------------------------------------------------+
| result
  |
+-------------------------------------------------------------------------------------------+
| CREATE TABLE scratch.show_create_table_demo (
  |
|   id INT COMMENT 'Unique ID',
  |
|   y DOUBLE,
  |
|   s STRING
  |
| )
  |
| PARTITIONED BY (
  |
|   year SMALLINT
  |
| )
  |
| STORED AS PARQUET
  |
| LOCATION 'hdfs://127.0.0.1:8020/user/hive/warehouse/scratch.db/show_create_table_demo'
 |
| TBLPROPERTIES ('transient_lastDdlTime'='1418152582')
  |
+-------------------------------------------------------------------------------------------+
```

The following example shows how, after a sequence of ALTER TABLE statements, the output from SHOW CREATE TABLE represents the current state of the table. This output could be used to create a matching table rather than executing the original CREATE TABLE and sequence of ALTER TABLE statements.

```
alter table show_create_table_demo drop column s;
alter table show_create_table_demo set fileformat textfile;

show create table show_create_table_demo;
+-------------------------------------------------------------------------------------------+
| result
  |
+-------------------------------------------------------------------------------------------+
| CREATE TABLE scratch.show_create_table_demo (
  |
|   id INT COMMENT 'Unique ID',
  |
|   y DOUBLE
  |
| )
  |
| PARTITIONED BY (
  |
|   year SMALLINT
  |
| )
  |
| STORED AS TEXTFILE
  |
| LOCATION 'hdfs://127.0.0.1:8020/user/hive/warehouse/demo.db/show_create_table_demo'
  |
| TBLPROPERTIES ('transient_lastDdlTime'='1418152638')
  |
```

```
+------------------------------------------------------------------------------------+
```

Related information:

CREATE TABLE Statement on page 234, DESCRIBE Statement on page 251, SHOW TABLES Statement on page 369

SHOW CREATE VIEW Statement

The SHOW CREATE VIEW, it returns a CREATE VIEW statement with column names and the original SQL statement to reproduce the view. You need the VIEW_METADATA privilege on the view and SELECT privilege on all underlying views and tables to successfully run the SHOW CREATE VIEW statement for a view.

The SHOW CREATE VIEW is an alias for SHOW CREATE TABLE.

SHOW TABLE STATS Statement

The SHOW TABLE STATS and SHOW COLUMN STATS variants are important for tuning performance and diagnosing performance issues, especially with the largest tables and the most complex join queries.

Any values that are not available (because the COMPUTE STATS statement has not been run yet) are displayed as -1.

SHOW TABLE STATS provides some general information about the table, such as the number of files, overall size of the data, whether some or all of the data is in the HDFS cache, and the file format, that is useful whether or not you have run the COMPUTE STATS statement. A -1 in the #Rows output column indicates that the COMPUTE STATS statement has never been run for this table. If the table is partitioned, SHOW TABLE STATS provides this information for each partition. (It produces the same output as the SHOW PARTITIONS statement in this case.)

The output of SHOW COLUMN STATS is primarily only useful after the COMPUTE STATS statement has been run on the table. A -1 in the #Distinct Values output column indicates that the COMPUTE STATS statement has never been run for this table. Currently, Impala always leaves the #Nulls column as -1, even after COMPUTE STATS has been run.

These SHOW statements work on actual tables only, not on views.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

Kudu considerations:

Because Kudu tables do not have characteristics derived from HDFS, such as number of files, file format, and HDFS cache status, the output of SHOW TABLE STATS reflects different characteristics that apply to Kudu tables. If the Kudu table is created with the clause PARTITIONS 20, then the result set of SHOW TABLE STATS consists of 20 rows, each representing one of the numbered partitions. For example:

```
show table stats kudu_table;
+--------+----------+----------+----------------------+------------+
| # Rows | Start Key | Stop Key | Leader Replica       | # Replicas |
+--------+----------+----------+----------------------+------------+
| -1     |          | 00000001 | host.example.com:7050 | 3          |
| -1     | 00000001  | 00000002 | host.example.com:7050 | 3          |
| -1     | 00000002  | 00000003 | host.example.com:7050 | 3          |
| -1     | 00000003  | 00000004 | host.example.com:7050 | 3          |
| -1     | 00000004  | 00000005 | host.example.com:7050 | 3          |
...
```

Impala does not compute the number of rows for each partition for Kudu tables. Therefore, you do not need to re-run COMPUTE STATS when you see -1 in the # Rows column of the output from SHOW TABLE STATS. That column always shows -1 for all Kudu tables.

Examples:

The following examples show how the SHOW TABLE STATS statement displays physical information about a table and the associated data files:

```
show table stats store_sales;
+-------+--------+----------+--------------+--------+------------------+
| #Rows | #Files | Size     | Bytes Cached | Format | Incremental stats |
+-------+--------+----------+--------------+--------+------------------+
| -1    | 1      | 370.45MB | NOT CACHED   | TEXT   | false            |
+-------+--------+----------+--------------+--------+------------------+

show table stats customer;
+-------+--------+---------+--------------+--------+------------------+
| #Rows | #Files | Size    | Bytes Cached | Format | Incremental stats |
+-------+--------+---------+--------------+--------+------------------+
| -1    | 1      | 12.60MB | NOT CACHED   | TEXT   | false            |
+-------+--------+---------+--------------+--------+------------------+
```

The following example shows how, after a COMPUTE STATS or COMPUTE INCREMENTAL STATS statement, the #Rows field is now filled in. Because the STORE_SALES table in this example is not partitioned, the COMPUTE INCREMENTAL STATS statement produces regular stats rather than incremental stats, therefore the Incremental stats field remains false.

```
compute stats customer;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 18 column(s). |
+-----------------------------------------+

show table stats customer;
+--------+--------+---------+--------------+--------+------------------+
| #Rows  | #Files | Size    | Bytes Cached | Format | Incremental stats |
+--------+--------+---------+--------------+--------+------------------+
| 100000 | 1      | 12.60MB | NOT CACHED   | TEXT   | false            |
+--------+--------+---------+--------------+--------+------------------+

compute incremental stats store_sales;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 23 column(s). |
+-----------------------------------------+

show table stats store_sales;
+---------+--------+----------+--------------+--------+------------------+
| #Rows   | #Files | Size     | Bytes Cached | Format | Incremental stats |
+---------+--------+----------+--------------+--------+------------------+
| 2880404 | 1      | 370.45MB | NOT CACHED   | TEXT   | false            |
+---------+--------+----------+--------------+--------+------------------+
```

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.) The Impala user must also have execute permission for the database directory, and any parent directories of the database directory in HDFS.

Related information:

COMPUTE STATS Statement on page 219, SHOW COLUMN STATS Statement on page 375

See Table and Column Statistics on page 575 for usage information and examples.

SHOW COLUMN STATS Statement

The SHOW TABLE STATS and SHOW COLUMN STATS variants are important for tuning performance and diagnosing performance issues, especially with the largest tables and the most complex join queries.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

Kudu considerations:

The output for SHOW COLUMN STATS includes the relevant information for Kudu tables. The information for column statistics that originates in the underlying Kudu storage layer is also represented in the metastore database that Impala uses.

Examples:

The following examples show the output of the SHOW COLUMN STATS statement for some tables, before the COMPUTE STATS statement is run. Impala deduces some information, such as maximum and average size for fixed-length columns, and leaves and unknown values as -1.

```
show column stats customer;
+------------------------+--------+------------------+--------+----------+----------+
| Column                 | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+------------------------+--------+------------------+--------+----------+----------+
| c_customer_sk          | INT    | -1               | -1     | 4        | 4        |
| c_customer_id          | STRING | -1               | -1     | -1       | -1       |
| c_current_cdemo_sk     | INT    | -1               | -1     | 4        | 4        |
| c_current_hdemo_sk     | INT    | -1               | -1     | 4        | 4        |
| c_current_addr_sk      | INT    | -1               | -1     | 4        | 4        |
| c_first_shipto_date_sk | INT    | -1               | -1     | 4        | 4        |
| c_first_sales_date_sk  | INT    | -1               | -1     | 4        | 4        |
| c_salutation           | STRING | -1               | -1     | -1       | -1       |
| c_first_name           | STRING | -1               | -1     | -1       | -1       |
| c_last_name            | STRING | -1               | -1     | -1       | -1       |
| c_preferred_cust_flag  | STRING | -1               | -1     | -1       | -1       |
| c_birth_day            | INT    | -1               | -1     | 4        | 4        |
| c_birth_month          | INT    | -1               | -1     | 4        | 4        |
| c_birth_year           | INT    | -1               | -1     | 4        | 4        |
| c_birth_country        | STRING | -1               | -1     | -1       | -1       |
| c_login                | STRING | -1               | -1     | -1       | -1       |
| c_email_address        | STRING | -1               | -1     | -1       | -1       |
| c_last_review_date     | STRING | -1               | -1     | -1       | -1       |
+------------------------+--------+------------------+--------+----------+----------+

show column stats store_sales;
+----------------------+-------+------------------+--------+----------+----------+
| Column               | Type  | #Distinct Values | #Nulls | Max Size | Avg Size |
+----------------------+-------+------------------+--------+----------+----------+
| ss_sold_date_sk      | INT   | -1               | -1     | 4        | 4        |
| ss_sold_time_sk      | INT   | -1               | -1     | 4        | 4        |
| ss_item_sk           | INT   | -1               | -1     | 4        | 4        |
| ss_customer_sk       | INT   | -1               | -1     | 4        | 4        |
| ss_cdemo_sk          | INT   | -1               | -1     | 4        | 4        |
| ss_hdemo_sk          | INT   | -1               | -1     | 4        | 4        |
| ss_addr_sk           | INT   | -1               | -1     | 4        | 4        |
| ss_store_sk          | INT   | -1               | -1     | 4        | 4        |
| ss_promo_sk          | INT   | -1               | -1     | 4        | 4        |
| ss_ticket_number     | INT   | -1               | -1     | 4        | 4        |
| ss_quantity          | INT   | -1               | -1     | 4        | 4        |
| ss_wholesale_cost    | FLOAT | -1               | -1     | 4        | 4        |
| ss_list_price        | FLOAT | -1               | -1     | 4        | 4        |
| ss_sales_price       | FLOAT | -1               | -1     | 4        | 4        |
| ss_ext_discount_amt  | FLOAT | -1               | -1     | 4        | 4        |
| ss_ext_sales_price   | FLOAT | -1               | -1     | 4        | 4        |
| ss_ext_wholesale_cost| FLOAT | -1               | -1     | 4        | 4        |
| ss_ext_list_price    | FLOAT | -1               | -1     | 4        | 4        |
| ss_ext_tax           | FLOAT | -1               | -1     | 4        | 4        |
```

```
| ss_coupon_amt         | FLOAT | -1               | -1     | 4        | 4        |
| ss_net_paid           | FLOAT | -1               | -1     | 4        | 4        |
| ss_net_paid_inc_tax   | FLOAT | -1               | -1     | 4        | 4        |
| ss_net_profit         | FLOAT | -1               | -1     | 4        | 4        |
+-----------------------+-------+------------------+--------+----------+----------+
```

The following examples show the output of the `SHOW COLUMN STATS` statement for some tables, after the `COMPUTE STATS` statement is run. Now most of the `-1` values are changed to reflect the actual table data. The `#Nulls` column remains `-1` because Impala does not use the number of `NULL` values to influence query planning.

```
compute stats customer;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 18 column(s). |
+-----------------------------------------+

compute stats store_sales;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 23 column(s). |
+-----------------------------------------+

show column stats customer;
+------------------------+--------+------------------+--------+----------+----------+
| Column                 | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+------------------------+--------+------------------+--------+----------+----------+
| c_customer_sk          | INT    | 139017           | -1     | 4        | 4        |
| c_customer_id          | STRING | 111904           | -1     | 16       | 16       |
| c_current_cdemo_sk     | INT    | 95837            | -1     | 4        | 4        |
| c_current_hdemo_sk     | INT    | 8097             | -1     | 4        | 4        |
| c_current_addr_sk      | INT    | 57334            | -1     | 4        | 4        |
| c_first_shipto_date_sk | INT    | 4374             | -1     | 4        | 4        |
| c_first_sales_date_sk  | INT    | 4409             | -1     | 4        | 4        |
| c_salutation           | STRING | 7                | -1     | 4        | 3.1308   |
| c_first_name           | STRING | 3887             | -1     | 11       | 5.6356   |
| c_last_name            | STRING | 4739             | -1     | 13       | 5.9106   |
| c_preferred_cust_flag  | STRING | 3                | -1     | 1        | 0.9656   |
| c_birth_day            | INT    | 31               | -1     | 4        | 4        |
| c_birth_month          | INT    | 12               | -1     | 4        | 4        |
| c_birth_year           | INT    | 71               | -1     | 4        | 4        |
| c_birth_country        | STRING | 205              | -1     | 20       | 8.4001   |
| c_login                | STRING | 1                | -1     | 0        | 0        |
| c_email_address        | STRING | 94492            | -1     | 46       | 26.485   |
| c_last_review_date     | STRING | 349              | -1     | 7        | 6.7561   |
+------------------------+--------+------------------+--------+----------+----------+

show column stats store_sales;
+-----------------------+-------+------------------+--------+----------+----------+
| Column                | Type  | #Distinct Values | #Nulls | Max Size | Avg Size |
+-----------------------+-------+------------------+--------+----------+----------+
| ss_sold_date_sk       | INT   | 4395             | -1     | 4        | 4        |
| ss_sold_time_sk       | INT   | 63617            | -1     | 4        | 4        |
| ss_item_sk            | INT   | 19463            | -1     | 4        | 4        |
| ss_customer_sk        | INT   | 122720           | -1     | 4        | 4        |
| ss_cdemo_sk           | INT   | 242982           | -1     | 4        | 4        |
| ss_hdemo_sk           | INT   | 8097             | -1     | 4        | 4        |
| ss_addr_sk            | INT   | 70770            | -1     | 4        | 4        |
| ss_store_sk           | INT   | 6                | -1     | 4        | 4        |
| ss_promo_sk           | INT   | 355              | -1     | 4        | 4        |
| ss_ticket_number      | INT   | 304098           | -1     | 4        | 4        |
| ss_quantity           | INT   | 105              | -1     | 4        | 4        |
| ss_wholesale_cost     | FLOAT | 9600             | -1     | 4        | 4        |
| ss_list_price         | FLOAT | 22191            | -1     | 4        | 4        |
| ss_sales_price        | FLOAT | 20693            | -1     | 4        | 4        |
| ss_ext_discount_amt   | FLOAT | 228141           | -1     | 4        | 4        |
| ss_ext_sales_price    | FLOAT | 433550           | -1     | 4        | 4        |
| ss_ext_wholesale_cost | FLOAT | 406291           | -1     | 4        | 4        |
| ss_ext_list_price     | FLOAT | 574871           | -1     | 4        | 4        |
| ss_ext_tax            | FLOAT | 91806            | -1     | 4        | 4        |
| ss_coupon_amt         | FLOAT | 228141           | -1     | 4        | 4        |
```

```
| ss_net_paid           | FLOAT | 493107           | -1     | 4        | 4        |
| ss_net_paid_inc_tax   | FLOAT | 653523           | -1     | 4        | 4        |
| ss_net_profit         | FLOAT | 611934           | -1     | 4        | 4        |
+-----------------------+-------+------------------+--------+----------+----------+
```

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its `LOCATION` attribute.) The Impala user must also have execute permission for the database directory, and any parent directories of the database directory in HDFS.

Related information:

COMPUTE STATS Statement on page 219, SHOW TABLE STATS Statement on page 373

See Table and Column Statistics on page 575 for usage information and examples.

SHOW PARTITIONS Statement

`SHOW PARTITIONS` displays information about each partition for a partitioned table. (The output is the same as the `SHOW TABLE STATS` statement, but `SHOW PARTITIONS` only works on a partitioned table.) Because it displays table statistics for all partitions, the output is more informative if you have run the `COMPUTE STATS` statement after creating all the partitions. See COMPUTE STATS Statement on page 219 for details. For example, on a `CENSUS` table partitioned on the `YEAR` column:

Because Kudu tables are all considered to be partitioned, the `SHOW PARTITIONS` statement works for any Kudu table. The default output is the same as for `SHOW TABLE STATS`, with the same Kudu-specific columns in the result set:

```
show table stats kudu_table;
+--------+-----------+----------+----------------------+------------+
| # Rows | Start Key | Stop Key | Leader Replica       | # Replicas |
+--------+-----------+----------+----------------------+------------+
| -1     |           | 00000001 | host.example.com:7050 | 3          |
| -1     | 00000001  | 00000002 | host.example.com:7050 | 3          |
| -1     | 00000002  | 00000003 | host.example.com:7050 | 3          |
| -1     | 00000003  | 00000004 | host.example.com:7050 | 3          |
| -1     | 00000004  | 00000005 | host.example.com:7050 | 3          |
...
```

Security considerations:

When authorization is enabled, the output of the `SHOW` statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the `SHOW` output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

Kudu considerations:

The optional `RANGE` clause only applies to Kudu tables. It displays only the partitions defined by the `RANGE` clause of `CREATE TABLE` or `ALTER TABLE`.

Although you can specify < or <= comparison operators when defining range partitions for Kudu tables, Kudu rewrites them if necessary to represent each range as *`low_bound`* `<= VALUES <` *`high_bound`*. This rewriting might involve incrementing one of the boundary values or appending a `\0` for string values, so that the partition covers the same range as originally specified.

Examples:

The following example shows the output for a Parquet, text, or other HDFS-backed table partitioned on the YEAR column:

```
[localhost:21000] > show partitions census;
+-------+-------+--------+------+---------+
| year  | #Rows | #Files | Size | Format  |
+-------+-------+--------+------+---------+
| 2000  | -1    | 0      | 0B   | TEXT    |
| 2004  | -1    | 0      | 0B   | TEXT    |
| 2008  | -1    | 0      | 0B   | TEXT    |
| 2010  | -1    | 0      | 0B   | TEXT    |
| 2011  | 4     | 1      | 22B  | TEXT    |
| 2012  | 4     | 1      | 22B  | TEXT    |
| 2013  | 1     | 1      | 231B | PARQUET |
| Total | 9     | 3      | 275B |         |
+-------+-------+--------+------+---------+
```

The following example shows the output for a Kudu table using the hash partitioning mechanism. The number of rows in the result set corresponds to the values used in the PARTITIONS *N* clause of CREATE TABLE.

```
show partitions million_rows_hash;

+--------+-----------+----------+----------------------+--
| # Rows | Start Key | Stop Key | Leader Replica       | # Replicas
+--------+-----------+----------+----------------------+--
| -1     |           | 00000001 | n236.example.com:7050 | 3
| -1     | 00000001  | 00000002 | n236.example.com:7050 | 3
| -1     | 00000002  | 00000003 | n336.example.com:7050 | 3
| -1     | 00000003  | 00000004 | n238.example.com:7050 | 3
| -1     | 00000004  | 00000005 | n338.example.com:7050 | 3
....
| -1     | 0000002E  | 0000002F | n240.example.com:7050 | 3
| -1     | 0000002F  | 00000030 | n336.example.com:7050 | 3
| -1     | 00000030  | 00000031 | n240.example.com:7050 | 3
| -1     | 00000031  |          | n334.example.com:7050 | 3
+--------+-----------+----------+----------------------+--
Fetched 50 row(s) in 0.05s
```

The following example shows the output for a Kudu table using the range partitioning mechanism:

```
show range partitions million_rows_range;
+-----------------------+
| RANGE (id)            |
+-----------------------+
| VALUES < "A"          |
| "A" <= VALUES < "["   |
| "a" <= VALUES < "{"   |
| "{" <= VALUES < "~\0" |
+-----------------------+
```

HDFS permissions:

The user ID that the impalad daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.) The Impala user must also have execute permission for the database directory, and any parent directories of the database directory in HDFS.

Related information:

See Table and Column Statistics on page 575 for usage information and examples.

SHOW TABLE STATS Statement on page 373, Partitioning for Impala Tables on page 625

SHOW FUNCTIONS Statement

By default, SHOW FUNCTIONS displays user-defined functions (UDFs) and SHOW AGGREGATE FUNCTIONS displays user-defined aggregate functions (UDAFs) associated with a particular database. The output from SHOW FUNCTIONS includes the argument signature of each function. You specify this argument signature as part of the DROP FUNCTION statement. You might have several UDFs with the same name, each accepting different argument data types.

Usage notes:

In CDH 5.7 / Impala 2.5 and higher, the SHOW FUNCTIONS output includes a new column, labelled is persistent. This property is true for Impala built-in functions, C++ UDFs, and Java UDFs created using the new CREATE FUNCTION syntax with no signature. It is false for Java UDFs created using the old CREATE FUNCTION syntax that includes the types for the arguments and return value. Any functions with false shown for this property must be created again by the CREATE FUNCTION statement each time the Impala catalog server is restarted. See CREATE FUNCTION for information on switching to the new syntax, so that Java UDFs are preserved across restarts. Java UDFs that are persisted this way are also easier to share across Impala and Hive.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object. See Enabling Sentry Authorization for Impala on page 87 for how to set up authorization and add privileges for specific objects.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

To display Impala built-in functions, specify the special database name _impala_builtins:

```
show functions in _impala_builtins;
+-------------+-----------------------------------------------+-------------+--------------+
| return type | signature                                     | binary type | is
persistent |
+-------------+-----------------------------------------------+-------------+--------------+
| BIGINT      | abs(BIGINT)                                   | BUILTIN     | true
           |
| DECIMAL(*,*) | abs(DECIMAL(*,*))                            | BUILTIN     | true
           |
| DOUBLE      | abs(DOUBLE)                                   | BUILTIN     | true
           |
| FLOAT       | abs(FLOAT)                                    | BUILTIN     | true
           |
+----------------+----------------------------------------+
...

show functions in _impala_builtins like '*week*';
+-------------+-------------------------------+-------------+---------------+
| return type | signature                     | binary type | is persistent |
+-------------+-------------------------------+-------------+---------------+
| INT         | dayofweek(TIMESTAMP)          | BUILTIN     | true          |
| INT         | weekofyear(TIMESTAMP)         | BUILTIN     | true          |
| TIMESTAMP   | weeks_add(TIMESTAMP, BIGINT)  | BUILTIN     | true          |
| TIMESTAMP   | weeks_add(TIMESTAMP, INT)     | BUILTIN     | true          |
| TIMESTAMP   | weeks_sub(TIMESTAMP, BIGINT)  | BUILTIN     | true          |
| TIMESTAMP   | weeks_sub(TIMESTAMP, INT)     | BUILTIN     | true          |
+-------------+-------------------------------+-------------+---------------+
```

Related information:

Overview of Impala Functions on page 194, Impala Built-In Functions on page 391, User-Defined Functions (UDFs) on page 525, SHOW DATABASES on page 368, SHOW TABLES Statement on page 369

SHUTDOWN Statement

The SHUTDOWN statement performs a graceful shutdown of Impala Daemon. The Impala daemon will notify other Impala daemons that it is shutting down, wait for a grace period, then shut itself down once no more queries or

fragments are executing on that daemon. The `--shutdown_grace_period_s` flag determines the duration of the grace period in seconds.

Syntax:

```
:SHUTDOWN()
:SHUTDOWN([host_name[:port_number] )
:SHUTDOWN(deadline)
:SHUTDOWN([host_name[:port_number], deadline)
```

Usage notes:

All arguments are optional for SHUTDOWN.

Argument	Type	Default	Description
host_name	STRING	The current impalad host to whom the SHUTDOWN statement is submitted.	Address of the impalad to be shut down.
port_number	INT	• In Impala 3.1 / CDH 6.1, the current impalad's port used for the thrift based communication with other impalads (by default, 22000). • In Impala 3.2 and higher, the current impalad's port used for the KRPC based communication with other impalads (by default, 27000).	Specifies the port by which the impalad can be contacted. • In Impala 3.1 / CDH 6.1, use the same impalad port used for the thrift based inter-Impala communication. • In Impala 3.2 and higher, use the same impalad port used for the KRPC based inter-Impala communication.
deadline	INT	The value of the `--shutdown_deadline_s` flag, which defaults to 1 hour.	*deadline* must be a non-negative number, specified in seconds. The value, 0, for *deadline* specifies an immediate shutdown.

Take the following points into consideration when running the SHUTDOWN statement:

- A client can shut down the coordinator impalad that it is connected to via `:SHUTDOWN()`.
- A client can remotely shut down any impalad via `:SHUTDOWN('hostname')`.
- The shutdown time limit can be overridden to force a quicker or slower shutdown by specifying a deadline. The default deadline is determined by the `--shutdown_deadline_s` flag, which defaults to 1 hour.
- Executors can be shut down without disrupting running queries. Short-running queries will finish, and long-running queries will continue until the *deadline* is reached.
- If queries are submitted to a coordinator after shutdown of that coordinator has started, they will fail.
- Long running queries or other issues, such as stuck fragments, will slow down but not prevent eventual shutdown.

Security considerations:

The ALL privilege is required on the server.

Cancellation: Cannot be cancelled.

Examples:

```
:SHUTDOWN(); -- Shut down the current impalad  with the default deadline.
:SHUTDOWN('hostname'); --  Shut down impalad running on hostname  with the default
deadline.
:SHUTDOWN(\"hostname:1234\"); -- Shut down impalad running on host at port 1234  with
the default deadline.
:SHUTDOWN(10); - Shut down the current impalad after 10 seconds.
:SHUTDOWN('hostname', 10); - Shut down impalad running on hostname when all queries
running on hostname finish, or after 10 seconds.
:SHUTDOWN('hostname:11', 10 * 60); -- Shut down impalad running on hostname at port 11
```

```
 when all queries running on hostname finish, or after 600 seconds.
:SHUTDOWN(0); -- Perform an immdediate shutdown of the current impalad.
```

Added in: CDH 6.1

TRUNCATE TABLE Statement (CDH 5.5 or higher only)

Removes the data from an Impala table while leaving the table itself.

Syntax:

```
TRUNCATE [TABLE] [IF EXISTS] [db_name.]table_name
```

Statement type: DDL

Usage notes:

Often used to empty tables that are used during ETL cycles, after the data has been copied to another table for the next stage of processing. This statement is a low-overhead alternative to dropping and recreating the table, or using `INSERT OVERWRITE` to replace the data during the next ETL cycle.

This statement removes all the data and associated data files in the table. It can remove data files from internal tables, external tables, partitioned tables, and tables mapped to HBase or the Amazon Simple Storage Service (S3). The data removal applies to the entire table, including all partitions of a partitioned table.

Any statistics produced by the `COMPUTE STATS` statement are reset when the data is removed.

Make sure that you are in the correct database before truncating a table, either by issuing a `USE` statement first or by using a fully qualified name *db_name*`.`*table_name*.

The optional `TABLE` keyword does not affect the behavior of the statement.

The optional `IF EXISTS` clause makes the statement succeed whether or not the table exists. If the table does exist, it is truncated; if it does not exist, the statement has no effect. This capability is useful in standardized setup scripts that are might be run both before and after some of the tables exist. This clause is available in CDH 5.7 / Impala 2.5 and higher.

For other tips about managing and reclaiming Impala disk space, see Managing Disk Space for Impala Data on page 77.

Amazon S3 considerations:

Although Impala cannot write new data to a table stored in the Amazon S3 filesystem, the `TRUNCATE TABLE` statement can remove data files from S3. See Using Impala with the Amazon S3 Filesystem on page 692 for details about working with S3 tables.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the `impala` user, must have write permission for all the files and directories that make up the table.

Kudu considerations:

Currently, the `TRUNCATE TABLE` statement cannot be used with Kudu tables.

Examples:

The following example shows a table containing some data and with table and column statistics. After the `TRUNCATE TABLE` statement, the data is removed and the statistics are reset.

```
CREATE TABLE truncate_demo (x INT);
INSERT INTO truncate_demo VALUES (1), (2), (4), (8);
SELECT COUNT(*) FROM truncate_demo;
+----------+
```

```
| count(*) |
+----------+
| 4        |
+----------+
COMPUTE STATS truncate_demo;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
+-----------------------------------------+
SHOW TABLE STATS truncate_demo;
+-------+--------+------+--------------+-------------------+--------+------------------+
| #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incremental stats
 |
+-------+--------+------+--------------+-------------------+--------+------------------+
| 4     | 1      | 8B   | NOT CACHED   | NOT CACHED        | TEXT   | false
  |
+-------+--------+------+--------------+-------------------+--------+------------------+
SHOW COLUMN STATS truncate_demo;
+--------+------+------------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+------+------------------+--------+----------+----------+
| x      | INT  | 4                | -1     | 4        | 4        |
+--------+------+------------------+--------+----------+----------+

-- After this statement, the data and the table/column stats will be gone.
TRUNCATE TABLE truncate_demo;

SELECT COUNT(*) FROM truncate_demo;
+----------+
| count(*) |
+----------+
| 0        |
+----------+
SHOW TABLE STATS truncate_demo;
+-------+--------+------+--------------+-------------------+--------+------------------+
| #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incremental stats
 |
+-------+--------+------+--------------+-------------------+--------+------------------+
| -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT   | false
  |
+-------+--------+------+--------------+-------------------+--------+------------------+
SHOW COLUMN STATS truncate_demo;
+--------+------+------------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+------+------------------+--------+----------+----------+
| x      | INT  | -1               | -1     | 4        | 4        |
+--------+------+------------------+--------+----------+----------+
```

The following example shows how the `IF EXISTS` clause allows the `TRUNCATE TABLE` statement to be run without error whether or not the table exists:

```
CREATE TABLE staging_table1 (x INT, s STRING);
Fetched 0 row(s) in 0.33s

SHOW TABLES LIKE 'staging*';
+----------------+
| name           |
+----------------+
| staging_table1 |
+----------------+
Fetched 1 row(s) in 0.25s

-- Our ETL process involves removing all data from several staging tables
-- even though some might be already dropped, or not created yet.

TRUNCATE TABLE IF EXISTS staging_table1;
Fetched 0 row(s) in 5.04s

TRUNCATE TABLE IF EXISTS staging_table2;
Fetched 0 row(s) in 0.25s
```

```
TRUNCATE TABLE IF EXISTS staging_table3;
Fetched 0 row(s) in 0.25s
```

Related information:

Overview of Impala Tables on page 196, ALTER TABLE Statement on page 205, CREATE TABLE Statement on page 234, Partitioning for Impala Tables on page 625, Internal Tables on page 197, External Tables on page 197

UPDATE Statement (CDH 5.10 or higher only)

Updates an arbitrary number of rows in a Kudu table. This statement only works for Impala tables that use the Kudu storage engine.

Syntax:

```
UPDATE [database_name.]table_name SET col = val [, col = val ... ]
  [ FROM joined_table_refs ]
  [ WHERE where_conditions ]
```

Usage notes:

None of the columns that make up the primary key can be updated by the SET clause.

The conditions in the WHERE clause are the same ones allowed for the SELECT statement. See SELECT Statement on page 295 for details.

If the WHERE clause is omitted, all rows in the table are updated.

The conditions in the WHERE clause can refer to any combination of primary key columns or other columns. Referring to primary key columns in the WHERE clause is more efficient than referring to non-primary key columns.

Because Kudu currently does not enforce strong consistency during concurrent DML operations, be aware that the results after this statement finishes might be different than you intuitively expect:

- If some rows cannot be updated because their some primary key columns are not found, due to their being deleted by a concurrent DELETE operation, the statement succeeds but returns a warning.
- An UPDATE statement might also overlap with INSERT, UPDATE, or UPSERT statements running concurrently on the same table. After the statement finishes, there might be more or fewer matching rows than expected in the table because it is undefined whether the UPDATE applies to rows that are inserted or updated while the UPDATE is in progress.

The number of affected rows is reported in an `impala-shell` message and in the query profile.

The optional FROM clause lets you restrict the updates to only the rows in the specified table that are part of the result set for a join query. The join clauses can include non-Kudu tables, but the table from which the rows are deleted must be a Kudu table.

Statement type: DML



Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following examples show how to perform a simple update on a table, with or without a WHERE clause:

```
-- Set all rows to the same value for column c3.
```

```
-- In this case, c1 and c2 are primary key columns
-- and so cannot be updated.
UPDATE kudu_table SET c3 = 'not applicable';

-- Update only the rows that match the condition.
UPDATE kudu_table SET c3 = NULL WHERE c1 > 100 AND c3 IS NULL;

-- Does not update any rows, because the WHERE condition is always false.
UPDATE kudu_table SET c3 = 'impossible' WHERE 1 = 0;

-- Change the values of multiple columns in a single UPDATE statement.
UPDATE kudu_table SET c3 = upper(c3), c4 = FALSE, c5 = 0 WHERE c6 = TRUE;
```

The following examples show how to perform an update using the FROM keyword with a join clause:

```
-- Uppercase a column value, only for rows that have
-- an ID that matches the value from another table.
UPDATE kudu_table SET c3 = upper(c3)
  FROM kudu_table JOIN non_kudu_table
  ON kudu_table.id = non_kudu_table.id;

-- Same effect as previous statement.
-- Assign table aliases in FROM clause, then refer to
-- short names elsewhere in the statement.
UPDATE t1 SET c3 = upper(c3)
  FROM kudu_table t1 JOIN non_kudu_table t2
  ON t1.id = t2.id;

-- Same effect as previous statements, but more efficient.
-- Use WHERE clause to skip updating values that are
-- already uppercase.
UPDATE t1 SET c3 = upper(c3)
  FROM kudu_table t1 JOIN non_kudu_table t2
  ON t1.id = t2.id
  WHERE c3 != upper(c3);
```

Related information:

Using Impala to Query Kudu Tables on page 670, INSERT Statement on page 277, DELETE Statement (CDH 5.10 or higher only) on page 249, UPSERT Statement (CDH 5.10 or higher only) on page 384

UPSERT Statement (CDH 5.10 or higher only)

Acts as a combination of the INSERT and UPDATE statements. For each row processed by the UPSERT statement:

- If another row already exists with the same set of primary key values, the other columns are updated to match the values from the row being “UPSERTed”.
- If there is not any row with the same set of primary key values, the row is created, the same as if the INSERT statement was used.

This statement only works for Impala tables that use the Kudu storage engine.

Syntax:

```
UPSERT [hint_clause] INTO [TABLE] [db_name.]table_name
  [(column_list)]
{
    [hint_clause] select_statement
  | VALUES (value [, value ...]) [, (value [, value ...]) ...]
}

hint_clause ::= [SHUFFLE] | [NOSHUFFLE]
  (Note: the square brackets are part of the syntax.)
```

The *select_statement* clause can use the full syntax, such as WHERE and join clauses, as SELECT Statement on page 295.

Statement type: DML

Usage notes:

If you specify a column list, any omitted columns in the inserted or updated rows are set to their default value (if the column has one) or NULL (if the column does not have a default value). Therefore, if a column is not nullable and has no default value, it must be included in the column list for any UPSERT statement. Because all primary key columns meet these conditions, all the primary key columns must be specified in every UPSERT statement.

Because Kudu tables can efficiently handle small incremental changes, the VALUES clause is more practical to use with Kudu tables than with HDFS-based tables.



Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

```
UPSERT INTO kudu_table (pk, c1, c2, c3) VALUES (0, 'hello', 50, true), (1, 'world', -1,
 false);
UPSERT INTO production_table SELECT * FROM staging_table;
UPSERT INTO production_table SELECT * FROM staging_table WHERE c1 IS NOT NULL AND c2 >
 0;
```

Related information:

Using Impala to Query Kudu Tables on page 670, INSERT Statement on page 277, UPDATE Statement (CDH 5.10 or higher only) on page 383

USE Statement

Switches the current session to a specified database. The ***current database*** is where any CREATE TABLE, INSERT, SELECT, or other statements act when you specify a table or other object name, without prefixing it with a database name. The new current database applies for the duration of the session or unti another USE statement is executed.

Syntax:

```
USE db_name
```

By default, when you connect to an Impala instance, you begin in a database named default.

Usage notes:

Switching the default database is convenient in the following situations:

- To avoid qualifying each reference to a table with the database name. For example, SELECT * FROM t1 JOIN t2 rather than SELECT * FROM db.t1 JOIN db.t2.
- To do a sequence of operations all within the same database, such as creating a table, inserting data, and querying the table.

To start the impala-shell interpreter and automatically issue a USE statement for a particular database, specify the option -d *db_name* for the impala-shell command. The -d option is useful to run SQL scripts, such as setup or test scripts, against multiple databases without hardcoding a USE statement into the SQL source.

Examples:

See CREATE DATABASE Statement on page 226 for examples covering CREATE DATABASE, USE, and DROP DATABASE.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related information:

CREATE DATABASE Statement on page 226, DROP DATABASE Statement on page 262, SHOW DATABASES on page 368

VALUES Statement

In addition to being part of the INSERT statement, the VALUES clause can be used as stand-alone statement or with the SELECT statement to construct a data set without creating a table. For example, the following statement returns a data set of 2 rows and 3 columns.

```
VALUES ('r1_c1', 'r1_c2', 'r1_c3')
     , ('r2_c1', 'r2_c2', 'r2_c3');
```

Syntax:

```
VALUES (row)[, (row), ...];

SELECT select_list FROM (VALUES (row)[, (row), ...]) AS alias;

row ::= column [[AS alias], column [AS alias], ...]
```

- The VALUES keyword is followed by a comma separated list of one or more *row*s.
- *row* is a comma-separated list of one or more *column*s.
- Each *row* must have the same number of *column*s.
- *column* can be a constant, a variable, or an expression.
- The corresponding *column*s must have compatible data types in all *row*s. See the third query in the Examples section below.
- By default, the first row is used to name columns. But using the AS keyword, you can optionally give the column an *alias*.
- If used in the SELECT statement, the AS keyword with an *alias* is required.
- *select_list* is the columns to be selected for the result set.

Examples:

```
> SELECT * FROM (VALUES(4,5,6),(7,8,9)) AS t;
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 4 | 5 | 6 |
| 7 | 8 | 9 |
+---+---+---+

> SELECT * FROM (VALUES(1 AS c1, true AS c2, 'abc' AS c3),(100,false,'xyz')) AS t;
+-----+-------+-----+
| c1  | c2    | c3  |
+-----+-------+-----+
| 1   | true  | abc |
| 100 | false | xyz |
+-----+-------+-----+

> VALUES (CAST('2019-01-01' AS TIMESTAMP)), ('2019-02-02');
+---------------------------------+
| cast('2019-01-01' as timestamp) |
+---------------------------------+
| 2019-01-01 00:00:00             |
| 2019-02-02 00:00:00             |
+---------------------------------+
```

Related information:

SELECT Statement on page 295

Optimizer Hints in Impala

The Impala SQL dialect supports query hints, for fine-tuning the inner workings of queries. Specify hints as a temporary workaround for expensive queries, where missing statistics or other factors cause inefficient performance.

Hints are most often used for the most resource-intensive kinds of Impala queries:

- Join queries involving large tables, where intermediate result sets are transmitted across the network to evaluate the join conditions.
- Inserting into partitioned Parquet tables, where many memory buffers could be allocated on each host to hold intermediate results for each partition.

Syntax:

In CDH 5.2 / Impala 2.0 and higher, you can specify the hints inside comments that use either the `/* */` or `--` notation. Specify a + symbol immediately before the hint name. Recently added hints are only available using the `/* */` and `--` notation. For clarity, the `/* */` and `--` styles are used in the syntax and examples throughout this section. With the `/* */` or `--` notation for hints, specify a + symbol immediately before the first hint name. Multiple hints can be specified separated by commas, for example `/* +clustered,shuffle */`

```
SELECT STRAIGHT_JOIN select_list FROM
join_left_hand_table
  JOIN /* +BROADCAST|SHUFFLE */
join_right_hand_table
remainder_of_query;

SELECT select_list FROM
join_left_hand_table
  JOIN -- +BROADCAST|SHUFFLE
join_right_hand_table
remainder_of_query;

INSERT insert_clauses
  /* +SHUFFLE|NOSHUFFLE */
  SELECT remainder_of_query;

INSERT insert_clauses
  -- +SHUFFLE|NOSHUFFLE
  SELECT remainder_of_query;

INSERT /* +SHUFFLE|NOSHUFFLE */
  insert_clauses
  SELECT remainder_of_query;

INSERT -- +SHUFFLE|NOSHUFFLE
  insert_clauses
  SELECT remainder_of_query;

UPSERT /* +SHUFFLE|NOSHUFFLE */
  upsert_clauses
  SELECT remainder_of_query;

UPSERT -- +SHUFFLE|NOSHUFFLE
  upsert_clauses
  SELECT remainder_of_query;

SELECT select_list FROM
table_ref
  /* +{SCHEDULE_CACHE_LOCAL | SCHEDULE_DISK_LOCAL | SCHEDULE_REMOTE}
    [,RANDOM_REPLICA] */
remainder_of_query;

INSERT insert_clauses
  -- +CLUSTERED
  SELECT remainder_of_query;

INSERT insert_clauses
  /* +CLUSTERED */
  SELECT remainder_of_query;
```

```
INSERT -- +CLUSTERED
  insert_clauses
  SELECT remainder_of_query;

INSERT /* +CLUSTERED */
  insert_clauses
  SELECT remainder_of_query;

UPSERT -- +CLUSTERED
  upsert_clauses
  SELECT remainder_of_query;

UPSERT /* +CLUSTERED */
  upsert_clauses
  SELECT remainder_of_query;

CREATE /* +SHUFFLE|NOSHUFFLE */
  table_clauses
  AS SELECT remainder_of_query;

CREATE -- +SHUFFLE|NOSHUFFLE
  table_clauses
  AS SELECT remainder_of_query;

CREATE /* +CLUSTERED|NOCLUSTERED */
  table_clauses
  AS SELECT remainder_of_query;

CREATE -- +CLUSTERED|NOCLUSTERED
  table_clauses
  AS SELECT remainder_of_query;
```

The square bracket style hints are supported for backward compatibility, but the syntax is deprecated and will be removed in a future release. For that reason, any newly added hints are not available with the square bracket syntax.

```
SELECT STRAIGHT_JOIN select_list FROM
join_left_hand_table
  JOIN [{ /* +BROADCAST */ | /* +SHUFFLE */ }]
join_right_hand_table
remainder_of_query;

INSERT insert_clauses
  [{ /* +SHUFFLE */ | /* +NOSHUFFLE */ }]
  [/* +CLUSTERED */]
  SELECT remainder_of_query;

UPSERT [{ /* +SHUFFLE */ | /* +NOSHUFFLE */ }]
  [/* +CLUSTERED */]
  upsert_clauses
  SELECT remainder_of_query;
```

Usage notes:

With both forms of hint syntax, include the `STRAIGHT_JOIN` keyword immediately after the `SELECT` and any `DISTINCT` or `ALL` keywords to prevent Impala from reordering the tables in a way that makes the join-related hints ineffective.

The `STRAIGHT_JOIN` hint affects the join order of table references in the query block containing the hint. It does not affect the join order of nested queries, such as views, inline views, or `WHERE`-clause subqueries. To use this hint for performance tuning of complex queries, apply the hint to all query blocks that need a fixed join order.

To reduce the need to use hints, run the `COMPUTE STATS` statement against all tables involved in joins, or used as the source tables for `INSERT ... SELECT` operations where the destination is a partitioned Parquet table. Do this operation after loading data or making substantial changes to the data within each table. Having up-to-date statistics helps Impala choose more efficient query plans without the need for hinting. See Table and Column Statistics on page 575 for details and examples.

To see which join strategy is used for a particular query, examine the `EXPLAIN` output for that query. See Using the EXPLAIN Plan for Performance Tuning on page 602 for details and examples.

Hints for join queries:

The `/* +BROADCAST */` and `/* +SHUFFLE */` hints control the execution strategy for join queries. Specify one of the following constructs immediately after the `JOIN` keyword in a query:

- `/* +SHUFFLE */` makes that join operation use the "partitioned" technique, which divides up corresponding rows from both tables using a hashing algorithm, sending subsets of the rows to other nodes for processing. (The keyword `SHUFFLE` is used to indicate a "partitioned join", because that type of join is not related to "partitioned tables".) Since the alternative "broadcast" join mechanism is the default when table and index statistics are unavailable, you might use this hint for queries where broadcast joins are unsuitable; typically, partitioned joins are more efficient for joins between large tables of similar size.
- `/* +BROADCAST */` makes that join operation use the "broadcast" technique that sends the entire contents of the right-hand table to all nodes involved in processing the join. This is the default mode of operation when table and index statistics are unavailable, so you would typically only need it if stale metadata caused Impala to mistakenly choose a partitioned join operation. Typically, broadcast joins are more efficient in cases where one table is much smaller than the other. (Put the smaller table on the right side of the `JOIN` operator.)

Hints for INSERT ... SELECT and CREATE TABLE AS SELECT (CTAS):

When inserting into partitioned tables, such as using the Parquet file format, you can include a hint in the `INSERT` or `CREATE TABLE AS SELECT(CTAS)` statements to fine-tune the overall performance of the operation and its resource usage.

You would only use hints if an `INSERT` or `CTAS` into a partitioned table was failing due to capacity limits, or if such an operation was succeeding but with less-than-optimal performance.

- `/* +SHUFFLE */` and `/* +NOSHUFFLE */` Hints
  - `/* +SHUFFLE */` adds an exchange node, before writing the data, which re-partitions the result of the `SELECT` based on the partitioning columns of the target table. With this hint, only one node writes to a partition at a time, minimizing the global number of simultaneous writes and the number of memory buffers holding data for individual partitions. This also reduces fragmentation, resulting in fewer files. Thus it reduces overall resource usage of the `INSERT` or `CTAS` operation and allows some operations to succeed that otherwise would fail. It does involve some data transfer between the nodes so that the data files for a particular partition are all written on the same node.

    Use `/* +SHUFFLE */` in cases where an `INSERT` or `CTAS` statement fails or runs inefficiently due to all nodes attempting to write data for all partitions.

    If the table is unpartitioned or every partitioning expression is constant, then `/* +SHUFFLE */` will cause every write to happen on the coordinator node.

  - `/* +NOSHUFFLE */` does not add exchange node before inserting to partitioned tables and disables re-partitioning. So the selected execution plan might be faster overall, but might also produce a larger number of small data files or exceed capacity limits, causing the `INSERT` or `CTAS` operation to fail.

    Impala automatically uses the `/* +SHUFFLE */` method if any partition key column in the source table, mentioned in the `SELECT` clause, does not have column statistics. In this case, use the `/* +NOSHUFFLE */` hint if you want to override this default behavior.

  - If column statistics are available for all partition key columns in the source table mentioned in the `INSERT ... SELECT` or `CTAS` query, Impala chooses whether to use the `/* +SHUFFLE */` or `/* +NOSHUFFLE */` technique based on the estimated number of distinct values in those columns and the number of nodes involved in the operation. In this case, you might need the `/* +SHUFFLE */` or the `/* +NOSHUFFLE */` hint to override the execution plan selected by Impala.

- `/* +CLUSTERED */` and `/* +NOCLUSTERED */` Hints
  - `/* +CLUSTERED */` sorts data by the partition columns before inserting to ensure that only one partition is written at a time per node. Use this hint to reduce the number of files kept open and the number of buffers kept in memory simultaneously. This technique is primarily useful for inserts into Parquet tables, where the large block size requires substantial memory to buffer data for multiple output files at once. This hint is available in CDH 5.10 / Impala 2.8 or higher.

Starting in CDH 6.0 / Impala 3.0, /* +CLUSTERED */ is the default behavior for HDFS tables.

– /* +NOCLUSTERED */ does not sort by primary key before insert. This hint is available in CDH 5.10 / Impala 2.8 or higher.

  Use this hint when inserting to Kudu tables.

  In the versions lower than CDH 6.0 / Impala 3.0, /* +NOCLUSTERED */ is the default in HDFS tables.

Kudu consideration:

Starting from CDH 5.12 / Impala 2.9, the INSERT or UPSERT operations into Kudu tables automatically add an exchange and a sort node to the plan that partitions and sorts the rows according to the partitioning/primary key scheme of the target table (unless the number of rows to be inserted is small enough to trigger single node execution). Since Kudu partitions and sorts rows on write, pre-partitioning and sorting takes some of the load off of Kudu and helps large INSERT operations to complete without timing out. However, this default behavior may slow down the end-to-end performance of the INSERT or UPSERT operations. Starting from CDH 5.13 / Impala 2.10, you can use the /* +NOCLUSTERED */ and /* +NOSHUFFLE */ hints together to disable partitioning and sorting before the rows are sent to Kudu. Additionally, since sorting may consume a large amount of memory, consider setting the MEM_LIMIT query option for those queries.

Hints for scheduling of scan ranges (HDFS data blocks or Kudu tablets):

The hints /* +SCHEDULE_CACHE_LOCAL */, /* +SCHEDULE_DISK_LOCAL */, and /* +SCHEDULE_REMOTE */ have the same effect as specifying the REPLICA_PREFERENCE query option with the respective option settings of CACHE_LOCAL, DISK_LOCAL, or REMOTE.

Specifying the replica preference as a query hint always overrides the query option setting.

The hint /* +RANDOM_REPLICA */ is the same as enabling the SCHEDULE_RANDOM_REPLICA query option.

You can use these hints in combination by separating them with commas, for example, /* +SCHEDULE_CACHE_LOCAL,RANDOM_REPLICA */. See REPLICA_PREFERENCE Query Option (CDH 5.9 or higher only) on page 355 and SCHEDULE_RANDOM_REPLICA Query Option (CDH 5.7 or higher only) on page 360 for information about how these settings influence the way Impala processes HDFS data blocks or Kudu tablets.

Specifying either the SCHEDULE_RANDOM_REPLICA query option or the corresponding RANDOM_REPLICA query hint enables the random tie-breaking behavior when processing data blocks during the query.

Suggestions versus directives:

In early Impala releases, hints were always obeyed and so acted more like directives. Once Impala gained join order optimizations, sometimes join queries were automatically reordered in a way that made a hint irrelevant. Therefore, the hints act more like suggestions in Impala 1.2.2 and higher.

To force Impala to follow the hinted execution mechanism for a join query, include the STRAIGHT_JOIN keyword in the SELECT statement. See Overriding Join Reordering with STRAIGHT_JOIN on page 569 for details. When you use this technique, Impala does not reorder the joined tables at all, so you must be careful to arrange the join order to put the largest table (or subquery result set) first, then the smallest, second smallest, third smallest, and so on. This ordering lets Impala do the most I/O-intensive parts of the query using local reads on the DataNodes, and then reduce the size of the intermediate result set as much as possible as each subsequent table or subquery result set is joined.

Restrictions:

Queries that include subqueries in the WHERE clause can be rewritten internally as join queries. Currently, you cannot apply hints to the joins produced by these types of queries.

Because hints can prevent queries from taking advantage of new metadata or improvements in query planning, use them only when required to work around performance issues, and be prepared to remove them when they are no longer required, such as after a new Impala release or bug fix.

In particular, the /* +BROADCAST */ and /* +SHUFFLE */ hints are expected to be needed much less frequently in Impala 1.2.2 and higher, because the join order optimization feature in combination with the COMPUTE STATS

statement now automatically choose join order and join mechanism without the need to rewrite the query and add hints. See Performance Considerations for Join Queries on page 568 for details.

Compatibility:

The hints embedded within `--` comments are compatible with Hive queries. The hints embedded within `/* */` comments or `[ ]` square brackets are not recognized by or not compatible with Hive. For example, Hive raises an error for Impala hints within `/* */` comments because it does not recognize the Impala hint names.

Considerations for views:

If you use a hint in the query that defines a view, the hint is preserved when you query the view. Impala internally rewrites all hints in views to use the `--` comment notation, so that Hive can query such views without errors due to unrecognized hint names.

Examples:

For example, this query joins a large customer table with a small lookup table of less than 100 rows. The right-hand table can be broadcast efficiently to all nodes involved in the join. Thus, you would use the `/* +broadcast */` hint to force a broadcast join strategy:

```
select straight_join customer.address, state_lookup.state_name
  from customer join /* +broadcast */ state_lookup
  on customer.state_id = state_lookup.state_id;
```

This query joins two large tables of unpredictable size. You might benchmark the query with both kinds of hints and find that it is more efficient to transmit portions of each table to other nodes for processing. Thus, you would use the `/* +shuffle */` hint to force a partitioned join strategy:

```
select straight_join weather.wind_velocity, geospatial.altitude
  from weather join /* +shuffle */ geospatial
  on weather.lat = geospatial.lat and weather.long = geospatial.long;
```

For joins involving three or more tables, the hint applies to the tables on either side of that specific `JOIN` keyword. The `STRAIGHT_JOIN` keyword ensures that joins are processed in a predictable order from left to right. For example, this query joins `t1` and `t2` using a partitioned join, then joins that result set to `t3` using a broadcast join:

```
select straight_join t1.name, t2.id, t3.price
  from t1 join /* +shuffle */ t2 join /* +broadcast */ t3
  on t1.id = t2.id and t2.id = t3.id;
```

Related information:

For more background information about join queries, see Joins in Impala SELECT Statements on page 296. For performance considerations, see Performance Considerations for Join Queries on page 568.

Impala Built-In Functions

Impala supports several categories of built-in functions. These functions let you perform mathematical calculations, string manipulation, date calculations, and other kinds of data transformations directly in SQL statements.

The categories of built-in functions supported by Impala are:

- Impala Mathematical Functions on page 397
- Impala Type Conversion Functions on page 423
- Impala Date and Time Functions on page 424
- Impala Conditional Functions on page 457
- Impala String Functions on page 462
- Impala Aggregate Functions on page 479.
- Impala Analytic Functions on page 506
- Impala Bit Functions on page 414

- Impala Miscellaneous Functions on page 477

The following is a complete list of built-in functions supported in Impala:

ABS
ACOS
ADD_MONTHS
ADDDATE
APPX_MEDIAN
ASCII
ASIN
ATAN
ATAN2
AVG
AVG - Analytic Function
BASE64DECODE
BASE64ENCODE
BITAND
BIN
BITNOT
BITOR
BITXOR
BTRIM
CASE
CASE WHEN
CAST
CEIL, CEILING, DCEIL
CHAR_LENGTH
CHR
COALESCE
CONCAT
CONCAT_WS
CONV
COS
COSH
COT
COUNT
COUNT - Analytic Function

COUNTSET
CUME_DIST
CURRENT_DATABASE
CURRENT_TIMESTAMP
DATE_ADD
DATE_PART
DATE_SUB
DATE_TRUNC
DATEDIFF
DAY
DAYNAME
DAYOFWEEK
DAYOFYEAR
DAYS_ADD
DAYS_SUB
DECODE
DEGREES
DENSE_RANK
E
EFFECTIVE_USER
EXP
EXTRACT
FACTORIAL
FIND_IN_SET
FIRST_VALUE
FLOOR, DFLOOR
FMOD
FNV_HASH
GET_JSON_OBJECT
FROM_UNIXTIME
FROM_TIMESTAMP
FROM_UTC_TIMESTAMP
GETBIT
GREATEST
GROUP_CONCAT
GROUP_CONCAT - Analytic Function

HEX
HOUR
HOURS_ADD
HOURS_SUB
IF
IFNULL
INITCAP
INSTR
INT_MONTHS_BETWEEN
IS_INF
IS_NAN
ISFALSE
ISNOTFALSE
ISNOTTRUE
ISNULL
ISTRUE
LAG
LAST_VALUE
LEAD
LEAST
LEFT
LENGTH
LN
LOCATE
LOG
LOG10
LOG2
LOWER, LCASE
LPAD
LTRIM
MAX
MAX - Analytic Function
MAX_INT, MAX_TINYINT, MAX_SMALLINT, MAX_BIGINT
MICROSECONDS_ADD
MICROSECONDS_SUB
MILLISECOND

MILLISECONDS_ADD
MILLISECONDS_SUB
MIN
MIN - Analytic Function
MIN_INT, MIN_TINYINT, MIN_SMALLINT, MIN_BIGINT
MINUTE
MINUTES_ADD
MINUTES_SUB
MOD
MONTH
MONTHNAME
MONTHS_ADD
MONTHS_BETWEEN
MONTHS_SUB
MURMUR_HASH
NANOSECONDS_ADD
NANOSECONDS_SUB
NDV
NEGATIVE
NEXT_DAY
NONNULLVALUE
NOW
NTILE
NULLIF
NULLIFZERO
NULLVALUE
NVL
NVL2
OVER Clause
PARSE_URL
PERCENT_RANK
PI
PID
PMOD
POSITIVE
POW, POWER, DPOW, FPOW

PRECISION
QUARTER
QUOTIENT
RADIANS
RAND, RANDOM
RANK
REGEXP_ESCAPE
REGEXP_EXTRACT
REGEXP_LIKE
REGEXP_REPLACE
REPEAT
REPLACE
REVERSE
RIGHT
ROTATELEFT
ROTATERIGHT
ROUND, DROUND
ROW_NUMBER
RPAD
RTRIM
SCALE
SECOND
SECONDS_ADD
SECONDS_SUB
SETBIT
SHIFTLEFT
SHIFTRIGHT
SIGN
SIN
SINH
SLEEP
SPACE
SPLIT_PART
SQRT
STDDEV, STDDEV_SAMP, STDDEV_POP
STRLEFT

STRRIGHT
SUBDATE
SUBSTR, SUBSTRING
SUM
SUM - Analytic Function
TAN
TANH
TIMEOFDAY
TIMESTAMP_CMP
TO_DATE
TO_TIMESTAMP
TO_UTC_TIMESTAMP
TRANSLATE
TRIM
TRUNC
TRUNCATE, DTRUNC, TRUNC
TYPEOF
UNHEX
UNIX_TIMESTAMP
UPPER, UCASE
USER
UTC_TIMESTAMP
UUID
VARIANCE, VARIANCE_SAMP, VARIANCE_POP, VAR_SAMP, VAR_POP
VERSION
WEEKOFYEAR
WEEKS_ADD
WEEKS_SUB
WIDTH_BUCKET
YEAR
YEARS_ADD
YEARS_SUB
ZEROIFNULL

Impala Mathematical Functions

Mathematical functions, or arithmetic functions, perform numeric calculations that are typically more complex than basic addition, subtraction, multiplication, and division. For example, these functions include trigonometric, logarithmic, and base conversion operations.



Note: In Impala, exponentiation uses the `POW()` function rather than an exponentiation operator such as `**`.

Related information:

The mathematical functions operate mainly on these data types: INT Data Type on page 117, BIGINT Data Type on page 105, SMALLINT Data Type on page 122, TINYINT Data Type on page 136, DOUBLE Data Type on page 114, FLOAT Data Type on page 116, and DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only) on page 109. For the operators that perform the standard operations such as addition, subtraction, multiplication, and division, see Arithmetic Operators on page 171.

Functions that perform bitwise operations are explained in Impala Bit Functions on page 414.

Function reference:

Impala supports the following mathematical functions:

- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- BIN
- CEIL, CEILING, DCEIL
- CONV
- COS
- COSH
- COT
- DEGREES
- E
- EXP
- FACTORIAL
- FLOOR, DFLOOR
- FMOD
- FNV_HASH
- GREATEST
- HEX
- IS_INF
- IS_NAN
- LEAST
- LN
- LOG
- LOG10
- LOG2
- MAX_INT, MAX_TINYINT, MAX_SMALLINT, MAX_BIGINT
- MIN_INT, MIN_TINYINT, MIN_SMALLINT, MIN_BIGINT
- MOD
- MURMUR_HASH
- NEGATIVE
- PI
- PMOD
- POSITIVE
- POW, POWER, DPOW, FPOW

- PRECISION
- QUOTIENT
- RADIANS
- RAND, RANDOM
- ROUND, DROUND
- SCALE
- SIGN
- SIN
- SINH
- SQRT
- TAN
- TANH
- TRUNCATE, DTRUNC, TRUNC
- UNHEX
- WIDTH_BUCKET

ABS(numeric_type a)

Purpose: Returns the absolute value of the argument.

Return type: Same as the input value

Usage notes: Use this function to ensure all return values are positive. This is different than the `positive()` function, which returns its argument unchanged (even if the argument was negative).

ACOS(DOUBLE a)

Purpose: Returns the arccosine of the argument.

Return type: `DOUBLE`

ASIN(DOUBLE a)

Purpose: Returns the arcsine of the argument.

Return type: `DOUBLE`

ATAN(DOUBLE a)

Purpose: Returns the arctangent of the argument.

Return type: `DOUBLE`

ATAN(DOUBLE a, DOUBLE b)

Purpose: Returns the arctangent of the two arguments, with the signs of the arguments used to determine the quadrant of the result.

Return type: `DOUBLE`

BIN(BIGINT a)

Purpose: Returns the binary representation of an integer value, that is, a string of 0 and 1 digits.

Return type: `STRING`

CEIL(DOUBLE a), CEIL(DECIMAL(p,s) a), CEILING(DOUBLE a), CEILING(DECIMAL(p,s) a), DCEIL(DOUBLE a), DCEIL(DECIMAL(p,s) a)

Purpose: Returns the smallest integer that is greater than or equal to the argument.

Return type: Same as the input type

CONV(BIGINT n, INT from_base, INT to_base), CONV(STRING s, INT from_base, INT to_base)

Purpose: Returns a string representation of the first argument converted from `from_base` to `to_base`. The first argument can be specified as a number or a string. For example, `conv(100, 2, 10)` and `conv('100', 2, 10)` both return `'4'`.

Return type: STRING

Usage notes:

If to_base is negative, the first argument is treated as signed, and otherwise, it is treated as unsigned. For example:

- conv(-17, 10, -2) returns '-10001', -17 in base 2.
- conv(-17, 10, 10) returns '18446744073709551599'. -17 is interpreted as an unsigned, 2^64-17, and then the value is returned in base 10.

The function returns NULL when the following illegal arguments are specified:

- Any argument is NULL.
- from_base or to_base is below -36 or above 36.
- from_base or to_base is -1, 0, or 1.
- The first argument represents a positive number and from_base is a negative number.

If the first argument represents a negative number and from_base is a negative number, the function returns 0.

If the first argument represents a number larger than the maximum bigint, the function returns:

- The string representation of -1 in to_base if to_base is negative.
- The string representation of 18446744073709551615' (2^64 - 1) in to_base if to_base is positive.

If the first argument does not represent a valid number in from_base, e.g. 3 in base 2 or '1a23' in base 10, the digits in the first argument are evaluated from left-to-right and used if a valid digit in from_base. The invalid digit and the digits to the right are ignored.

For example:

- conv(445, 5, 10) is converted to conv(44, 5, 10) and returns '24'.
- conv('1a23', 10, 16) is converted to conv('1', 10 , 16) and returns '1'.

COS(DOUBLE a)

Purpose: Returns the cosine of the argument.

Return type: DOUBLE

COSH(DOUBLE a)

Purpose: Returns the hyperbolic cosine of the argument.

Return type: DOUBLE

COT(DOUBLE a)

Purpose: Returns the cotangent of the argument.

Return type: DOUBLE

Added in: CDH 5.5.0 / Impala 2.3.0

DEGREES(DOUBLE a)

Purpose: Converts argument value from radians to degrees.

Return type: DOUBLE

E()

Purpose: Returns the mathematical constant e.

Return type: DOUBLE

EXP(DOUBLE a), DEXP(DOUBLE a)

Purpose: Returns the mathematical constant e raised to the power of the argument.

Return type: DOUBLE

FACTORIAL(integer_type a)

Purpose: Computes the factorial of an integer value. It works with any integer type.

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes: You can use either the `factorial()` function or the `!` operator. The factorial of 0 is 1. Likewise, the `factorial()` function returns 1 for any negative value. The maximum positive value for the input argument is 20; a value of 21 or greater overflows the range for a `BIGINT` and causes an error.

Return type: `BIGINT`

Added in: CDH 5.5.0 / Impala 2.3.0

```
select factorial(5);
+--------------+
| factorial(5) |
+--------------+
| 120          |
+--------------+

select 5!;
+-----+
| 5!  |
+-----+
| 120 |
+-----+
```

FLOOR(DOUBLE a), FLOOR(DECIMAL(p,s) a), DFLOOR(DOUBLE a), DFLOOR(DECIMAL(p,s) a)

Purpose: Returns the largest integer that is less than or equal to the argument.

Return type: Same as the input type

FMOD(DOUBLE a, DOUBLE b), FMOD(FLOAT a, FLOAT b)

Purpose: Returns the modulus of a floating-point number.

Return type: `FLOAT` or `DOUBLE`, depending on type of arguments

Added in: Impala 1.1.1

Usage notes:

Because this function operates on `DOUBLE` or `FLOAT` values, it is subject to potential rounding errors for values that cannot be represented precisely. Prefer to use whole numbers, or values that you know can be represented precisely by the `DOUBLE` or `FLOAT` types.

Examples:

The following examples show equivalent operations with the `fmod()` function and the `%` arithmetic operator, for values not subject to any rounding error.

```
select fmod(10,3);
+-------------+
| fmod(10, 3) |
+-------------+
| 1           |
+-------------+

select fmod(5.5,2);
+--------------+
| fmod(5.5, 2) |
+--------------+
| 1.5          |
+--------------+

select 10 % 3;
+--------+
| 10 % 3 |
+--------+
```

```
| 1      |
+--------+

select 5.5 % 2;
+---------+
| 5.5 % 2 |
+---------+
| 1.5     |
+---------+
```

The following examples show operations with the `fmod()` function for values that cannot be represented precisely by the `DOUBLE` or `FLOAT` types, and thus are subject to rounding error. `fmod(9.9,3.0)` returns a value slightly different than the expected 0.9 because of rounding. `fmod(9.9,3.3)` returns a value quite different from the expected value of 0 because of rounding error during intermediate calculations.

```
select fmod(9.9,3.0);
+--------------------+
| fmod(9.9, 3.0)     |
+--------------------+
| 0.8999996185302734 |
+--------------------+

select fmod(9.9,3.3);
+-------------------+
| fmod(9.9, 3.3)    |
+-------------------+
| 3.299999713897705 |
+-------------------+
```

FNV_HASH(type v)

Purpose: Returns a consistent 64-bit value derived from the input argument, for convenience of implementing hashing logic in an application.

Return type: `BIGINT`

Usage notes:

You might use the return value in an application where you perform load balancing, bucketing, or some other technique to divide processing or storage.

Because the result can be any 64-bit value, to restrict the value to a particular range, you can use an expression that includes the `ABS()` function and the `%` (modulo) operator. For example, to produce a hash value in the range 0-9, you could use the expression `ABS(FNV_HASH(x)) % 10`.

This function implements the same algorithm that Impala uses internally for hashing, on systems where the CRC32 instructions are not available.

This function implements the Fowler–Noll–Vo hash function, in particular the FNV-1a variation. This is not a perfect hash function: some combinations of values could produce the same result value. It is not suitable for cryptographic use.

Similar input values of different types could produce different hash values, for example the same numeric value represented as `SMALLINT` or `BIGINT`, `FLOAT` or `DOUBLE`, or `DECIMAL(5,2)` or `DECIMAL(20,5)`.

Examples:

```
[localhost:21000] > create table h (x int, s string);
[localhost:21000] > insert into h values (0, 'hello'), (1,'world'),
(1234567890,'antidisestablishmentarianism');
[localhost:21000] > select x, fnv_hash(x) from h;
+------------+---------------------+
| x          | fnv_hash(x)         |
+------------+---------------------+
| 0          | -2611523532599129963 |
| 1          | 4307505193096137732  |
| 1234567890 | 3614724209955230832  |
+------------+---------------------+
```

```
[localhost:21000] > select s, fnv_hash(s) from h;
+------------------------------+---------------------+
| s                            | fnv_hash(s)         |
+------------------------------+---------------------+
| hello                        | 6414202926103426347 |
| world                        | 6535280128821139475 |
| antidisestablishmentarianism | -209330013948433970 |
+------------------------------+---------------------+
[localhost:21000] > select s, abs(fnv_hash(s)) % 10 from h;
+------------------------------+------------------------+
| s                            | abs(fnv_hash(s)) % 10.0 |
+------------------------------+------------------------+
| hello                        | 8                      |
| world                        | 6                      |
| antidisestablishmentarianism | 4                      |
+------------------------------+------------------------+
```

For short argument values, the high-order bits of the result have relatively low entropy:

```
[localhost:21000] > create table b (x boolean);
[localhost:21000] > insert into b values (true), (true), (false), (false);
[localhost:21000] > select x, fnv_hash(x) from b;
+-------+---------------------+
| x     | fnv_hash(x)         |
+-------+---------------------+
| true  | 2062020650953872396 |
| true  | 2062020650953872396 |
| false | 2062021750465500607 |
| false | 2062021750465500607 |
+-------+---------------------+
```

Added in: Impala 1.2.2

GREATEST(BIGINT a[, BIGINT b ...]), GREATEST(DOUBLE a[, DOUBLE b ...]), GREATEST(DECIMAL(p,s) a[, DECIMAL(p,s) b ...]), GREATEST(STRING a[, STRING b ...]), GREATEST(TIMESTAMP a[, TIMESTAMP b ...])

Purpose: Returns the largest value from a list of expressions.

Return type: same as the initial argument value, except that integer values are promoted to `BIGINT` and floating-point values are promoted to `DOUBLE`; use `CAST()` when inserting into a smaller numeric column

HEX(BIGINT a), HEX(STRING a)

Purpose: Returns the hexadecimal representation of an integer value, or of the characters in a string.

Return type: `STRING`

IS_INF(DOUBLE a)

Purpose: Tests whether a value is equal to the special value “inf”, signifying infinity.

Return type: `BOOLEAN`

Usage notes:

Infinity and NaN can be specified in text data files as `inf` and `nan` respectively, and Impala interprets them as these special values. They can also be produced by certain arithmetic expressions; for example, `1/0` returns `Infinity` and `pow(-1, 0.5)` returns `NaN`. Or you can cast the literal values, such as `CAST('nan' AS DOUBLE)` or `CAST('inf' AS DOUBLE)`.

IS_NAN(DOUBLE a)

Purpose: Tests whether a value is equal to the special value “NaN”, signifying “not a number”.

Return type: `BOOLEAN`

Usage notes:

Infinity and NaN can be specified in text data files as `inf` and `nan` respectively, and Impala interprets them as these special values. They can also be produced by certain arithmetic expressions; for example, `1/0` returns `Infinity`

and `pow(-1, 0.5)` returns `NaN`. Or you can cast the literal values, such as `CAST('nan' AS DOUBLE)` or `CAST('inf' AS DOUBLE)`.

LEAST(BIGINT a[, BIGINT b ...]), LEAST(DOUBLE a[, DOUBLE b ...]), LEAST(DECIMAL(p,s) a[, DECIMAL(p,s) b ...]), LEAST(STRING a[, STRING b ...]), LEAST(TIMESTAMP a[, TIMESTAMP b ...])

Purpose: Returns the smallest value from a list of expressions.

Return type: same as the initial argument value, except that integer values are promoted to `BIGINT` and floating-point values are promoted to `DOUBLE`; use `CAST()` when inserting into a smaller numeric column

LN(DOUBLE a), DLOG1(DOUBLE a)

Purpose: Returns the natural logarithm of the argument.

Return type: `DOUBLE`

LOG(DOUBLE base, DOUBLE a)

Purpose: Returns the logarithm of the second argument to the specified base.

Return type: `DOUBLE`

LOG10(DOUBLE a), DLOG10(DOUBLE a)

Purpose: Returns the logarithm of the argument to the base 10.

Return type: `DOUBLE`

LOG2(DOUBLE a)

Purpose: Returns the logarithm of the argument to the base 2.

Return type: `DOUBLE`

MAX_INT(), MAX_TINYINT(), MAX_SMALLINT(), MAX_BIGINT()

Purpose: Returns the largest value of the associated integral type.

Return type: The same as the integral type being checked.

Usage notes: Use the corresponding `min_` and `max_` functions to check if all values in a column are within the allowed range, before copying data or altering column definitions. If not, switch to the next higher integral type or to a `DECIMAL` with sufficient precision.

MIN_INT(), MIN_TINYINT(), MIN_SMALLINT(), MIN_BIGINT()

Purpose: Returns the smallest value of the associated integral type (a negative number).

Return type: The same as the integral type being checked.

Usage notes: Use the corresponding `min_` and `max_` functions to check if all values in a column are within the allowed range, before copying data or altering column definitions. If not, switch to the next higher integral type or to a `DECIMAL` with sufficient precision.

MOD(numeric_type a, same_type b)

Purpose: Returns the modulus of a number. Equivalent to the `%` arithmetic operator. Works with any size integer type, any size floating-point type, and `DECIMAL` with any precision and scale.

Return type: Same as the input value

Added in: CDH 5.4.0 / Impala 2.2.0

Usage notes:

Because this function works with `DECIMAL` values, prefer it over `fmod()` when working with fractional values. It is not subject to the rounding errors that make `fmod()` problematic with floating-point numbers.

Query plans shows the `MOD()` function as the `%` operator.

Examples:

The following examples show how the `mod()` function works for whole numbers and fractional values, and how the `%` operator works the same way. In the case of `mod(9.9,3)`, the type conversion for the second argument results in the first argument being interpreted as `DOUBLE`, so to produce an accurate `DECIMAL` result requires casting the second argument or writing it as a `DECIMAL` literal, 3.0.

```
select mod(10,3);
+-------------+
| mod(10, 3) |
+-------------+
| 1           |
+-------------+

select mod(5.5,2);
+--------------+
| mod(5.5, 2) |
+--------------+
| 1.5          |
+--------------+

select 10 % 3;
+--------+
| 10 % 3 |
+--------+
| 1      |
+--------+

select 5.5 % 2;
+---------+
| 5.5 % 2 |
+---------+
| 1.5     |
+---------+

select mod(9.9,3.3);
+---------------+
| mod(9.9, 3.3) |
+---------------+
| 0.0           |
+---------------+

select mod(9.9,3);
+--------------------+
| mod(9.9, 3)        |
+--------------------+
| 0.8999996185302734 |
+--------------------+

select mod(9.9, cast(3 as decimal(2,1)));
+-----------------------------------+
| mod(9.9, cast(3 as decimal(2,1))) |
+-----------------------------------+
| 0.9                               |
+-----------------------------------+

select mod(9.9,3.0);
+---------------+
| mod(9.9, 3.0) |
+---------------+
| 0.9           |
+---------------+
```

MURMUR_HASH(type v)

Purpose: Returns a consistent 64-bit value derived from the input argument, for convenience of implementing MurmurHash2 non-cryptographic hash function.

Return type: `BIGINT`

Usage notes:

You might use the return value in an application where you perform load balancing, bucketing, or some other technique to divide processing or storage. This function provides a good performance for all kinds of keys such as number, ascii string and UTF-8. It can be recommended as general-purpose hashing function.

Regarding comparison of murmur_hash with fnv_hash, murmur_hash is based on Murmur2 hash algorithm and fnv_hash function is based on FNV-1a hash algorithm. Murmur2 and FNV-1a can show very good randomness and performance compared with well known other hash algorithms, but Murmur2 slightly show better randomness and performance than FNV-1a. See [1][2][3] for details.

Similar input values of different types could produce different hash values, for example the same numeric value represented as `SMALLINT` or `BIGINT`, `FLOAT` or `DOUBLE`, or `DECIMAL(5,2)` or `DECIMAL(20,5)`.

Examples:

```
[localhost:21000] > create table h (x int, s string);
[localhost:21000] > insert into h values (0, 'hello'), (1,'world'),
(1234567890,'antidisestablishmentarianism');
[localhost:21000] > select x, murmur_hash(x) from h;
+------------+----------------------+
| x          | murmur_hash(x)       |
+------------+----------------------+
| 0          | 6960269033020761575  |
| 1          | -780611581681153783  |
| 1234567890 | -5754914572385924334 |
+------------+----------------------+
[localhost:21000] > select s, murmur_hash(s) from h;
+-------------------------------+----------------------+
| s                             | murmur_hash(s)       |
+-------------------------------+----------------------+
| hello                         | 2191231550387646743  |
| world                         | 5568329560871645431  |
| antidisestablishmentarianism | -2261804666958489663 |
+-------------------------------+----------------------+
```

For short argument values, the high-order bits of the result have relatively higher entropy than fnv_hash:

```
[localhost:21000] > create table b (x boolean);
[localhost:21000] > insert into b values (true), (true), (false), (false);
[localhost:21000] > select x, murmur_hash(x) from b;
+-------+----------------------+
| x     | murmur_hash(x)       |
+-------+----------------------++
| true  | -5720937396023583481 |
| true  | -5720937396023583481 |
| false | 6351753276682545529  |
| false | 6351753276682545529  |
+-------+--------------------+-+
```

Added in: Impala 2.12.0

NEGATIVE(numeric_type a)

Purpose: Returns the argument with the sign reversed; returns a positive value if the argument was already negative.

Return type: Same as the input value

Usage notes: Use `-abs(a)` instead if you need to ensure all return values are negative.

PI()

Purpose: Returns the constant pi.

Return type: `double`

PMOD(BIGINT a, BIGINT b), PMOD(DOUBLE a, DOUBLE b)

Purpose: Returns the positive modulus of a number. Primarily for HiveQL compatibility.

Return type: `INT` or `DOUBLE`, depending on type of arguments

Examples:

The following examples show how the `fmod()` function sometimes returns a negative value depending on the sign of its arguments, and the `pmod()` function returns the same value as `fmod()`, but sometimes with the sign flipped.

```
select fmod(-5,2);
+-------------+
| fmod(-5, 2) |
+-------------+
| -1          |
+-------------+

select pmod(-5,2);
+-------------+
| pmod(-5, 2) |
+-------------+
| 1           |
+-------------+

select fmod(-5,-2);
+--------------+
| fmod(-5, -2) |
+--------------+
| -1           |
+--------------+

select pmod(-5,-2);
+--------------+
| pmod(-5, -2) |
+--------------+
| -1           |
+--------------+

select fmod(5,-2);
+-------------+
| fmod(5, -2) |
+-------------+
| 1           |
+-------------+

select pmod(5,-2);
+-------------+
| pmod(5, -2) |
+-------------+
| -1          |
+-------------+
```

POSITIVE(numeric_type a)

Purpose: Returns the original argument unchanged (even if the argument is negative).

Return type: Same as the input value

Usage notes: Use `abs()` instead if you need to ensure all return values are positive.

POW(DOUBLE a, double p), POWER(DOUBLE a, DOUBLE p), DPOW(DOUBLE a, DOUBLE p), FPOW(DOUBLE a, DOUBLE p)

Purpose: Returns the first argument raised to the power of the second argument.

Return type: `DOUBLE`

PRECISION(numeric_expression)

Purpose: Computes the precision (number of decimal digits) needed to represent the type of the argument expression as a `DECIMAL` value.

Usage notes:

Typically used in combination with the `scale()` function, to determine the appropriate `DECIMAL(`*`precision`*`,`*`scale`*`)` type to declare in a `CREATE TABLE` statement or `CAST()` function.

Return type: `INT`

Examples:

The following examples demonstrate how to check the precision and scale of numeric literals or other numeric expressions. Impala represents numeric literals in the smallest appropriate type. 5 is a TINYINT value, which ranges from -128 to 127, therefore 3 decimal digits are needed to represent the entire range, and because it is an integer value there are no fractional digits. 1.333 is interpreted as a DECIMAL value, with 4 digits total and 3 digits after the decimal point.

```
[localhost:21000] > select precision(5), scale(5);
+--------------+----------+
| precision(5) | scale(5) |
+--------------+----------+
| 3            | 0        |
+--------------+----------+
[localhost:21000] > select precision(1.333), scale(1.333);
+------------------+--------------+
| precision(1.333) | scale(1.333) |
+------------------+--------------+
| 4                | 3            |
+------------------+--------------+
[localhost:21000] > with t1 as
  ( select cast(12.34 as decimal(20,2)) x union select cast(1 as decimal(8,6)) x )
  select precision(x), scale(x) from t1 limit 1;
+--------------+----------+
| precision(x) | scale(x) |
+--------------+----------+
| 24           | 6        |
+--------------+----------+
```

QUOTIENT(BIGINT numerator, BIGINT denominator), QUOTIENT(DOUBLE numerator, DOUBLE denominator)

Purpose: Returns the first argument divided by the second argument, discarding any fractional part. Avoids promoting integer arguments to DOUBLE as happens with the / SQL operator. Also includes an overload that accepts DOUBLE arguments, discards the fractional part of each argument value before dividing, and again returns BIGINT. With integer arguments, this function works the same as the DIV operator.

Return type: BIGINT

RADIANS(DOUBLE a)

Purpose: Converts argument value from degrees to radians.

Return type: DOUBLE

RAND(), RAND(BIGINT seed), RANDOM(), RANDOM(BIGINT seed)

Purpose: Returns a random value between 0 and 1. After `rand()` is called with a seed argument, it produces a consistent random sequence based on the seed value.

Return type: DOUBLE

Usage notes: Currently, the random sequence is reset after each query, and multiple calls to `rand()` within the same query return the same value each time. For different number sequences that are different for each query, pass a unique seed value to each call to `rand()`. For example, `select rand(unix_timestamp()) from ...`

Examples:

The following examples show how `rand()` can produce sequences of varying predictability, so that you can reproduce query results involving random values or generate unique sequences of random values for each query. When `rand()` is called with no argument, it generates the same sequence of values each time, regardless of the ordering of the result set. When `rand()` is called with a constant integer, it generates a different sequence of values, but still always the same sequence for the same seed value. If you pass in a seed value that changes, such as the return value of the expression `unix_timestamp(now())`, each query will use a different sequence of random values, potentially more useful in probability calculations although more difficult to reproduce at a later time. Therefore, the final two examples with an unpredictable seed value also include the seed in the result set, to make it possible to reproduce the same random sequence later.

```
select x, rand() from three_rows;
+---+-----------------------+
```

```
| x | rand()                |
+---+-----------------------+
| 1 | 0.0004714746030380365 |
| 2 | 0.5895895192351144    |
| 3 | 0.4431900859080209    |
+---+-----------------------+

select x, rand() from three_rows order by x desc;
+---+-----------------------+
| x | rand()                |
+---+-----------------------+
| 3 | 0.0004714746030380365 |
| 2 | 0.5895895192351144    |
| 1 | 0.4431900859080209    |
+---+-----------------------+

select x, rand(1234) from three_rows order by x;
+---+----------------------+
| x | rand(1234)           |
+---+----------------------+
| 1 | 0.7377511392057646   |
| 2 | 0.009428468537250751 |
| 3 | 0.208117277924026    |
+---+----------------------+

select x, rand(1234) from three_rows order by x desc;
+---+----------------------+
| x | rand(1234)           |
+---+----------------------+
| 3 | 0.7377511392057646   |
| 2 | 0.009428468537250751 |
| 1 | 0.208117277924026    |
+---+----------------------+

select x, unix_timestamp(now()), rand(unix_timestamp(now()))
  from three_rows order by x;
+---+-----------------------+----------------------------+
| x | unix_timestamp(now()) | rand(unix_timestamp(now())) |
+---+-----------------------+----------------------------+
| 1 | 1440777752            | 0.002051228658320023       |
| 2 | 1440777752            | 0.5098743483004506         |
| 3 | 1440777752            | 0.9517714925817081         |
+---+-----------------------+----------------------------+

select x, unix_timestamp(now()), rand(unix_timestamp(now()))
  from three_rows order by x desc;
+---+-----------------------+----------------------------+
| x | unix_timestamp(now()) | rand(unix_timestamp(now())) |
+---+-----------------------+----------------------------+
| 3 | 1440777761            | 0.9985985015512437         |
| 2 | 1440777761            | 0.3251255333074953         |
| 1 | 1440777761            | 0.02422675025846192        |
+---+-----------------------+----------------------------+
```

ROUND(DOUBLE a), ROUND(DOUBLE a, INT d), ROUND(DECIMAL a, int_type d), DROUND(DOUBLE a), DROUND(DOUBLE a, INT d), DROUND(DECIMAL(p,s) a, int_type d)

Purpose: Rounds a floating-point value. By default (with a single argument), rounds to the nearest integer. Values ending in .5 are rounded up for positive numbers, down for negative numbers (that is, away from zero). The optional second argument specifies how many digits to leave after the decimal point; values greater than zero produce a floating-point return value rounded to the requested number of digits to the right of the decimal point.

Return type: Same as the input type

SCALE(numeric_expression)

Purpose: Computes the scale (number of decimal digits to the right of the decimal point) needed to represent the type of the argument expression as a `DECIMAL` value.

Usage notes:

Typically used in combination with the `precision()` function, to determine the appropriate `DECIMAL(`*`precision`*`,`*`scale`*`)` type to declare in a `CREATE TABLE` statement or `CAST()` function.

Return type: `int`

Examples:

The following examples demonstrate how to check the precision and scale of numeric literals or other numeric expressions. Impala represents numeric literals in the smallest appropriate type. 5 is a `TINYINT` value, which ranges from -128 to 127, therefore 3 decimal digits are needed to represent the entire range, and because it is an integer value there are no fractional digits. 1.333 is interpreted as a `DECIMAL` value, with 4 digits total and 3 digits after the decimal point.

```
[localhost:21000] > select precision(5), scale(5);
+--------------+----------+
| precision(5) | scale(5) |
+--------------+----------+
| 3            | 0        |
+--------------+----------+
[localhost:21000] > select precision(1.333), scale(1.333);
+------------------+--------------+
| precision(1.333) | scale(1.333) |
+------------------+--------------+
| 4                | 3            |
+------------------+--------------+
[localhost:21000] > with t1 as
  ( select cast(12.34 as decimal(20,2)) x union select cast(1 as decimal(8,6)) x )
  select precision(x), scale(x) from t1 limit 1;
+--------------+----------+
| precision(x) | scale(x) |
+--------------+----------+
| 24           | 6        |
+--------------+----------+
```

SIGN(DOUBLE a)

Purpose: Returns -1, 0, or 1 to indicate the signedness of the argument value.

Return type: `INT`

SIN(DOUBLE a)

Purpose: Returns the sine of the argument.

Return type: `DOUBLE`

SINH(DOUBLE a)

Purpose: Returns the hyperbolic sine of the argument.

Return type: `DOUBLE`

SQRT(DOUBLE a), DSQRT(DOUBLE a)

Purpose: Returns the square root of the argument.

Return type: `DOUBLE`

TAN(DOUBLE a)

Purpose: Returns the tangent of the argument.

Return type: `DOUBLE`

TANH(DOUBLE a)

Purpose: Returns the hyperbolic tangent of the argument.

Return type: `DOUBLE`

TRUNCATE(DOUBLE_or_DECIMAL a[, digits_to_leave]), DTRUNC(DOUBLE_or_DECIMAL a[, digits_to_leave]), TRUNC(DOUBLE_or_DECIMAL a[, digits_to_leave])

Purpose: Removes some or all fractional digits from a numeric value.

Arguments: With a single floating-point argument, removes all fractional digits, leaving an integer value. The optional second argument specifies the number of fractional digits to include in the return value, and only applies when the argument type is `DECIMAL`. A second argument of 0 truncates to a whole integer value. A second argument of negative N sets N digits to 0 on the left side of the decimal

Scale argument: The scale argument applies only when truncating `DECIMAL` values. It is an integer specifying how many significant digits to leave to the right of the decimal point. A scale argument of 0 truncates to a whole integer value. A scale argument of negative N sets N digits to 0 on the left side of the decimal point.

`TRUNCATE()`, `DTRUNC()`, and `TRUNC()` are aliases for the same function.

Return type: Same as the input type

Added in: The `TRUNC()` alias was added in CDH 5.13 / Impala 2.10.

Usage notes:

You can also pass a `DOUBLE` argument, or `DECIMAL` argument with optional scale, to the `DTRUNC()` or `TRUNCATE` functions. Using the `TRUNC()` function for numeric values is common with other industry-standard database systems, so you might find such `TRUNC()` calls in code that you are porting to Impala.

The `TRUNC()` function also has a signature that applies to `TIMESTAMP` values. See Impala Date and Time Functions for details.

Examples:

The following examples demonstrate the `TRUNCATE()` and `DTRUNC()` signatures for this function:

```
select truncate(3.45);
+----------------+
| truncate(3.45) |
+----------------+
| 3              |
+----------------+

select truncate(-3.45);
+-----------------+
| truncate(-3.45) |
+-----------------+
| -3              |
+-----------------+

select truncate(3.456,1);
+--------------------+
| truncate(3.456, 1) |
+--------------------+
| 3.4                |
+--------------------+

select dtrunc(3.456,1);
+------------------+
| dtrunc(3.456, 1) |
+------------------+
| 3.4              |
+------------------+

select truncate(3.456,2);
+--------------------+
| truncate(3.456, 2) |
+--------------------+
| 3.45               |
+--------------------+

select truncate(3.456,7);
+--------------------+
| truncate(3.456, 7) |
+--------------------+
| 3.4560000          |
+--------------------+
```

The following examples demonstrate using TRUNC( ) with DECIMAL or DOUBLE values, and with an optional scale argument for DECIMAL values. (The behavior is the same for the TRUNCATE( ) and DTRUNC( ) aliases also.)

```
create table t1 (d decimal(20,7));

-- By default, no digits to the right of the decimal point.
insert into t1 values (1.1), (2.22), (3.333), (4.4444), (5.55555);
select trunc(d) from t1 order by d;
+----------+
| trunc(d) |
+----------+
| 1        |
| 2        |
| 3        |
| 4        |
| 5        |
+----------+

-- 1 digit to the right of the decimal point.
select trunc(d,1) from t1 order by d;
+-------------+
| trunc(d, 1) |
+-------------+
| 1.1         |
| 2.2         |
| 3.3         |
| 4.4         |
| 5.5         |
+-------------+

-- 2 digits to the right of the decimal point,
-- including trailing zeroes if needed.
select trunc(d,2) from t1 order by d;
+-------------+
| trunc(d, 2) |
+-------------+
| 1.10        |
| 2.22        |
| 3.33        |
| 4.44        |
| 5.55        |
+-------------+

insert into t1 values (9999.9999), (8888.8888);

-- Negative scale truncates digits to the left
-- of the decimal point.
select trunc(d,-2) from t1 where d > 100 order by d;
+--------------+
| trunc(d, -2) |
+--------------+
| 8800         |
| 9900         |
+--------------+

-- The scale of the result is adjusted to match the
-- scale argument.
select trunc(d,2),
  precision(trunc(d,2)) as p,
  scale(trunc(d,2)) as s
from t1 order by d;
+-------------+----+---+
| trunc(d, 2) | p  | s |
+-------------+----+---+
| 1.10        | 15 | 2 |
| 2.22        | 15 | 2 |
| 3.33        | 15 | 2 |
| 4.44        | 15 | 2 |
| 5.55        | 15 | 2 |
| 8888.88     | 15 | 2 |
```

```
| 9999.99     | 15 | 2 |
+-------------+----+---+
```

```
create table dbl (d double);

insert into dbl values
  (1.1), (2.22), (3.333), (4.4444), (5.55555),
  (8888.8888), (9999.9999);

-- With double values, there is no optional scale argument.
select trunc(d) from dbl order by d;
+----------+
| trunc(d) |
+----------+
| 1        |
| 2        |
| 3        |
| 4        |
| 5        |
| 8888     |
| 9999     |
+----------+
```

UNHEX(STRING a)

Purpose: Returns a string of characters with ASCII values corresponding to pairs of hexadecimal digits in the argument.

Return type: `STRING`

WIDTH_BUCKET(DECIMAL expr, DECIMAL min_value, DECIMAL max_value, INT num_buckets)

Purpose: Returns the bucket number in which the `expr` value would fall in the histogram where its range between `min_value` and `max_value` is divided into `num_buckets` buckets of identical sizes.

The function returns:

- `NULL` if any argument is `NULL`.
- `0` if `expr` < `min_value`.
- `num_buckets + 1` if `expr` >= `max_val`.
- If none of the above, the bucket number where `expr` falls.

Arguments: The following rules apply to the arguments.

- `min_val` is the minimum value of the histogram range.
- `max_val` is the maximum value of the histogram range.
- `num_buckets` must be greater than `0`.
- `min_value` must be less than `max_value`.

Usage notes:

Each bucket contains values equal to or greater than the base value of that bucket and less than the base value of the next bucket. For example, with `width_bucket(8, 1, 10, 3)`, the bucket ranges are actually the 0th "underflow bucket" with the range (-infinity to 0.999...), (1 to 3.999...), (4, to 6.999...), (7 to 9.999...), and the "overflow bucket" with the range (10 to infinity).

Return type: `BIGINT`

Added in: CDH 6.1.

Examples:

The below function creates 3 buckets between the range of `1` and `20` with the bucket width of 6.333, and returns `2` for the bucket #2 where the value `8` falls in:

```
WIDTH_BUCKET(8, 1, 20, 3)
```

The below statement returns a list of accounts with the energy spending and the spending bracket each account falls in, between 0 and 11. Bucket 0 (underflow bucket) will be assigned to the accounts whose energy spendings are less than $50. Bucket 11 (overflow bucket) will be assigned to the accounts whose energy spendings are more than or equal to $1000.

```
SELECT account, invoice_amount, WIDTH_BUCKET(invoice_amount,50,1000,10)
FROM invoices_june2018
ORDER BY 3;
```

Impala Bit Functions

Bit manipulation functions perform bitwise operations involved in scientific processing or computer science algorithms. For example, these functions include setting, clearing, or testing bits within an integer value, or changing the positions of bits with or without wraparound.

If a function takes two integer arguments that are required to be of the same type, the smaller argument is promoted to the type of the larger one if required. For example, `BITAND(1,4096)` treats both arguments as `SMALLINT`, because 1 can be represented as a `TINYINT` but 4096 requires a `SMALLINT`.

Remember that all Impala integer values are signed. Therefore, when dealing with binary values where the most significant bit is 1, the specified or returned values might be negative when represented in base 10.

Whenever any argument is `NULL`, either the input value, bit position, or number of shift or rotate positions, the return value from any of these functions is also `NULL`

Related information:

The bit functions operate on all the integral data types: INT Data Type on page 117, BIGINT Data Type on page 105, SMALLINT Data Type on page 122, and TINYINT Data Type on page 136.

Function reference:

Impala supports the following bit functions:

- BITAND
- BITNOT
- BITOR
- BITXOR
- COUNTSET
- GETBIT
- ROTATELEFT
- ROTATERIGHT
- SETBIT
- SHIFTLEFT
- SHIFTRIGHT

BITAND(integer_type a, same_type b)

Purpose: Returns an integer value representing the bits that are set to 1 in both of the arguments. If the arguments are of different sizes, the smaller is promoted to the type of the larger.

Usage notes: The `BITAND()` function is equivalent to the & binary operator.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following examples show the results of ANDing integer values. 255 contains all 1 bits in its lowermost 7 bits. 32767 contains all 1 bits in its lowermost 15 bits. You can use the `bin()` function to check the binary representation

of any integer value, although the result is always represented as a 64-bit value. If necessary, the smaller argument is promoted to the type of the larger one.

```
select bitand(255, 32767); /* 0000000011111111 & 0111111111111111 */
+--------------------+
| bitand(255, 32767) |
+--------------------+
| 255                |
+--------------------+

select bitand(32767, 1); /* 0111111111111111 & 0000000000000001 */
+------------------+
| bitand(32767, 1) |
+------------------+
| 1                |
+------------------+

select bitand(32, 16); /* 00010000 & 00001000 */
+----------------+
| bitand(32, 16) |
+----------------+
| 0              |
+----------------+

select bitand(12,5); /* 00001100 & 00000101 */
+---------------+
| bitand(12, 5) |
+---------------+
| 4             |
+---------------+

select bitand(-1,15); /* 11111111 & 00001111 */
+----------------+
| bitand(-1, 15) |
+----------------+
| 15             |
+----------------+
```

BITNOT(integer_type a)

Purpose: Inverts all the bits of the input argument.

Usage notes: The `BITNOT()` function is equivalent to the ~ unary operator.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

These examples illustrate what happens when you flip all the bits of an integer value. The sign always changes. The decimal representation is one different between the positive and negative values.

```
select bitnot(127); /* 01111111 -> 10000000 */
+-------------+
| bitnot(127) |
+-------------+
| -128        |
+-------------+

select bitnot(16); /* 00010000 -> 11101111 */
+------------+
| bitnot(16) |
+------------+
| -17        |
+------------+

select bitnot(0); /* 00000000 -> 11111111 */
+-----------+
| bitnot(0) |
+-----------+
| -1        |
```

```
+-----------+

select bitnot(-128); /* 10000000 -> 01111111 */
+--------------+
| bitnot(-128) |
+--------------+
| 127          |
+--------------+
```

BITOR(integer_type a, same_type b)

Purpose: Returns an integer value representing the bits that are set to 1 in either of the arguments. If the arguments are of different sizes, the smaller is promoted to the type of the larger.

Usage notes: The `BITOR( )` function is equivalent to the `|` binary operator.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following examples show the results of ORing integer values.

```
select bitor(1,4); /* 00000001 | 00000100 */
+-------------+
| bitor(1, 4) |
+-------------+
| 5           |
+-------------+

select bitor(16,48); /* 00001000 | 00011000 */
+---------------+
| bitor(16, 48) |
+---------------+
| 48            |
+---------------+

select bitor(0,7); /* 00000000 | 00000111 */
+-------------+
| bitor(0, 7) |
+-------------+
| 7           |
+-------------+
```

BITXOR(integer_type a, same_type b)

Purpose: Returns an integer value representing the bits that are set to 1 in one but not both of the arguments. If the arguments are of different sizes, the smaller is promoted to the type of the larger.

Usage notes: The `BITXOR( )` function is equivalent to the `^` binary operator.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following examples show the results of XORing integer values. XORing a non-zero value with zero returns the non-zero value. XORing two identical values returns zero, because all the 1 bits from the first argument are also 1 bits in the second argument. XORing different non-zero values turns off some bits and leaves others turned on, based on whether the same bit is set in both arguments.

```
select bitxor(0,15); /* 00000000 ^ 00001111 */
+---------------+
| bitxor(0, 15) |
+---------------+
| 15            |
+---------------+
```

```
select bitxor(7,7); /* 00000111 ^ 00000111 */
+--------------+
| bitxor(7, 7) |
+--------------+
| 0            |
+--------------+

select bitxor(8,4); /* 00001000 ^ 00000100 */
+--------------+
| bitxor(8, 4) |
+--------------+
| 12           |
+--------------+

select bitxor(3,7); /* 00000011 ^ 00000111 */
+--------------+
| bitxor(3, 7) |
+--------------+
| 4            |
+--------------+
```

COUNTSET(integer_type a [, INT zero_or_one])

Purpose: By default, returns the number of 1 bits in the specified integer value. If the optional second argument is set to zero, it returns the number of 0 bits instead.

Usage notes:

In discussions of information theory, this operation is referred to as the "population count" or "popcount".

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following examples show how to count the number of 1 bits in an integer value.

```
select countset(1); /* 00000001 */
+-------------+
| countset(1) |
+-------------+
| 1           |
+-------------+

select countset(3); /* 00000011 */
+-------------+
| countset(3) |
+-------------+
| 2           |
+-------------+

select countset(16); /* 00010000 */
+--------------+
| countset(16) |
+--------------+
| 1            |
+--------------+

select countset(17); /* 00010001 */
+--------------+
| countset(17) |
+--------------+
| 2            |
+--------------+

select countset(7,1); /* 00000111 = 3 1 bits; the function counts 1 bits by default */
+----------------+
| countset(7, 1) |
+----------------+
| 3              |
+----------------+
```

```
select countset(7,0); /* 00000111 = 5 0 bits; third argument can only be 0 or 1 */
+----------------+
| countset(7, 0) |
+----------------+
| 5              |
+----------------+
```

GETBIT(integer_type a, INT position)

Purpose: Returns a 0 or 1 representing the bit at a specified position. The positions are numbered right to left, starting at zero. The position argument cannot be negative.

Usage notes:

When you use a literal input value, it is treated as an 8-bit, 16-bit, and so on value, the smallest type that is appropriate. The type of the input value limits the range of the positions. Cast the input value to the appropriate type if you need to ensure it is treated as a 64-bit, 32-bit, and so on value.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following examples show how to test a specific bit within an integer value.

```
select getbit(1,0); /* 00000001 */
+--------------+
| getbit(1, 0) |
+--------------+
| 1            |
+--------------+

select getbit(16,1) /* 00010000 */
+---------------+
| getbit(16, 1) |
+---------------+
| 0             |
+---------------+

select getbit(16,4) /* 00010000 */
+---------------+
| getbit(16, 4) |
+---------------+
| 1             |
+---------------+

select getbit(16,5) /* 00010000 */
+---------------+
| getbit(16, 5) |
+---------------+
| 0             |
+---------------+

select getbit(-1,3); /* 11111111 */
+---------------+
| getbit(-1, 3) |
+---------------+
| 1             |
+---------------+

select getbit(-1,25); /* 11111111 */
ERROR: Invalid bit position: 25

select getbit(cast(-1 as int),25); /* 11111111111111111111111111111111 */
+----------------------------+
| getbit(cast(-1 as int), 25) |
+----------------------------+
| 1                          |
+----------------------------+
```

ROTATELEFT(integer_type a, INT positions)

Purpose: Rotates an integer value left by a specified number of bits. As the most significant bit is taken out of the original value, if it is a 1 bit, it is "rotated" back to the least significant bit. Therefore, the final value has the same number of 1 bits as the original value, just in different positions. In computer science terms, this operation is a "circular shift".

Usage notes:

Specifying a second argument of zero leaves the original value unchanged. Rotating a -1 value by any number of positions still returns -1, because the original value has all 1 bits and all the 1 bits are preserved during rotation. Similarly, rotating a 0 value by any number of positions still returns 0. Rotating a value by the same number of bits as in the value returns the same value. Because this is a circular operation, the number of positions is not limited to the number of bits in the input value. For example, rotating an 8-bit value by 1, 9, 17, and so on positions returns an identical result in each case.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
select rotateleft(1,4); /* 00000001 -> 00010000 */
+------------------+
| rotateleft(1, 4) |
+------------------+
| 16               |
+------------------+

select rotateleft(-1,155); /* 11111111 -> 11111111 */
+--------------------+
| rotateleft(-1, 155) |
+--------------------+
| -1                 |
+--------------------+

select rotateleft(-128,1); /* 10000000 -> 00000001 */
+--------------------+
| rotateleft(-128, 1) |
+--------------------+
| 1                  |
+--------------------+

select rotateleft(-127,3); /* 10000001 -> 00001100 */
+--------------------+
| rotateleft(-127, 3) |
+--------------------+
| 12                 |
+--------------------+
```

ROTATERIGHT(integer_type a, INT positions)

Purpose: Rotates an integer value right by a specified number of bits. As the least significant bit is taken out of the original value, if it is a 1 bit, it is "rotated" back to the most significant bit. Therefore, the final value has the same number of 1 bits as the original value, just in different positions. In computer science terms, this operation is a "circular shift".

Usage notes:

Specifying a second argument of zero leaves the original value unchanged. Rotating a -1 value by any number of positions still returns -1, because the original value has all 1 bits and all the 1 bits are preserved during rotation. Similarly, rotating a 0 value by any number of positions still returns 0. Rotating a value by the same number of bits as in the value returns the same value. Because this is a circular operation, the number of positions is not limited to the number of bits in the input value. For example, rotating an 8-bit value by 1, 9, 17, and so on positions returns an identical result in each case.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
select rotateright(16,4); /* 00010000 -> 00000001 */
+--------------------+
| rotateright(16, 4) |
+--------------------+
| 1                  |
+--------------------+

select rotateright(-1,155); /* 11111111 -> 11111111 */
+----------------------+
| rotateright(-1, 155) |
+----------------------+
| -1                   |
+----------------------+

select rotateright(-128,1); /* 10000000 -> 01000000 */
+----------------------+
| rotateright(-128, 1) |
+----------------------+
| 64                   |
+----------------------+

select rotateright(-127,3); /* 10000001 -> 00110000 */
+----------------------+
| rotateright(-127, 3) |
+----------------------+
| 48                   |
+----------------------+
```

SETBIT(integer_type a, INT position [, INT zero_or_one])

Purpose: By default, changes a bit at a specified position to a 1, if it is not already. If the optional third argument is set to zero, the specified bit is set to 0 instead.

Usage notes:

If the bit at the specified position was already 1 (by default) or 0 (with a third argument of zero), the return value is the same as the first argument. The positions are numbered right to left, starting at zero. (Therefore, the return value could be different from the first argument even if the position argument is zero.) The position argument cannot be negative.

When you use a literal input value, it is treated as an 8-bit, 16-bit, and so on value, the smallest type that is appropriate. The type of the input value limits the range of the positions. Cast the input value to the appropriate type if you need to ensure it is treated as a 64-bit, 32-bit, and so on value.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
select setbit(0,0); /* 00000000 -> 00000001 */
+--------------+
| setbit(0, 0) |
+--------------+
| 1            |
+--------------+

select setbit(0,3); /* 00000000 -> 00001000 */
+--------------+
| setbit(0, 3) |
+--------------+
| 8            |
+--------------+

select setbit(7,3); /* 00000111 -> 00001111 */
+--------------+
| setbit(7, 3) |
```

```
+--------------+
| 15           |
+--------------+

select setbit(15,3); /* 00001111 -> 00001111 */
+---------------+
| setbit(15, 3) |
+---------------+
| 15            |
+---------------+

select setbit(0,32); /* By default, 0 is a TINYINT with only 8 bits. */
ERROR: Invalid bit position: 32

select setbit(cast(0 as bigint),32); /* For BIGINT, the position can be 0..63. */
+--------------------------------+
| setbit(cast(0 as bigint), 32) |
+--------------------------------+
| 4294967296                     |
+--------------------------------+

select setbit(7,3,1); /* 00000111 -> 00001111; setting to 1 is the default */
+-----------------+
| setbit(7, 3, 1) |
+-----------------+
| 15              |
+-----------------+

select setbit(7,2,0); /* 00000111 -> 00000011; third argument of 0 clears instead of
sets */
+-----------------+
| setbit(7, 2, 0) |
+-----------------+
| 3               |
+-----------------+
```

SHIFTLEFT(integer_type a, INT positions)

Purpose: Shifts an integer value left by a specified number of bits. As the most significant bit is taken out of the original value, it is discarded and the least significant bit becomes 0. In computer science terms, this operation is a "logical shift".

Usage notes:

The final value has either the same number of 1 bits as the original value, or fewer. Shifting an 8-bit value by 8 positions, a 16-bit value by 16 positions, and so on produces a result of zero.

Specifying a second argument of zero leaves the original value unchanged. Shifting any value by 0 returns the original value. Shifting any value by 1 is the same as multiplying it by 2, as long as the value is small enough; larger values eventually become negative when shifted, as the sign bit is set. Starting with the value 1 and shifting it left by N positions gives the same result as 2 to the Nth power, or `pow(2,N)`.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
select shiftleft(1,0); /* 00000001 -> 00000001 */
+-----------------+
| shiftleft(1, 0) |
+-----------------+
| 1               |
+-----------------+

select shiftleft(1,3); /* 00000001 -> 00001000 */
+-----------------+
| shiftleft(1, 3) |
+-----------------+
| 8               |
+-----------------+
```

```
select shiftleft(8,2); /* 00001000 -> 00100000 */
+-----------------+
| shiftleft(8, 2) |
+-----------------+
| 32              |
+-----------------+

select shiftleft(127,1); /* 01111111 -> 11111110 */
+-------------------+
| shiftleft(127, 1) |
+-------------------+
| -2                |
+-------------------+

select shiftleft(127,5); /* 01111111 -> 11100000 */
+-------------------+
| shiftleft(127, 5) |
+-------------------+
| -32               |
+-------------------+

select shiftleft(-1,4); /* 11111111 -> 11110000 */
+------------------+
| shiftleft(-1, 4) |
+------------------+
| -16              |
+------------------+
```

SHIFTRIGHT(integer_type a, INT positions)

Purpose: Shifts an integer value right by a specified number of bits. As the least significant bit is taken out of the original value, it is discarded and the most significant bit becomes 0. In computer science terms, this operation is a "logical shift".

Usage notes:

Therefore, the final value has either the same number of 1 bits as the original value, or fewer. Shifting an 8-bit value by 8 positions, a 16-bit value by 16 positions, and so on produces a result of zero.

Specifying a second argument of zero leaves the original value unchanged. Shifting any value by 0 returns the original value. Shifting any positive value right by 1 is the same as dividing it by 2. Negative values become positive when shifted right.

Return type: Same as the input value

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
select shiftright(16,0); /* 00010000 -> 00010000 */
+-------------------+
| shiftright(16, 0) |
+-------------------+
| 16                |
+-------------------+

select shiftright(16,4); /* 00010000 -> 00000001 */
+-------------------+
| shiftright(16, 4) |
+-------------------+
| 1                 |
+-------------------+

select shiftright(16,5); /* 00010000 -> 00000000 */
+-------------------+
| shiftright(16, 5) |
+-------------------+
| 0                 |
+-------------------+
```

```
select shiftright(-1,1); /* 11111111 -> 01111111 */
+-------------------+
| shiftright(-1, 1) |
+-------------------+
| 127               |
+-------------------+

select shiftright(-1,5); /* 11111111 -> 00000111 */
+-------------------+
| shiftright(-1, 5) |
+-------------------+
| 7                 |
+-------------------+
```

Impala Type Conversion Functions

Conversion functions are usually used in combination with other functions, to explicitly pass the expected data types. Impala has strict rules regarding data types for function parameters. For example, Impala does not automatically convert a DOUBLE value to FLOAT, a BIGINT value to INT, or other conversion where precision could be lost or overflow could occur. Also, for reporting or dealing with loosely defined schemas in big data contexts, you might frequently need to convert values to or from the STRING type.



Note: Although in CDH 5.5 / Impala 2.3, the SHOW FUNCTIONS output for database _IMPALA_BUILTINS contains some function signatures matching the pattern castto*, these functions are not intended for public use and are expected to be hidden in future.

Function reference:

Impala supports the following type conversion functions:

- CAST
- TYPEOF

CAST(expr AS type)

Purpose: Converts the value of an expression to any other type. If the expression value is of a type that cannot be converted to the target type, the result is NULL.

Usage notes: Use CAST when passing a column value or literal to a function that expects a parameter with a different type. Frequently used in SQL operations such as CREATE TABLE AS SELECT and INSERT ... VALUES to ensure that values from various sources are of the appropriate type for the destination columns. Where practical, do a one-time CAST() operation during the ingestion process to make each column into the appropriate type, rather than using many CAST() operations in each query; doing type conversions for each row during each query can be expensive for tables with millions or billions of rows.

The way this function deals with time zones when converting to or from TIMESTAMP values is affected by the --use_local_tz_for_unix_timestamp_conversions startup flag for the impalad daemon. See TIMESTAMP Data Type on page 130 for details about how Impala handles time zone considerations for the TIMESTAMP data type.

Examples:

```
SELECT CONCAT('Here are the first ',10,' results.'); -- Fails
SELECT CONCAT('Here are the first ',CAST(10 AS STRING),' results.'); -- Succeeds
```

The following example starts with a text table where every column has a type of STRING, which might be how you ingest data of unknown schema until you can verify the cleanliness of the underly values. Then it uses CAST() to create a new Parquet table with the same data, but using specific numeric data types for the columns with numeric

data. Using numeric types of appropriate sizes can result in substantial space savings on disk and in memory, and performance improvements in queries, over using strings or larger-than-necessary numeric types.

```
CREATE TABLE t1 (name STRING, x STRING, y STRING, z STRING);

CREATE TABLE t2 STORED AS PARQUET
AS SELECT
  name,
  CAST(x AS BIGINT) x,
  CAST(y AS TIMESTAMP) y,
  CAST(z AS SMALLINT) z
FROM t1;
```

Related information:

For details of casts from each kind of data type, see the description of the appropriate type: TINYINT Data Type on page 136, SMALLINT Data Type on page 122, INT Data Type on page 117, BIGINT Data Type on page 105, FLOAT Data Type on page 116, DOUBLE Data Type on page 114, DECIMAL Data Type (CDH 6.0 / Impala 3.0 or higher only) on page 109, STRING Data Type on page 123, CHAR Data Type (CDH 5.2 or higher only) on page 107, VARCHAR Data Type (CDH 5.2 or higher only) on page 137, TIMESTAMP Data Type on page 130, BOOLEAN Data Type on page 106

TYPEOF(type value)

Purpose: Returns the name of the data type corresponding to an expression. For types with extra attributes, such as length for CHAR and VARCHAR, or precision and scale for DECIMAL, includes the full specification of the type.

Return type: STRING

Usage notes: Typically used in interactive exploration of a schema, or in application code that programmatically generates schema definitions such as CREATE TABLE statements, for example, to get the type of an expression such as col1 / col2 or CONCAT(col1, col2, col3). This function is especially useful for arithmetic expressions involving DECIMAL types because the precision and scale of the result is can be different than that of the operands.

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
SELECT TYPEOF(2), TYPEOF(2+2);
+-----------+---------------+
| typeof(2) | typeof(2 + 2) |
+-----------+---------------+
| TINYINT   | SMALLINT      |
+-----------+---------------+
```

Impala Date and Time Functions

The underlying Impala data type for date and time data is TIMESTAMP, which has both a date and a time portion. Functions that extract a single field, such as hour() or minute(), typically return an integer value. Functions that format the date portion, such as date_add() or to_date(), typically return a string value.

You can also adjust a TIMESTAMP value by adding or subtracting an INTERVAL expression. See TIMESTAMP Data Type on page 130 for details. INTERVAL expressions are also allowed as the second argument for the date_add() and date_sub() functions, rather than integers.

Some of these functions are affected by the setting of the --use_local_tz_for_unix_timestamp_conversions startup flag for the impalad daemon. This setting is off by default, meaning that functions such as from_unixtime() and unix_timestamp() consider the input values to always represent the UTC time zone. This setting also applies when you CAST() a BIGINT value to TIMESTAMP, or a TIMESTAMP value to BIGINT. When this setting is enabled, these functions and operations convert to and from values representing the local time zone. See TIMESTAMP Data Type on page 130 for details about how Impala handles time zone considerations for the TIMESTAMP data type.

Function reference:

Impala supports the following data and time functions:

- ADD_MONTHS

- ADDDATE
- CURRENT_TIMESTAMP
- DATE_ADD
- DATE_PART
- DATE_SUB
- DATE_TRUNC
- DATEDIFF
- DAY
- DAYNAME
- DAYOFWEEK
- DAYOFYEAR
- DAYS_ADD
- DAYS_SUB
- EXTRACT
- FROM_TIMESTAMP
- FROM_UNIXTIME
- FROM_UTC_TIMESTAMP
- HOUR
- HOURS_ADD
- HOURS_SUB
- INT_MONTHS_BETWEEN
- MICROSECONDS_ADD
- MICROSECONDS_SUB
- MILLISECOND
- MILLISECONDS_ADD
- MILLISECONDS_SUB
- MINUTE
- MINUTES_ADD
- MINUTES_SUB
- MONTH
- MONTHNAME
- MONTHS_ADD
- MONTHS_BETWEEN
- MONTHS_SUB
- NANOSECONDS_ADD
- NANOSECONDS_SUB
- NEXT_DAY
- NOW
- QUARTER
- SECOND
- SECONDS_ADD
- SECONDS_SUB
- SUBDATE
- TIMEOFDAY
- TIMESTAMP_CMP
- TO_DATE
- TO_TIMESTAMP
- TO_UTC_TIMESTAMP
- TRUNC
- UNIX_TIMESTAMP

- UTC_TIMESTAMP
- WEEKOFYEAR
- WEEKS_ADD
- WEEKS_SUB
- YEAR
- YEARS_ADD
- YEARS_SUB

ADD_MONTHS(TIMESTAMP date, INT months), ADD_MONTHS(TIMESTAMP date, BIGINT months)

Purpose: Returns the specified date and time plus some number of months.

Return type: TIMESTAMP

Usage notes:

Same as MONTHS_ADD(). Available in Impala 1.4 and higher. For compatibility when porting code with vendor extensions.

Examples:

The following examples demonstrate adding months to construct the same day of the month in a different month; how if the current day of the month does not exist in the target month, the last day of that month is substituted; and how a negative argument produces a return value from a previous month.

```
select now(), add_months(now(), 2);
+-------------------------------+-------------------------------+
| now()                         | add_months(now(), 2)          |
+-------------------------------+-------------------------------+
| 2016-05-31 10:47:00.429109000 | 2016-07-31 10:47:00.429109000 |
+-------------------------------+-------------------------------+

select now(), add_months(now(), 1);
+-------------------------------+-------------------------------+
| now()                         | add_months(now(), 1)          |
+-------------------------------+-------------------------------+
| 2016-05-31 10:47:14.540226000 | 2016-06-30 10:47:14.540226000 |
+-------------------------------+-------------------------------+

select now(), add_months(now(), -1);
+-------------------------------+-------------------------------+
| now()                         | add_months(now(), -1)         |
+-------------------------------+-------------------------------+
| 2016-05-31 10:47:31.732298000 | 2016-04-30 10:47:31.732298000 |
+-------------------------------+-------------------------------+
```

ADDDATE(TIMESTAMP startdate, INT days), ADDDATE(TIMESTAMP startdate, BIGINT days)

Purpose: Adds a specified number of days to a TIMESTAMP value. Similar to DATE_ADD( ), but starts with an actual TIMESTAMP value instead of a string that is converted to a TIMESTAMP.

Return type: TIMESTAMP

Examples:

The following examples show how to add a number of days to a TIMESTAMP. The number of days can also be negative, which gives the same effect as the subdate( ) function.

```
select now() as right_now, adddate(now(), 30) as now_plus_30;
+-------------------------------+-------------------------------+
| right_now                     | now_plus_30                   |
+-------------------------------+-------------------------------+
| 2016-05-20 10:23:08.640111000 | 2016-06-19 10:23:08.640111000 |
+-------------------------------+-------------------------------+

select now() as right_now, adddate(now(), -15) as now_minus_15;
```

```
+-------------------------------+-------------------------------+
| right_now                     | now_minus_15                  |
+-------------------------------+-------------------------------+
| 2016-05-20 10:23:38.214064000 | 2016-05-05 10:23:38.214064000 |
+-------------------------------+-------------------------------+
```

CURRENT_TIMESTAMP()

Purpose: Alias for the `NOW()` function.

Return type: `TIMESTAMP`

Examples:

```
select now(), current_timestamp();
+-------------------------------+-------------------------------+
| now()                         | current_timestamp()           |
+-------------------------------+-------------------------------+
| 2016-05-19 16:10:14.237849000 | 2016-05-19 16:10:14.237849000 |
+-------------------------------+-------------------------------+

select current_timestamp() as right_now,
  current_timestamp() + interval 3 hours as in_three_hours;
+-------------------------------+-------------------------------+
| right_now                     | in_three_hours                |
+-------------------------------+-------------------------------+
| 2016-05-19 16:13:20.017117000 | 2016-05-19 19:13:20.017117000 |
+-------------------------------+-------------------------------+
```

DATE_ADD(TIMESTAMP startdate, INT days), DATE_ADD(TIMESTAMP startdate, interval_expression)

Purpose: Adds a specified number of days to a `TIMESTAMP` value. With an `INTERVAL` expression as the second argument, you can calculate a delta value using other units such as weeks, years, hours, seconds, and so on; see TIMESTAMP Data Type on page 130 for details.

Return type: `TIMESTAMP`

Examples:

The following example shows the simplest usage, of adding a specified number of days to a `TIMESTAMP` value:

```
select now() as right_now, date_add(now(), 7) as next_week;
+-------------------------------+-------------------------------+
| right_now                     | next_week                     |
+-------------------------------+-------------------------------+
| 2016-05-20 11:03:48.687055000 | 2016-05-27 11:03:48.687055000 |
+-------------------------------+-------------------------------+
```

The following examples show the shorthand notation of an `INTERVAL` expression, instead of specifying the precise number of days. The `INTERVAL` notation also lets you work with units smaller than a single day.

```
select now() as right_now, date_add(now(), interval 3 weeks) as in_3_weeks;
+-------------------------------+-------------------------------+
| right_now                     | in_3_weeks                    |
+-------------------------------+-------------------------------+
| 2016-05-20 11:05:39.173331000 | 2016-06-10 11:05:39.173331000 |
+-------------------------------+-------------------------------+

select now() as right_now, date_add(now(), interval 6 hours) as in_6_hours;
+-------------------------------+-------------------------------+
| right_now                     | in_6_hours                    |
+-------------------------------+-------------------------------+
| 2016-05-20 11:13:51.492536000 | 2016-05-20 17:13:51.492536000 |
+-------------------------------+-------------------------------+
```

Like all date/time functions that deal with months, `date_add()` handles nonexistent dates past the end of a month by setting the date to the last day of the month. The following example shows how the nonexistent date April 31st is normalized to April 30th:

```
select date_add(cast('2016-01-31' as timestamp), interval 3 months) as 'april_31st';
+---------------------+
| april_31st          |
+---------------------+
| 2016-04-30 00:00:00 |
+---------------------+
```

DATE_PART(STRING a, TIMESTAMP timestamp)

Purpose: Similar to EXTRACT(), with the argument order reversed. Supports the same date and time units as EXTRACT(). For compatibility with SQL code containing vendor extensions.

Return type: `bigint`

Examples:

```
select date_part('year',now()) as current_year;
+--------------+
| current_year |
+--------------+
| 2016         |
+--------------+

select date_part('hour',now()) as hour_of_day;
+-------------+
| hour_of_day |
+-------------+
| 11          |
+-------------+
```

DATE_SUB(TIMESTAMP startdate, INT days), DATE_SUB(TIMESTAMP startdate, interval_expression)

Purpose: Subtracts a specified number of days from a `TIMESTAMP` value. With an `INTERVAL` expression as the second argument, you can calculate a delta value using other units such as weeks, years, hours, seconds, and so on; see TIMESTAMP Data Type on page 130 for details.

Return type: `TIMESTAMP`

Examples:

The following example shows the simplest usage, of subtracting a specified number of days from a `TIMESTAMP` value:

```
select now() as right_now, date_sub(now(), 7) as last_week;
+-------------------------------+-------------------------------+
| right_now                     | last_week                     |
+-------------------------------+-------------------------------+
| 2016-05-20 11:21:30.491011000 | 2016-05-13 11:21:30.491011000 |
+-------------------------------+-------------------------------+
```

The following examples show the shorthand notation of an `INTERVAL` expression, instead of specifying the precise number of days. The `INTERVAL` notation also lets you work with units smaller than a single day.

```
select now() as right_now, date_sub(now(), interval 3 weeks) as 3_weeks_ago;
+-------------------------------+-------------------------------+
| right_now                     | 3_weeks_ago                   |
+-------------------------------+-------------------------------+
| 2016-05-20 11:23:05.176953000 | 2016-04-29 11:23:05.176953000 |
+-------------------------------+-------------------------------+

select now() as right_now, date_sub(now(), interval 6 hours) as 6_hours_ago;
```

```
+-------------------------------+-------------------------------+
| right_now                     | 6_hours_ago                   |
+-------------------------------+-------------------------------+
| 2016-05-20 11:23:35.439631000 | 2016-05-20 05:23:35.439631000 |
+-------------------------------+-------------------------------+
```

Like all date/time functions that deal with months, DATE_ADD( ) handles nonexistent dates past the end of a month by setting the date to the last day of the month. The following example shows how the nonexistent date April 31st is normalized to April 30th:

```
select date_sub(cast('2016-05-31' as timestamp), interval 1 months) as 'april_31st';
+---------------------+
| april_31st          |
+---------------------+
| 2016-04-30 00:00:00 |
+---------------------+
```

DATE_TRUNC(STRING unit, TIMESTAMP timestamp)

Purpose: Truncates a TIMESTAMP value to the specified precision.

Unit argument: The unit argument value for truncating TIMESTAMP values is not case-sensitive. This argument string can be one of:

- microseconds
- milliseconds
- second
- minute
- hour
- day
- week
- month
- year
- decade
- century
- millennium

For example, calling date_trunc('hour',ts) truncates ts to the beginning of the corresponding hour, with all minutes, seconds, milliseconds, and so on set to zero. Calling date_trunc('milliseconds',ts) truncates ts to the beginning of the corresponding millisecond, with all microseconds and nanoseconds set to zero.



Note: The sub-second units are specified in plural form. All units representing one second or more are specified in singular form.

Added in: CDH 5.14.0 / Impala 2.11.0

Usage notes:

Although this function is similar to calling TRUNC( ) with a TIMESTAMP argument, the order of arguments and the recognized units are different between TRUNC( ) and DATE_TRUNC( ). Therefore, these functions are not interchangeable.

This function is typically used in GROUP BY queries to aggregate results from the same hour, day, week, month, quarter, and so on. You can also use this function in an INSERT ... SELECT into a partitioned table to divide TIMESTAMP values into the correct partition.

Because the return value is a TIMESTAMP, if you cast the result of DATE_TRUNC( ) to STRING, you will often see zeroed-out portions such as 00:00:00 in the time field. If you only need the individual units such as hour, day, month, or year, use the EXTRACT( ) function instead. If you need the individual units from a truncated TIMESTAMP value, run the TRUNCATE( ) function on the original value, then run EXTRACT( ) on the result.

Return type: TIMESTAMP

Examples:

The following examples show how to call DATE_TRUNC( ) with different unit values:

```
select now(), date_trunc('second', now());
+-------------------------------+-----------------------------------+
| now()                         | date_trunc('second', now())       |
+-------------------------------+-----------------------------------+
| 2017-12-05 13:58:04.565403000 | 2017-12-05 13:58:04               |
+-------------------------------+-----------------------------------+

select now(), date_trunc('hour', now());
+-------------------------------+---------------------------+
| now()                         | date_trunc('hour', now()) |
+-------------------------------+---------------------------+
| 2017-12-05 13:59:01.884459000 | 2017-12-05 13:00:00       |
+-------------------------------+---------------------------+

select now(), date_trunc('millennium', now());
+-------------------------------+---------------------------------+
| now()                         | date_trunc('millennium', now()) |
+-------------------------------+---------------------------------+
| 2017-12-05 14:00:30.296812000 | 2000-01-01 00:00:00             |
+-------------------------------+---------------------------------+
```

DATEDIFF(TIMESTAMP enddate, TIMESTAMP startdate)

Purpose: Returns the number of days between two TIMESTAMP values.

Return type: INT

Usage notes:

If the first argument represents a later date than the second argument, the return value is positive. If both arguments represent the same date, the return value is zero. The time portions of the TIMESTAMP values are irrelevant. For example, 11:59 PM on one day and 12:01 on the next day represent a datediff( ) of -1 because the date/time values represent different days, even though the TIMESTAMP values differ by only 2 minutes.

Examples:

The following example shows how comparing a “late” value with an “earlier” value produces a positive number. In this case, the result is (365 * 5) + 1, because one of the intervening years is a leap year.

```
select now() as right_now, datediff(now() + interval 5 years, now()) as in_5_years;
+-------------------------------+------------+
| right_now                     | in_5_years |
+-------------------------------+------------+
| 2016-05-20 13:43:55.873826000 | 1826       |
+-------------------------------+------------+
```

The following examples show how the return value represent the number of days between the associated dates, regardless of the time portion of each TIMESTAMP. For example, different times on the same day produce a DATE_DIFF( ) of 0, regardless of which one is earlier or later. But if the arguments represent different dates, DATE_DIFF( ) returns a non-zero integer value, regardless of the time portions of the dates.

```
select now() as right_now, datediff(now(), now() + interval 4 hours) as in_4_hours;
+-------------------------------+------------+
| right_now                     | in_4_hours |
+-------------------------------+------------+
| 2016-05-20 13:42:05.302747000 | 0          |
+-------------------------------+------------+

select now() as right_now, datediff(now(), now() - interval 4 hours) as 4_hours_ago;
+-------------------------------+-------------+
```

```
| right_now                     | 4_hours_ago |
+-------------------------------+-------------+
| 2016-05-20 13:42:21.134958000 | 0           |
+-------------------------------+-------------+

select now() as right_now, datediff(now(), now() + interval 12 hours) as in_12_hours;
+-------------------------------+-------------+
| right_now                     | in_12_hours |
+-------------------------------+-------------+
| 2016-05-20 13:42:44.765873000 | -1          |
+-------------------------------+-------------+

select now() as right_now, datediff(now(), now() - interval 18 hours) as 18_hours_ago;
+-------------------------------+--------------+
| right_now                     | 18_hours_ago |
+-------------------------------+--------------+
| 2016-05-20 13:54:38.829827000 | 1            |
+-------------------------------+--------------+
```

DAY(TIMESTAMP date), DAYOFMONTH(TIMESTAMP date)

Purpose: Returns the day field from the date portion of a `TIMESTAMP`. The value represents the day of the month, therefore is in the range 1-31, or less for months without 31 days.

Return type: `INT`

Examples:

The following examples show how the day value corresponds to the day of the month, resetting back to 1 at the start of each month.

```
select now(), day(now());
+-------------------------------+------------+
| now()                         | day(now()) |
+-------------------------------+------------+
| 2016-05-20 15:01:51.042185000 | 20         |
+-------------------------------+------------+

select now() + interval 11 days, day(now() + interval 11 days);
+-------------------------------+-------------------------------+
| now() + interval 11 days      | day(now() + interval 11 days) |
+-------------------------------+-------------------------------+
| 2016-05-31 15:05:56.843139000 | 31                            |
+-------------------------------+-------------------------------+

select now() + interval 12 days, day(now() + interval 12 days);
+-------------------------------+-------------------------------+
| now() + interval 12 days      | day(now() + interval 12 days) |
+-------------------------------+-------------------------------+
| 2016-06-01 15:06:05.074236000 | 1                             |
+-------------------------------+-------------------------------+
```

The following examples show how the day value is `NULL` for nonexistent dates or misformatted date strings.

```
-- 2016 is a leap year, so it has a Feb. 29.
select day('2016-02-29');
+-------------------+
| day('2016-02-29') |
+-------------------+
| 29                |
+-------------------+

-- 2015 is not a leap year, so Feb. 29 is nonexistent.
select day('2015-02-29');
+-------------------+
| day('2015-02-29') |
+-------------------+
| NULL              |
+-------------------+
```

```
-- A string that does not match the expected YYYY-MM-DD format
-- produces an invalid TIMESTAMP, causing day() to return NULL.
select day('2016-02-028');
+--------------------+
| day('2016-02-028') |
+--------------------+
| NULL               |
+--------------------+
```

DAYNAME(TIMESTAMP date)

Purpose: Returns the day field from a TIMESTAMP value, converted to the string corresponding to that day name. The range of return values is 'Sunday' to 'Saturday'. Used in report-generating queries, as an alternative to calling DAYOFWEEK( ) and turning that numeric return value into a string using a CASE expression.

Return type: STRING

Examples:

The following examples show the day name associated with TIMESTAMP values representing different days.

```
select now() as right_now,
  dayofweek(now()) as todays_day_of_week,
  dayname(now()) as todays_day_name;
+-------------------------------+--------------------+-----------------+
| right_now                     | todays_day_of_week | todays_day_name |
+-------------------------------+--------------------+-----------------+
| 2016-05-31 10:57:03.953670000 | 3                  | Tuesday         |
+-------------------------------+--------------------+-----------------+

select now() + interval 1 day as tomorrow,
  dayname(now() + interval 1 day) as tomorrows_day_name;
+-------------------------------+--------------------+
| tomorrow                      | tomorrows_day_name |
+-------------------------------+--------------------+
| 2016-06-01 10:58:53.945761000 | Wednesday          |
+-------------------------------+--------------------+
```

DAYOFWEEK(TIMESTAMP date)

Purpose: Returns the day field from the date portion of a TIMESTAMP, corresponding to the day of the week. The range of return values is 1 (Sunday) to 7 (Saturday).

Return type: INT

Examples:

```
select now() as right_now,
  dayofweek(now()) as todays_day_of_week,
  dayname(now()) as todays_day_name;
+-------------------------------+--------------------+-----------------+
| right_now                     | todays_day_of_week | todays_day_name |
+-------------------------------+--------------------+-----------------+
| 2016-05-31 10:57:03.953670000 | 3                  | Tuesday         |
+-------------------------------+--------------------+-----------------+
```

DAYOFYEAR(TIMESTAMP date)

Purpose: Returns the day field from a TIMESTAMP value, corresponding to the day of the year. The range of return values is 1 (January 1) to 366 (December 31 of a leap year).

Return type: INT

Examples:

The following examples show return values from the `dayofyear()` function. The same date in different years returns a different day number for all dates after February 28, because 2016 is a leap year while 2015 is not a leap year.

```
select now() as right_now,
  dayofyear(now()) as today_day_of_year;
+-------------------------------+-------------------+
| right_now                     | today_day_of_year |
+-------------------------------+-------------------+
| 2016-05-31 11:05:48.314932000 | 152               |
+-------------------------------+-------------------+

select now() - interval 1 year as last_year,
  dayofyear(now() - interval 1 year) as year_ago_day_of_year;
+-------------------------------+----------------------+
| last_year                     | year_ago_day_of_year |
+-------------------------------+----------------------+
| 2015-05-31 11:07:03.733689000 | 151                  |
+-------------------------------+----------------------+
```

DAYS_ADD(TIMESTAMP startdate, INT days), DAYS_ADD(TIMESTAMP startdate, BIGINT days)

Purpose: Adds a specified number of days to a `TIMESTAMP` value. Similar to `DATE_ADD()`, but starts with an actual `TIMESTAMP` value instead of a string that is converted to a `TIMESTAMP`.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now, days_add(now(), 31) as 31_days_later;
+-------------------------------+-------------------------------+
| right_now                     | 31_days_later                 |
+-------------------------------+-------------------------------+
| 2016-05-31 11:12:32.216764000 | 2016-07-01 11:12:32.216764000 |
+-------------------------------+-------------------------------+
```

DAYS_SUB(TIMESTAMP startdate, INT days), DAYS_SUB(TIMESTAMP startdate, BIGINT days)

Purpose: Subtracts a specified number of days from a `TIMESTAMP` value. Similar to `DATE_SUB()`, but starts with an actual `TIMESTAMP` value instead of a string that is converted to a `TIMESTAMP`.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now, days_sub(now(), 31) as 31_days_ago;
+-------------------------------+-------------------------------+
| right_now                     | 31_days_ago                   |
+-------------------------------+-------------------------------+
| 2016-05-31 11:13:42.163905000 | 2016-04-30 11:13:42.163905000 |
+-------------------------------+-------------------------------+
```

EXTRACT(TIMESTAMP timestamp, STRING unit), EXTRACT(unit FROM TIMESTAMP ts)

Purpose: Returns one of the numeric date or time fields from a `TIMESTAMP` value.

Unit argument: The `unit` string can be one of `epoch`, `year`, `quarter`, `month`, `day`, `hour`, `minute`, `second`, or `millisecond`. This argument value is case-insensitive.

If you specify `millisecond` for the `unit` argument, the function returns the seconds component and the milliseconds component. For example, `EXTRACT(CAST('2006-05-12 18:27:28.123456789' AS TIMESTAMP), 'MILLISECOND')` will return 28123.

In Impala 2.0 and higher, you can use special syntax rather than a regular function call, for compatibility with code that uses the SQL-99 format with the FROM keyword. With this style, the unit names are identifiers rather than STRING literals. For example, the following calls are both equivalent:

```
EXTRACT(year FROM NOW());
EXTRACT(NOW(), 'year');
```

Usage notes:

Typically used in GROUP BY queries to arrange results by hour, day, month, and so on. You can also use this function in an INSERT ... SELECT statement to insert into a partitioned table to split up TIMESTAMP values into individual parts, if the partitioned table has separate partition key columns representing year, month, day, and so on. If you need to divide by more complex units of time, such as by week or by quarter, use the TRUNC( ) function instead.

Return type: BIGINT

Examples:

```
SELECT NOW() AS right_now,
  EXTRACT(day FROM NOW()) AS this_day,
  EXTRACT(hour FROM NOW()) AS this_hour;
+-------------------------------+----------+-----------+
| right_now                     | this_day | this_hour |
+-------------------------------+----------+-----------+
| 2016-05-31 11:19:24.025303000 | 31       | 11        |
+-------------------------------+----------+-----------+
```

FROM_TIMESTAMP(TIMESTAMP datetime, STRING pattern), FROM_TIMESTAMP(STRING datetime, STRING pattern)

Purpose: Converts a TIMESTAMP value into a string representing the same value.

Return type: STRING

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes:

The FROM_TIMESTAMP( ) function provides a flexible way to convert TIMESTAMP values into arbitrary string formats for reporting purposes.

Because Impala implicitly converts string values into TIMESTAMP, you can pass date/time values represented as strings (in the standard yyyy-MM-dd HH:mm:ss.SSS format) to this function. The result is a string using different separator characters, order of fields, spelled-out month names, or other variation of the date/time string representation.

The allowed tokens for the pattern string are the same as for the FROM_UNIXTIME( ) function.

Examples:

The following examples show different ways to format a TIMESTAMP value as a string:

```
-- Reformat a TIMESTAMP value.
SELECT FROM_TIMESTAMP(NOW(), 'yyyy/MM/dd');
+-----------------------------------+
| from_timestamp(now(), 'yyyy/mm/dd') |
+-----------------------------------+
| 2018/10/09                        |
+-----------------------------------+

-- Alternative format for reporting purposes.
SELECT FROM_TIMESTAMP('1984-09-25 16:45:30.125', 'MMM dd, yyyy HH:mm:ss.SSS');
+-----------------------------------------------------------------------+
| from_timestamp('1984-09-25 16:45:30.125', 'mmm dd, yyyy hh:mm:ss.sss') |
+-----------------------------------------------------------------------+
| Sep 25, 1984 16:45:30.125                                             |
+-----------------------------------------------------------------------+
```

FROM_UNIXTIME(BIGINT unixtime[, STRING format])

Purpose: Converts the number of seconds from the Unix epoch to the specified time into a string in the local time zone.

Return type: `STRING`

In Impala 2.2.0 and higher, built-in functions that accept or return integers representing `TIMESTAMP` values use the `BIGINT` type for parameters and return values, rather than `INT`. This change lets the date and time functions avoid an overflow error that would otherwise occur on January 19th, 2038 (known as the "Year 2038 problem" or "Y2K38 problem"). This change affects the `FROM_UNIXTIME()` and `UNIX_TIMESTAMP()` functions. You might need to change application code that interacts with these functions, change the types of columns that store the return values, or add `CAST()` calls to SQL statements that call these functions.

Usage notes:

The format string accepts the variations allowed for the `TIMESTAMP` data type: date plus time, date by itself, time by itself, and optional fractional seconds for the time. See TIMESTAMP Data Type on page 130 for details.

Currently, the format string is case-sensitive, especially to distinguish `m` for minutes and `M` for months. In Impala 1.3 and later, you can switch the order of elements, use alternative separator characters, and use a different number of placeholders for each unit. Adding more instances of `y`, `d`, `H`, and so on produces output strings zero-padded to the requested number of characters. The exception is `M` for months, where `M` produces a non-padded value such as 3, `MM` produces a zero-padded value such as `03`, `MMM` produces an abbreviated month name such as `Mar`, and sequences of 4 or more `M` are not allowed. A date string including all fields could be `'yyyy-MM-dd HH:mm:ss.SSSSSS'`, `'dd/MM/yyyy HH:mm:ss.SSSSSS'`, `'MMM dd, yyyy HH.mm.ss (SSSSSS)'` or other combinations of placeholders and separator characters.

The way this function deals with time zones when converting to or from `TIMESTAMP` values is affected by the `--use_local_tz_for_unix_timestamp_conversions` startup flag for the `impalad` daemon. See TIMESTAMP Data Type on page 130 for details about how Impala handles time zone considerations for the `TIMESTAMP` data type.



Note:

The more flexible format strings allowed with the built-in functions do not change the rules about using `CAST()` to convert from a string to a `TIMESTAMP` value. Strings being converted through `CAST()` must still have the elements in the specified order and use the specified delimiter characters, as described in TIMESTAMP Data Type on page 130.

Examples:

```
SELECT FROM_UNIXTIME(1392394861,'yyyy-MM-dd HH:mm:ss.SSSS');
+---------------------------------------------------+
| from_unixtime(1392394861, 'yyyy-mm-dd hh:mm:ss.ssss') |
+---------------------------------------------------+
| 2014-02-14 16:21:01.0000                          |
+---------------------------------------------------+

SELECT FROM_UNIXTIME(1392394861,'HH:mm:ss.SSSS');
+-------------------------------------------+
| from_unixtime(1392394861, 'hh:mm:ss.ssss') |
+-------------------------------------------+
| 16:21:01.0000                             |
+-------------------------------------------+
```

`UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` are often used in combination to convert a `TIMESTAMP` value into a particular string format. For example:

```
SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(NOW() + interval 3 days),
  'yyyy/MM/dd HH:mm') AS yyyy_mm_dd_hh_mm;
+------------------+
| yyyy_mm_dd_hh_mm |
```

```
+------------------+
| 2016/06/03 11:38 |
+------------------+
```

FROM_UTC_TIMESTAMP(TIMESTAMP timestamp, STRING timezone)

Purpose: Converts a specified UTC timestamp value into the appropriate value for a specified time zone.

Return type: `TIMESTAMP`

Usage notes: Often used to translate UTC time zone data stored in a table back to the local date and time for reporting. The opposite of the `TO_UTC_TIMESTAMP()` function.

To determine the time zone of the server you are connected to, in CDH 5.5 / Impala 2.3 and higher you can call the `timeofday()` function, which includes the time zone specifier in its return value. Remember that with cloud computing, the server you interact with might be in a different time zone than you are, or different sessions might connect to servers in different time zones, or a cluster might include servers in more than one time zone.

Examples:

See discussion of time zones in TIMESTAMP Data Type on page 130 for information about using this function for conversions between the local time zone and UTC.

The following example shows how when `TIMESTAMP` values representing the UTC time zone are stored in a table, a query can display the equivalent local date and time for a different time zone.

```
with t1 as (select cast('2016-06-02 16:25:36.116143000' as timestamp) as utc_datetime)
  select utc_datetime as 'Date/time in Greenwich UK',
    from_utc_timestamp(utc_datetime, 'PDT')
      as 'Equivalent in California USA'
  from t1;
+-------------------------------+-------------------------------+
| date/time in greenwich uk     | equivalent in california usa  |
+-------------------------------+-------------------------------+
| 2016-06-02 16:25:36.116143000 | 2016-06-02 09:25:36.116143000 |
+-------------------------------+-------------------------------+
```

The following example shows that for a date and time when daylight savings is in effect (PDT), the UTC time is 7 hours ahead of the local California time; while when daylight savings is not in effect (PST), the UTC time is 8 hours ahead of the local California time.

```
select now() as local_datetime,
  to_utc_timestamp(now(), 'PDT') as utc_datetime;
+-------------------------------+-------------------------------+
| local_datetime                | utc_datetime                  |
+-------------------------------+-------------------------------+
| 2016-05-31 11:50:02.316883000 | 2016-05-31 18:50:02.316883000 |
+-------------------------------+-------------------------------+

select '2016-01-05' as local_datetime,
  to_utc_timestamp('2016-01-05', 'PST') as utc_datetime;
+----------------+---------------------+
| local_datetime | utc_datetime        |
+----------------+---------------------+
| 2016-01-05     | 2016-01-05 08:00:00 |
+----------------+---------------------+
```

HOUR(TIMESTAMP date)

Purpose: Returns the hour field from a `TIMESTAMP` field.

Return type: `INT`

Examples:

```
select now() as right_now, hour(now()) as current_hour;
+-------------------------------+--------------+
| right_now                     | current_hour |
+-------------------------------+--------------+
| 2016-06-01 14:14:12.472846000 | 14           |
+-------------------------------+--------------+

select now() + interval 12 hours as 12_hours_from_now,
  hour(now() + interval 12 hours) as hour_in_12_hours;
+-------------------------------+-------------------+
| 12_hours_from_now             | hour_in_12_hours  |
+-------------------------------+-------------------+
| 2016-06-02 02:15:32.454750000 | 2                 |
+-------------------------------+-------------------+
```

HOURS_ADD(TIMESTAMP date, INT hours), HOURS_ADD(TIMESTAMP date, BIGINT hours)

Purpose: Returns the specified date and time plus some number of hours.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now,
  hours_add(now(), 12) as in_12_hours;
+-------------------------------+-------------------------------+
| right_now                     | in_12_hours                   |
+-------------------------------+-------------------------------+
| 2016-06-01 14:19:48.948107000 | 2016-06-02 02:19:48.948107000 |
+-------------------------------+-------------------------------+
```

HOURS_SUB(TIMESTAMP date, INT hours), HOURS_SUB(TIMESTAMP date, BIGINT hours)

Purpose: Returns the specified date and time minus some number of hours.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now,
  hours_sub(now(), 18) as 18_hours_ago;
+-------------------------------+-------------------------------+
| right_now                     | 18_hours_ago                  |
+-------------------------------+-------------------------------+
| 2016-06-01 14:23:13.868150000 | 2016-05-31 20:23:13.868150000 |
+-------------------------------+-------------------------------+
```

INT_MONTHS_BETWEEN(TIMESTAMP newer, TIMESTAMP older)

Purpose: Returns the number of months between the date portions of two `TIMESTAMP` values, as an `INT` representing only the full months that passed.

Return type: `INT`

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes:

Typically used in business contexts, for example to determine whether a specified number of months have passed or whether some end-of-month deadline was reached.

The method of determining the number of elapsed months includes some special handling of months with different numbers of days that creates edge cases for dates between the 28th and 31st days of certain months. See `MONTHS_BETWEEN()` for details. The `INT_MOINTHS_BETWEEN()` result is essentially the `FLOOR()` of the `MONTHS_BETWEEN()` result.

If either value is NULL, which could happen for example when converting a nonexistent date string such as '2015-02-29' to a TIMESTAMP, the result is also NULL.

If the first argument represents an earlier time than the second argument, the result is negative.

Examples:

```
/* Less than a full month = 0. */
select int_months_between('2015-02-28', '2015-01-29');
+-----------------------------------------------+
| int_months_between('2015-02-28', '2015-01-29') |
+-----------------------------------------------+
| 0                                             |
+-----------------------------------------------+

/* Last day of month to last day of next month = 1. */
select int_months_between('2015-02-28', '2015-01-31');
+-----------------------------------------------+
| int_months_between('2015-02-28', '2015-01-31') |
+-----------------------------------------------+
| 1                                             |
+-----------------------------------------------+

/* Slightly less than 2 months = 1. */
select int_months_between('2015-03-28', '2015-01-31');
+-----------------------------------------------+
| int_months_between('2015-03-28', '2015-01-31') |
+-----------------------------------------------+
| 1                                             |
+-----------------------------------------------+

/* 2 full months (identical days of the month) = 2. */
select int_months_between('2015-03-31', '2015-01-31');
+-----------------------------------------------+
| int_months_between('2015-03-31', '2015-01-31') |
+-----------------------------------------------+
| 2                                             |
+-----------------------------------------------+

/* Last day of month to last day of month-after-next = 2. */
select int_months_between('2015-03-31', '2015-01-30');
+-----------------------------------------------+
| int_months_between('2015-03-31', '2015-01-30') |
+-----------------------------------------------+
| 2                                             |
+-----------------------------------------------+
```

LAST_DAY(TIMESTAMP t)

Purpose: Returns a TIMESTAMP corresponding to the beginning of the last calendar day in the same month as the TIMESTAMP argument.

Return type: TIMESTAMP

Added in: CDH 5.12.0 / Impala 2.9.0

Usage notes:

If the input argument does not represent a valid Impala TIMESTAMP including both date and time portions, the function returns NULL. For example, if the input argument is a string that cannot be implicitly cast to TIMESTAMP, does not include a date portion, or is out of the allowed range for Impala TIMESTAMP values, the function returns NULL.

Examples:

The following example shows how to examine the current date, and dates around the end of the month, as TIMESTAMP values with any time portion removed:

```
select
    now() as right_now
```

```
  , trunc(now(),'dd') as today
  , last_day(now()) as last_day_of_month
  , last_day(now()) + interval 1 day as first_of_next_month;
+-------------------------------+-------------------+-------------------+--------------------+
| right_now                     | today             | last_day_of_month |
first_of_next_month |
+-------------------------------+-------------------+-------------------+--------------------+
| 2017-08-15 15:07:58.823812000 | 2017-08-15 00:00:00 | 2017-08-31 00:00:00 | 2017-09-01
 00:00:00 |
+-------------------------------+-------------------+-------------------+--------------------+
```

The following example shows how to examine the current date and dates around the end of the month as integers representing the day of the month:

```
select
    now() as right_now
  , dayofmonth(now()) as day
  , extract(day from now()) as also_day
  , dayofmonth(last_day(now())) as last_day
  , extract(day from last_day(now())) as also_last_day;
+-------------------------------+-----+----------+----------+---------------+
| right_now                     | day | also_day | last_day | also_last_day |
+-------------------------------+-----+----------+----------+---------------+
| 2017-08-15 15:07:59.417755000 | 15  | 15       | 31       | 31            |
+-------------------------------+-----+----------+----------+---------------+
```

MICROSECONDS_ADD(TIMESTAMP date, INT microseconds), MICROSECONDS_ADD(TIMESTAMP date, BIGINT microseconds)

Purpose: Returns the specified date and time plus some number of microseconds.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now,
  microseconds_add(now(), 500000) as half_a_second_from_now;
+-------------------------------+-------------------------------+
| right_now                     | half_a_second_from_now        |
+-------------------------------+-------------------------------+
| 2016-06-01 14:25:11.455051000 | 2016-06-01 14:25:11.955051000 |
+-------------------------------+-------------------------------+
```

MICROSECONDS_SUB(TIMESTAMP date, INT microseconds), MICROSECONDS_SUB(TIMESTAMP date, BIGINT microseconds)

Purpose: Returns the specified date and time minus some number of microseconds.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now,
  microseconds_sub(now(), 500000) as half_a_second_ago;
+-------------------------------+-------------------------------+
| right_now                     | half_a_second_ago             |
+-------------------------------+-------------------------------+
| 2016-06-01 14:26:16.509990000 | 2016-06-01 14:26:16.009990000 |
+-------------------------------+-------------------------------+
```

MILLISECOND(TIMESTAMP t)

Purpose: Returns the millisecond portion of a `t` value.

Return type: `INT`

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

The millisecond value is truncated, not rounded, if the TIMESTAMP value contains more than 3 significant digits to the right of the decimal point.

Examples:

```
252.4 milliseconds truncated to 252.

select now(), millisecond(now());
+-------------------------------+--------------------+
| now()                         | millisecond(now()) |
+-------------------------------+--------------------+
| 2016-03-14 22:30:25.252400000 | 252                |
+-------------------------------+--------------------+

761.767 milliseconds truncated to 761.

select now(), millisecond(now());
+-------------------------------+--------------------+
| now()                         | millisecond(now()) |
+-------------------------------+--------------------+
| 2016-03-14 22:30:58.761767000 | 761                |
+-------------------------------+--------------------+
```

MILLISECONDS_ADD(TIMESTAMP date, INT milliseconds), MILLISECONDS_ADD(TIMESTAMP date, BIGINT milliseconds)

Purpose: Returns the specified date and time plus some number of milliseconds.

Return type: TIMESTAMP

Examples:

```
select now() as right_now,
  milliseconds_add(now(), 1500) as 1_point_5_seconds_from_now;
+-------------------------------+-------------------------------+
| right_now                     | 1_point_5_seconds_from_now    |
+-------------------------------+-------------------------------+
| 2016-06-01 14:30:30.067366000 | 2016-06-01 14:30:31.567366000 |
+-------------------------------+-------------------------------+
```

MILLISECONDS_SUB(TIMESTAMP date, INT milliseconds), MILLISECONDS_SUB(TIMESTAMP date, BIGINT milliseconds)

Purpose: Returns the specified date and time minus some number of milliseconds.

Return type: TIMESTAMP

Examples:

```
select now() as right_now,
  milliseconds_sub(now(), 1500) as 1_point_5_seconds_ago;
+-------------------------------+-------------------------------+
| right_now                     | 1_point_5_seconds_ago         |
+-------------------------------+-------------------------------+
| 2016-06-01 14:30:53.467140000 | 2016-06-01 14:30:51.967140000 |
+-------------------------------+-------------------------------+
```

MINUTE(TIMESTAMP date)

Purpose: Returns the minute field from a TIMESTAMP value.

Return type: INT

Examples:

```
select now() as right_now, minute(now()) as current_minute;
+-------------------------------+----------------+
| right_now                     | current_minute |
+-------------------------------+----------------+
| 2016-06-01 14:34:08.051702000 | 34             |
+-------------------------------+----------------+
```

MINUTES_ADD(TIMESTAMP date, INT minutes), MINUTES_ADD(TIMESTAMP date, BIGINT minutes)

Purpose: Returns the specified date and time plus some number of minutes.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now, minutes_add(now(), 90) as 90_minutes_from_now;
+-------------------------------+-------------------------------+
| right_now                     | 90_minutes_from_now           |
+-------------------------------+-------------------------------+
| 2016-06-01 14:36:04.887095000 | 2016-06-01 16:06:04.887095000 |
+-------------------------------+-------------------------------+
```

MINUTES_SUB(TIMESTAMP date, INT minutes), MINUTES_SUB(TIMESTAMP date, BIGINT minutes)

Purpose: Returns the specified date and time minus some number of minutes.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now, minutes_sub(now(), 90) as 90_minutes_ago;
+-------------------------------+-------------------------------+
| right_now                     | 90_minutes_ago                |
+-------------------------------+-------------------------------+
| 2016-06-01 14:36:32.643061000 | 2016-06-01 13:06:32.643061000 |
+-------------------------------+-------------------------------+
```

MONTH(TIMESTAMP date)

Purpose: Returns the month field, represented as an integer, from the date portion of a `TIMESTAMP`.

Return type: `INT`

Examples:

```
select now() as right_now, month(now()) as current_month;
+-------------------------------+---------------+
| right_now                     | current_month |
+-------------------------------+---------------+
| 2016-06-01 14:43:37.141542000 | 6             |
+-------------------------------+---------------+
```

MONTHNAME(TIMESTAMP date)

Purpose: Returns the month field from a `TIMESTAMP` value, converted to the string corresponding to that month name.

Return type: `STRING`

MONTHS_ADD(TIMESTAMP date, INT months), MONTHS_ADD(TIMESTAMP date, BIGINT months)

Purpose: Returns the specified date and time plus some number of months.

Return type: `TIMESTAMP`

Examples:

The following example shows the effects of adding some number of months to a TIMESTAMP value, using both the months_add() function and its add_months() alias. These examples use trunc() to strip off the time portion and leave just the date.

```
with t1 as (select trunc(now(), 'dd') as today)
  select today, months_add(today,1) as next_month from t1;
+---------------------+---------------------+
| today               | next_month          |
+---------------------+---------------------+
| 2016-05-19 00:00:00 | 2016-06-19 00:00:00 |
+---------------------+---------------------+

with t1 as (select trunc(now(), 'dd') as today)
  select today, add_months(today,1) as next_month from t1;
+---------------------+---------------------+
| today               | next_month          |
+---------------------+---------------------+
| 2016-05-19 00:00:00 | 2016-06-19 00:00:00 |
+---------------------+---------------------+
```

The following examples show how if months_add() would return a nonexistent date, due to different months having different numbers of days, the function returns a TIMESTAMP from the last day of the relevant month. For example, adding one month to January 31 produces a date of February 29th in the year 2016 (a leap year), and February 28th in the year 2015 (a non-leap year).

```
with t1 as (select cast('2016-01-31' as timestamp) as jan_31)
  select jan_31, months_add(jan_31,1) as feb_31 from t1;
+---------------------+---------------------+
| jan_31              | feb_31              |
+---------------------+---------------------+
| 2016-01-31 00:00:00 | 2016-02-29 00:00:00 |
+---------------------+---------------------+

with t1 as (select cast('2015-01-31' as timestamp) as jan_31)
  select jan_31, months_add(jan_31,1) as feb_31 from t1;
+---------------------+---------------------+
| jan_31              | feb_31              |
+---------------------+---------------------+
| 2015-01-31 00:00:00 | 2015-02-28 00:00:00 |
+---------------------+---------------------+
```

MONTHS_BETWEEN(TIMESTAMP newer, TIMESTAMP older)

Purpose: Returns the number of months between the date portions of two TIMESTAMP values. Can include a fractional part representing extra days in addition to the full months between the dates. The fractional component is computed by dividing the difference in days by 31 (regardless of the month).

Return type: DOUBLE

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes:

Typically used in business contexts, for example to determine whether a specified number of months have passed or whether some end-of-month deadline was reached.

If the only consideration is the number of full months and any fractional value is not significant, use int_months_between() instead.

The method of determining the number of elapsed months includes some special handling of months with different numbers of days that creates edge cases for dates between the 28th and 31st days of certain months.

If either value is NULL, which could happen for example when converting a nonexistent date string such as '2015-02-29' to a TIMESTAMP, the result is also NULL.

If the first argument represents an earlier time than the second argument, the result is negative.

Examples:

The following examples show how dates that are on the same day of the month are considered to be exactly N months apart, even if the months have different numbers of days.

```
select months_between('2015-02-28', '2015-01-28');
+-----------------------------------------+
| months_between('2015-02-28', '2015-01-28') |
+-----------------------------------------+
| 1                                       |
+-----------------------------------------+

select months_between(now(), now() + interval 1 month);
+-------------------------------------------------+
| months_between(now(), now() + interval 1 month) |
+-------------------------------------------------+
| -1                                              |
+-------------------------------------------------+

select months_between(now() + interval 1 year, now());
+-------------------------------------------------+
| months_between(now() + interval 1 year, now()) |
+-------------------------------------------------+
| 12                                              |
+-------------------------------------------------+
```

The following examples show how dates that are on the last day of the month are considered to be exactly N months apart, even if the months have different numbers of days. For example, from January 28th to February 28th is exactly one month because the day of the month is identical; January 31st to February 28th is exactly one month because in both cases it is the last day of the month; but January 29th or 30th to February 28th is considered a fractional month.

```
select months_between('2015-02-28', '2015-01-31');
+-----------------------------------------+
| months_between('2015-02-28', '2015-01-31') |
+-----------------------------------------+
| 1                                       |
+-----------------------------------------+

select months_between('2015-02-28', '2015-01-29');
+-----------------------------------------+
| months_between('2015-02-28', '2015-01-29') |
+-----------------------------------------+
| 0.967741935483871                       |
+-----------------------------------------+

select months_between('2015-02-28', '2015-01-30');;
+-----------------------------------------+
| months_between('2015-02-28', '2015-01-30') |
+-----------------------------------------+
| 0.935483870967742                       |
+-----------------------------------------+
```

The following examples show how dates that are not a precise number of months apart result in a fractional return value.

```
select months_between('2015-03-01', '2015-01-28');
+-----------------------------------------+
| months_between('2015-03-01', '2015-01-28') |
+-----------------------------------------+
| 1.129032258064516                       |
+-----------------------------------------+

select months_between('2015-03-01', '2015-02-28');
+-----------------------------------------+
| months_between('2015-03-01', '2015-02-28') |
+-----------------------------------------+
| 0.1290322580645161                      |
+-----------------------------------------+
```

```
select months_between('2015-06-02', '2015-05-29');
+-----------------------------------------+
| months_between('2015-06-02', '2015-05-29') |
+-----------------------------------------+
| 0.1290322580645161                      |
+-----------------------------------------+

select months_between('2015-03-01', '2015-01-25');
+-----------------------------------------+
| months_between('2015-03-01', '2015-01-25') |
+-----------------------------------------+
| 1.225806451612903                       |
+-----------------------------------------+

select months_between('2015-03-01', '2015-02-25');
+-----------------------------------------+
| months_between('2015-03-01', '2015-02-25') |
+-----------------------------------------+
| 0.2258064516129032                      |
+-----------------------------------------+

select months_between('2015-02-28', '2015-02-01');
+-----------------------------------------+
| months_between('2015-02-28', '2015-02-01') |
+-----------------------------------------+
| 0.8709677419354839                      |
+-----------------------------------------+

select months_between('2015-03-28', '2015-03-01');
+-----------------------------------------+
| months_between('2015-03-28', '2015-03-01') |
+-----------------------------------------+
| 0.8709677419354839                      |
+-----------------------------------------+
```

The following examples show how the time portion of the `TIMESTAMP` values are irrelevant for calculating the month interval. Even the fractional part of the result only depends on the number of full days between the argument values, regardless of the time portion.

```
select months_between('2015-05-28 23:00:00', '2015-04-28 11:45:00');
+-------------------------------------------------------------+
| months_between('2015-05-28 23:00:00', '2015-04-28 11:45:00') |
+-------------------------------------------------------------+
| 1                                                           |
+-------------------------------------------------------------+

select months_between('2015-03-28', '2015-03-01');
+-----------------------------------------+
| months_between('2015-03-28', '2015-03-01') |
+-----------------------------------------+
| 0.8709677419354839                      |
+-----------------------------------------+

select months_between('2015-03-28 23:00:00', '2015-03-01 11:45:00');
+-------------------------------------------------------------+
| months_between('2015-03-28 23:00:00', '2015-03-01 11:45:00') |
+-------------------------------------------------------------+
| 0.8709677419354839                                          |
+-------------------------------------------------------------+
```

MONTHS_SUB(TIMESTAMP date, INT months), MONTHS_SUB(TIMESTAMP date, BIGINT months)

Purpose: Returns the specified date and time minus some number of months.

Return type: `TIMESTAMP`

Examples:

```
with t1 as (select trunc(now(), 'dd') as today)
```

```
  select today, months_sub(today,1) as last_month from t1;
+---------------------+---------------------+
| today               | last_month          |
+---------------------+---------------------+
| 2016-06-01 00:00:00 | 2016-05-01 00:00:00 |
+---------------------+---------------------+
```

NANOSECONDS_ADD(TIMESTAMP date, INT nanoseconds), NANOSECONDS_ADD(TIMESTAMP date, BIGINT nanoseconds)

Purpose: Returns the specified date and time plus some number of nanoseconds.

Return type: `TIMESTAMP`

Kudu considerations:

The nanosecond portion of an Impala `TIMESTAMP` value is rounded to the nearest microsecond when that value is stored in a Kudu table.

Examples:

```
select now() as right_now, nanoseconds_add(now(), 1) as 1_nanosecond_later;
+-------------------------------+-------------------------------+
| right_now                     | 1_nanosecond_later            |
+-------------------------------+-------------------------------+
| 2016-06-01 15:42:00.361026000 | 2016-06-01 15:42:00.361026001 |
+-------------------------------+-------------------------------+

-- 1 billion nanoseconds = 1 second.
select now() as right_now, nanoseconds_add(now(), 1e9) as 1_second_later;
+-------------------------------+-------------------------------+
| right_now                     | 1_second_later                |
+-------------------------------+-------------------------------+
| 2016-06-01 15:42:52.926706000 | 2016-06-01 15:42:53.926706000 |
+-------------------------------+-------------------------------+
```

NANOSECONDS_SUB(TIMESTAMP date, INT nanoseconds), NANOSECONDS_SUB(TIMESTAMP date, BIGINT nanoseconds)

Purpose: Returns the specified date and time minus some number of nanoseconds.

Return type: `TIMESTAMP`

Kudu considerations:

The nanosecond portion of an Impala `TIMESTAMP` value is rounded to the nearest microsecond when that value is stored in a Kudu table.

```
select now() as right_now, nanoseconds_sub(now(), 1) as 1_nanosecond_earlier;
+-------------------------------+-------------------------------+
| right_now                     | 1_nanosecond_earlier          |
+-------------------------------+-------------------------------+
| 2016-06-01 15:44:14.355837000 | 2016-06-01 15:44:14.355836999 |
+-------------------------------+-------------------------------+

-- 1 billion nanoseconds = 1 second.
select now() as right_now, nanoseconds_sub(now(), 1e9) as 1_second_earlier;
+-------------------------------+-------------------------------+
| right_now                     | 1_second_earlier              |
+-------------------------------+-------------------------------+
| 2016-06-01 15:44:54.474929000 | 2016-06-01 15:44:53.474929000 |
+-------------------------------+-------------------------------+
```

NEXT_DAY(TIMESTAMP date, STRING weekday)

Purpose: Returns the date of the *weekday* that follows the specified *date*.

Return type: `TIMESTAMP`

Usage notes:

The *weekday* parameter is case-insensitive. The following values are accepted for *weekday*: `"Sunday"/"Sun"`, `"Monday"/"Mon"`, `"Tuesday"/"Tue"`, `"Wednesday"/"Wed"`, `"Thursday"/"Thu"`, `"Friday"/"Fri"`, `"Saturday"/"Sat"`

Calling the function with the current date and weekday returns the date that is one week later.

Examples:

```
select next_day('2013-12-25','Saturday');
-- Returns '2013-12-28 00:00:00' the first Saturday after December 25, 2013.
select next_day(to_timestamp('08-1987-21', 'mm-yyyy-dd'), 'Friday');
-- Returns '1987-08-28 00:00:00' the first Friday after August 28, 1987.

select next_day(now(), 'Thu');
-- Executed on 2018-07-12, the function returns '2018-07-13 00:00:00', one week
-- after the current date.
```

NOW()

Purpose: Returns the current date and time (in the local time zone) as a `TIMESTAMP` value.

Return type: `TIMESTAMP`

Usage notes:

To find a date/time value in the future or the past relative to the current date and time, add or subtract an `INTERVAL` expression to the return value of `now()`. See TIMESTAMP Data Type on page 130 for examples.

To produce a `TIMESTAMP` representing the current date and time that can be shared or stored without interoperability problems due to time zone differences, use the `to_utc_timestamp()` function and specify the time zone of the server. When `TIMESTAMP` data is stored in UTC form, any application that queries those values can convert them to the appropriate local time zone by calling the inverse function, `from_utc_timestamp()`.

To determine the time zone of the server you are connected to, in CDH 5.5 / Impala 2.3 and higher you can call the `timeofday()` function, which includes the time zone specifier in its return value. Remember that with cloud computing, the server you interact with might be in a different time zone than you are, or different sessions might connect to servers in different time zones, or a cluster might include servers in more than one time zone.

Any references to the `now()` function are evaluated at the start of a query. All calls to `now()` within the same query return the same value, and the value does not depend on how long the query takes.

Examples:

```
select now() as 'Current time in California USA',
  to_utc_timestamp(now(), 'PDT') as 'Current time in Greenwich UK';
+--------------------------------+-------------------------------+
| current time in california usa | current time in greenwich uk  |
+--------------------------------+-------------------------------+
| 2016-06-01 15:52:08.980072000  | 2016-06-01 22:52:08.980072000 |
+--------------------------------+-------------------------------+

select now() as right_now,
  now() + interval 1 day as tomorrow,
  now() + interval 1 week - interval 3 hours as almost_a_week_from_now;
+-------------------------------+-------------------------------+-------------------------------+
| right_now                     | tomorrow                      | almost_a_week_from_now
      |
+-------------------------------+-------------------------------+-------------------------------+
| 2016-06-01 15:55:39.671690000 | 2016-06-02 15:55:39.671690000 | 2016-06-08
12:55:39.671690000 |
+-------------------------------+-------------------------------+-------------------------------+
```

QUARTER(TIMESTAMP date)

Purpose: Returns the quarter in the input TIMESTAMP expression as an integer value, 1, 2, 3, or 4, where 1 represents January 1 through March 31.

Return type: INT

SECOND(TIMESTAMP date)

Purpose: Returns the second field from a TIMESTAMP value.

Return type: INT

Examples:

```
select now() as right_now,
  second(now()) as seconds_in_current_minute;
+-------------------------------+---------------------------+
| right_now                     | seconds_in_current_minute |
+-------------------------------+---------------------------+
| 2016-06-01 16:03:57.006603000 | 57                        |
+-------------------------------+---------------------------+
```

SECONDS_ADD(TIMESTAMP date, INT seconds), SECONDS_ADD(TIMESTAMP date, BIGINT seconds)

Purpose: Returns the specified date and time plus some number of seconds.

Return type: TIMESTAMP

Examples:

```
select now() as right_now,
  seconds_add(now(), 10) as 10_seconds_from_now;
+-------------------------------+-------------------------------+
| right_now                     | 10_seconds_from_now           |
+-------------------------------+-------------------------------+
| 2016-06-01 16:05:21.573935000 | 2016-06-01 16:05:31.573935000 |
+-------------------------------+-------------------------------+
```

SECONDS_SUB(TIMESTAMP date, INT seconds), SECONDS_SUB(TIMESTAMP date, BIGINT seconds)

Purpose: Returns the specified date and time minus some number of seconds.

Return type: TIMESTAMP

Examples:

```
select now() as right_now,
  seconds_sub(now(), 10) as 10_seconds_ago;
+-------------------------------+-------------------------------+
| right_now                     | 10_seconds_ago                |
+-------------------------------+-------------------------------+
| 2016-06-01 16:06:03.467931000 | 2016-06-01 16:05:53.467931000 |
+-------------------------------+-------------------------------+
```

SUBDATE(TIMESTAMP startdate, INT days), SUBDATE(TIMESTAMP startdate, BIGINT days)

Purpose: Subtracts a specified number of days from a TIMESTAMP value. Similar to DATE_SUB( ), but starts with an actual TIMESTAMP value instead of a string that is converted to a TIMESTAMP.

Return type: TIMESTAMP

Examples:

The following examples show how to subtract a number of days from a TIMESTAMP. The number of days can also be negative, which gives the same effect as the ADDDATE() function.

```
select now() as right_now, subdate(now(), 30) as now_minus_30;
+-------------------------------+-------------------------------+
| right_now                     | now_minus_30                  |
+-------------------------------+-------------------------------+
| 2016-05-20 11:00:15.084991000 | 2016-04-20 11:00:15.084991000 |
+-------------------------------+-------------------------------+

select now() as right_now, subdate(now(), -15) as now_plus_15;
+-------------------------------+-------------------------------+
| right_now                     | now_plus_15                   |
+-------------------------------+-------------------------------+
| 2016-05-20 11:00:44.766091000 | 2016-06-04 11:00:44.766091000 |
+-------------------------------+-------------------------------+
```

TIMEOFDAY()

Purpose: Returns a string representation of the current date and time, according to the time of the local system, including any time zone designation.

Return type: STRING

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes: The result value represents similar information as the now() function, only as a STRING type and with somewhat different formatting. For example, the day of the week and the time zone identifier are included. This function is intended primarily for compatibility with SQL code from other systems that also have a timeofday() function. Prefer to use now() if practical for any new Impala code.

Examples:

The following examples show the format of the TIMEOFDAY() return value, illustrate how that value is represented as a STRING that you can manipulate with string processing functions, and how the format compares with the return value from the NOW() function.

```
/* Date and time fields in a STRING return value. */
select timeofday();
+-------------------------------+
| timeofday()                   |
+-------------------------------+
| Tue Sep 01 15:13:18 2015 PDT |
+-------------------------------+

/* The return value can be processed by other string functions. */
select upper(timeofday());
+-----------------------------+
| upper(timeofday())          |
+-----------------------------+
| TUE SEP 01 15:13:38 2015 PDT |
+-----------------------------+

/* The TIMEOFDAY() result is formatted differently than NOW(). NOW() returns a TIMESTAMP.
 */
select now(), timeofday();
+-------------------------------+-------------------------------+
| now()                         | timeofday()                   |
+-------------------------------+-------------------------------+
| 2015-09-01 15:15:25.930021000 | Tue Sep 01 15:15:25 2015 PDT |
+-------------------------------+-------------------------------+

/* You can strip out the time zone field to use in calls to from_utc_timestamp(). */
select regexp_replace(timeofday(), '.* ([A-Z]+)$', '\\1') as current_timezone;
+------------------+
| current_timezone |
+------------------+
| PDT              |
+------------------+
```

TIMESTAMP_CMP(TIMESTAMP t1, TIMESTAMP t2)

Purpose: Tests if one `TIMESTAMP` value is newer than, older than, or identical to another `TIMESTAMP`

Return type: `INT` (either -1, 0, 1, or `NULL`)

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes:

Usage notes: A comparison function for `TIMESTAMP` values that only tests whether the date and time increases, decreases, or stays the same. Similar to the `SIGN()` function for numeric values.

Examples:

The following examples show all the possible return values for `timestamp_cmp()`. If the first argument represents a later point in time than the second argument, the result is 1. The amount of the difference is irrelevant, only the fact that one argument is greater than or less than the other. If the first argument represents an earlier point in time than the second argument, the result is -1. If the first and second arguments represent identical points in time, the result is 0. If either argument is `NULL`, the result is `NULL`.

```
/* First argument 'later' than second argument. */

select timestamp_cmp(now() + interval 70 minutes, now())
  as now_vs_in_70_minutes;
+----------------------+
| now_vs_in_70_minutes |
+----------------------+
| 1                    |
+----------------------+

select timestamp_cmp(now() +
  interval 3 days +
  interval 5 hours, now())
  as now_vs_days_from_now;
+----------------------+
| now_vs_days_from_now |
+----------------------+
| 1                    |
+----------------------+

/* First argument 'earlier' than second argument. */
select timestamp_cmp(now(), now() + interval 2 hours)
  as now_vs_2_hours_ago;
+--------------------+
| now_vs_2_hours_ago |
+--------------------+
| -1                 |
+--------------------+

/* Both arguments represent the same point in time. */

select timestamp_cmp(now(), now())
  as identical_timestamps;
+----------------------+
| identical_timestamps |
+----------------------+
| 0                    |
+----------------------+

select timestamp_cmp
(
  now() + interval 1 hour,
  now() + interval 60 minutes
) as equivalent_date_times;
+----------------------+
| equivalent_date_times |
+----------------------+
| 0                    |
+----------------------+

/* Either argument NULL. */
```

```
select timestamp_cmp(now(), null)
  as now_vs_null;
+-------------+
| now_vs_null |
+-------------+
| NULL        |
+-------------+
```

TO_DATE(TIMESTAMP timestamp)

Purpose: Returns a string representation of the date field from a timestamp value.

Return type: `STRING`

Examples:

```
select now() as right_now,
  concat('The date today is ',to_date(now()),'.') as date_announcement;
+-------------------------------+-------------------------------+
| right_now                     | date_announcement             |
+-------------------------------+-------------------------------+
| 2016-06-01 16:30:36.890325000 | The date today is 2016-06-01. |
+-------------------------------+-------------------------------+
```

TO_TIMESTAMP(BIGINT unixtime), TO_TIMESTAMP(STRING date, STRING pattern)

Purpose: Converts an integer or string representing a date/time value into the corresponding `TIMESTAMP` value.

Return type: `TIMESTAMP`

Added in: CDH 5.5.0 / Impala 2.3.0

Usage notes:

An integer argument represents the number of seconds past the epoch (midnight on January 1, 1970). It is the converse of the `unix_timestamp( )` function, which produces a `BIGINT` representing the number of seconds past the epoch.

A string argument, plus another string argument representing the pattern, turns an arbitrary string representation of a date and time into a true `TIMESTAMP` value. The ability to parse many kinds of date and time formats allows you to deal with temporal data from diverse sources, and if desired to convert to efficient `TIMESTAMP` values during your ETL process. Using `TIMESTAMP` directly in queries and expressions lets you perform date and time calculations without the overhead of extra function calls and conversions each time you reference the applicable columns.

Examples:

The following examples demonstrate how to convert an arbitrary string representation to `TIMESTAMP` based on a pattern string:

```
select to_timestamp('Sep 25, 1984', 'MMM dd, yyyy');
+-----------------------------------------------+
| to_timestamp('sep 25, 1984', 'mmm dd, yyyy') |
+-----------------------------------------------+
| 1984-09-25 00:00:00                           |
+-----------------------------------------------+

select to_timestamp('1984/09/25', 'yyyy/MM/dd');
+---------------------------------------+
| to_timestamp('1984/09/25', 'yyyy/mm/dd') |
+---------------------------------------+
| 1984-09-25 00:00:00                   |
+---------------------------------------+
```

The following examples show how to convert a BIGINT representing seconds past epoch into a TIMESTAMP value:

```
-- One day past the epoch.
select to_timestamp(24 * 60 * 60);
+---------------------------+
| to_timestamp(24 * 60 * 60) |
+---------------------------+
| 1970-01-02 00:00:00        |
+---------------------------+

-- 60 seconds in the past.
select now() as 'current date/time',
  unix_timestamp(now()) 'now in seconds',
  to_timestamp(unix_timestamp(now()) - 60) as '60 seconds ago';
+-------------------------------+---------------+---------------------+
| current date/time             | now in seconds | 60 seconds ago      |
+-------------------------------+---------------+---------------------+
| 2017-10-01 22:03:46.885624000 | 1506895426     | 2017-10-01 22:02:46 |
+-------------------------------+---------------+---------------------+
```

TO_UTC_TIMESTAMP(TIMESTAMP, STRING timezone)

Purpose: Converts a specified timestamp value in a specified time zone into the corresponding value for the UTC time zone.

Return type: TIMESTAMP

Usage notes:

Often used in combination with the now( ) function, to translate local date and time values to the UTC time zone for consistent representation on disk. The opposite of the from_utc_timestamp( ) function.

See discussion of time zones in TIMESTAMP Data Type on page 130 for information about using this function for conversions between the local time zone and UTC.

Examples:

The simplest use of this function is to turn a local date/time value to one with the standardized UTC time zone. Because the time zone specifier is not saved as part of the Impala TIMESTAMP value, all applications that refer to such data must agree in advance which time zone the values represent. If different parts of the ETL cycle, or different instances of the application, occur in different time zones, the ideal reference point is to convert all TIMESTAMP values to UTC for storage.

```
select now() as 'Current time in California USA',
  to_utc_timestamp(now(), 'PDT') as 'Current time in Greenwich UK';
+--------------------------------+-------------------------------+
| current time in california usa | current time in greenwich uk  |
+--------------------------------+-------------------------------+
| 2016-06-01 15:52:08.980072000  | 2016-06-01 22:52:08.980072000 |
+--------------------------------+-------------------------------+
```

Once a value is converted to the UTC time zone by to_utc_timestamp( ), it can be converted back to the local time zone with from_utc_timestamp( ). You can combine these functions using different time zone identifiers to convert a TIMESTAMP between any two time zones. This example starts with a TIMESTAMP value representing Pacific Daylight Time, converts it to UTC, and converts it to the equivalent value in Eastern Daylight Time.

```
select now() as 'Current time in California USA',
  from_utc_timestamp
  (
    to_utc_timestamp(now(), 'PDT'),
    'EDT'
  ) as 'Current time in New York, USA';
+--------------------------------+-------------------------------+
| current time in california usa | current time in new york, usa |
+--------------------------------+-------------------------------+
```

```
| 2016-06-01 18:14:12.743658000  | 2016-06-01 21:14:12.743658000 |
+--------------------------------+--------------------------------+
```

TRUNC(TIMESTAMP timestamp, STRING unit)

Purpose: Strips off fields from a TIMESTAMP value.

Unit argument: The unit argument value for truncating TIMESTAMP values is case-sensitive. This argument string can be one of:

- SYYYY, YYYY, YEAR, SYEAR, YYY, YY, Y: Year.
- Q: Quarter.
- MONTH, MON, MM, RM: Month.
- WW, W: Same day of the week as the first day of the month.
- DDD, DD, J: Day.
- DAY, DY, D: Starting day of the week. (Not necessarily the current day.)
- HH, HH12, HH24: Hour. A TIMESTAMP value truncated to the hour is always represented in 24-hour notation, even for the HH12 argument string.
- MI: Minute.

Added in: The ability to truncate numeric values is new starting in CDH 5.13 / Impala 2.10.

Usage notes:

The TIMESTAMP form is typically used in GROUP BY queries to aggregate results from the same hour, day, week, month, quarter, and so on. You can also use this function in an INSERT ... SELECT into a partitioned table to divide TIMESTAMP values into the correct partition.

Because the return value is a TIMESTAMP, if you cast the result of TRUNC() to STRING, you will often see zeroed-out portions such as 00:00:00 in the time field. If you only need the individual units such as hour, day, month, or year, use the EXTRACT() function instead. If you need the individual units from a truncated TIMESTAMP value, run the TRUNCATE() function on the original value, then run EXTRACT() on the result.

The trunc() function also has a signature that applies to DOUBLE or DECIMALvalues. truncate(), trunc(), and dtrunc() are all aliased to the same function. See truncate() under Impala Mathematical Functions on page 397 for details.

Return type: TIMESTAMP

Examples:

The following example shows how the argument 'Q' returns a TIMESTAMP representing the beginning of the appropriate calendar quarter. This return value is the same for input values that could be separated by weeks or months. If you stored the trunc() result in a partition key column, the table would have four partitions per year.

```
select now() as right_now, trunc(now(), 'Q') as current_quarter;
+-------------------------------+---------------------+
| right_now                     | current_quarter     |
+-------------------------------+---------------------+
| 2016-06-01 18:32:02.097202000 | 2016-04-01 00:00:00 |
+-------------------------------+---------------------+

select now() + interval 2 weeks as 2_weeks_from_now,
  trunc(now() + interval 2 weeks, 'Q') as still_current_quarter;
+-------------------------------+----------------------+
| 2_weeks_from_now              | still_current_quarter |
+-------------------------------+----------------------+
| 2016-06-15 18:36:19.584257000 | 2016-04-01 00:00:00   |
+-------------------------------+----------------------+
```

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(STRING datetime), UNIX_TIMESTAMP(STRING datetime, STRING format), UNIX_TIMESTAMP(TIMESTAMP datetime)

Purpose: Returns a Unix time, which is a number of seconds elapsed since '1970-01-01 00:00:00' UTC. If called with no argument, the current date and time is converted to its Unix time. If called with arguments, the first argument represented as the TIMESTAMP or STRING is converted to its Unix time.

Return type: BIGINT

Usage notes:

See FROM_UNIXTIME() for details about the patterns you can use in the format string to represent the position of year, month, day, and so on in the date string. In Impala 1.3 and higher, you have more flexibility to switch the positions of elements and use different separator characters.

In CDH 5.4.3 and higher, you can include a trailing uppercase Z qualifier to indicate "Zulu" time, a synonym for UTC.

In CDH 5.5 / Impala 2.3 and higher, you can include a timezone offset specified as minutes and hours, provided you also specify the details in the format string argument. The offset is specified in the format string as a plus or minus sign followed by hh:mm, hhmm, or hh. The hh must be lowercase, to distinguish it from the HH represent hours in the actual time value. Currently, only numeric timezone offsets are allowed, not symbolic names.

In Impala 2.2.0 and higher, built-in functions that accept or return integers representing TIMESTAMP values use the BIGINT type for parameters and return values, rather than INT. This change lets the date and time functions avoid an overflow error that would otherwise occur on January 19th, 2038 (known as the "Year 2038 problem" or "Y2K38 problem"). This change affects the FROM_UNIXTIME() and UNIX_TIMESTAMP() functions. You might need to change application code that interacts with these functions, change the types of columns that store the return values, or add CAST() calls to SQL statements that call these functions.

UNIX_TIMESTAMP() and FROM_UNIXTIME() are often used in combination to convert a TIMESTAMP value into a particular string format. For example:

```
SELECT FROM_UNIXTIME(UNIX_TIMESTAMP(NOW() + interval 3 days),
  'yyyy/MM/dd HH:mm') AS yyyy_mm_dd_hh_mm;
+------------------+
| yyyy_mm_dd_hh_mm |
+------------------+
| 2016/06/03 11:38 |
+------------------+
```

The way this function deals with time zones when converting to or from TIMESTAMP values is affected by the --use_local_tz_for_unix_timestamp_conversions startup flag for the impalad daemon. See TIMESTAMP Data Type on page 130 for details about how Impala handles time zone considerations for the TIMESTAMP data type.

Examples:

The following examples show different ways of turning the same date and time into an integer value. A format string that Impala recognizes by default is interpreted as a UTC date and time. The trailing Z is a confirmation that the timezone is UTC. If the date and time string is formatted differently, a second argument specifies the position and units for each of the date and time values.

The final two examples show how to specify a timezone offset of Pacific Daylight Saving Time, which is 7 hours earlier than UTC. You can use the numeric offset -07:00 and the equivalent suffix of -hh:mm in the format string, or specify the mnemonic name for the time zone in a call to to_utc_timestamp(). This particular date and time expressed in PDT translates to a different number than the same date and time expressed in UTC.

```
-- 3 ways of expressing the same date/time in UTC and converting to an integer.

select unix_timestamp('2015-05-15 12:00:00');
+-------------------------------------+
| unix_timestamp('2015-05-15 12:00:00') |
+-------------------------------------+
| 1431691200                          |
```

```
+--------------------------------------+

select unix_timestamp('2015-05-15 12:00:00Z');
+---------------------------------------+
| unix_timestamp('2015-05-15 12:00:00z') |
+---------------------------------------+
| 1431691200                            |
+---------------------------------------+

select unix_timestamp
(
  'May 15, 2015 12:00:00',
  'MMM dd, yyyy HH:mm:ss'
) as may_15_month_day_year;
+-----------------------+
| may_15_month_day_year |
+-----------------------+
| 1431691200            |
+-----------------------+

-- 2 ways of expressing the same date and time but in a different timezone.
-- The resulting integer is different from the previous examples.

select unix_timestamp
(
  '2015-05-15 12:00:00-07:00',
  'yyyy-MM-dd HH:mm:ss-hh:mm'
) as may_15_year_month_day;
+-----------------------+
| may_15_year_month_day |
+-----------------------+
| 1431716400            |
+-----------------------+

select unix_timestamp
  (to_utc_timestamp(
    '2015-05-15 12:00:00',
    'PDT')
  ) as may_15_pdt;
+------------+
| may_15_pdt |
+------------+
| 1431716400 |
+------------+
```

UTC_TIMESTAMP()

Purpose: Returns a `TIMESTAMP` corresponding to the current date and time in the UTC time zone.

Return type: `TIMESTAMP`

Added in: CDH 5.13

Usage notes:

Similar to the `now()` or `current_timestamp()` functions, but does not use the local time zone as those functions do. Use `utc_timestamp()` to record `TIMESTAMP` values that are interoperable with servers around the world, in arbitrary time zones, without the need for additional conversion functions to standardize the time zone of each value representing a date/time.

For working with date/time values represented as integer values, you can convert back and forth between `TIMESTAMP` and `BIGINT` with the `unix_micros_to_utc_timestamp()` and `utc_to_unix_micros()` functions. The integer values represent the number of microseconds since the Unix epoch (midnight on January 1, 1970).

Examples:

The following example shows how NOW() and CURRENT_TIMESTAMP() represent the current date/time in the local time zone (in this case, UTC-7), while UTC_TIMESTAMP() represents the same date/time in the standardized UTC time zone:

```
select now(), utc_timestamp();
+-------------------------------+-------------------------------+
| now()                         | utc_timestamp()               |
+-------------------------------+-------------------------------+
| 2017-10-01 23:33:58.919688000 | 2017-10-02 06:33:58.919688000 |
+-------------------------------+-------------------------------+

select current_timestamp(), utc_timestamp();
+-------------------------------+-------------------------------+
| current_timestamp()           | utc_timestamp()               |
+-------------------------------+-------------------------------+
| 2017-10-01 23:34:07.400642000 | 2017-10-02 06:34:07.400642000 |
+-------------------------------+-------------------------------+
```

WEEK(TIMESTAMP date), WEEKOFYEAR(TIMESTAMP date)

Purpose: Returns the corresponding week (1-53) from the date portion of a TIMESTAMP.

Return type: INT

Examples:

```
select now() as right_now, weekofyear(now()) as this_week;
+-------------------------------+-----------+
| right_now                     | this_week |
+-------------------------------+-----------+
| 2016-06-01 22:40:06.763771000 | 22        |
+-------------------------------+-----------+

select now() + interval 2 weeks as in_2_weeks,
  weekofyear(now() + interval 2 weeks) as week_after_next;
+-------------------------------+-----------------+
| in_2_weeks                    | week_after_next |
+-------------------------------+-----------------+
| 2016-06-15 22:41:22.098823000 | 24              |
+-------------------------------+-----------------+
```

WEEKS_ADD(TIMESTAMP date, INT weeks), WEEKS_ADD(TIMESTAMP date, BIGINT weeks)

Purpose: Returns the specified date and time plus some number of weeks.

Return type: TIMESTAMP

Examples:

```
select now() as right_now, weeks_add(now(), 2) as week_after_next;
+-------------------------------+-------------------------------+
| right_now                     | week_after_next               |
+-------------------------------+-------------------------------+
| 2016-06-01 22:43:20.973834000 | 2016-06-15 22:43:20.973834000 |
+-------------------------------+-------------------------------+
```

WEEKS_SUB(TIMESTAMP date, INT weeks), WEEKS_SUB(TIMESTAMP date, BIGINT weeks)

Purpose: Returns the specified date and time minus some number of weeks.

Return type: TIMESTAMP

Examples:

```
select now() as right_now, weeks_sub(now(), 2) as week_before_last;
+-------------------------------+-------------------------------+
| right_now                     | week_before_last              |
```

```
+-------------------------------+-------------------------------+
| 2016-06-01 22:44:21.291913000 | 2016-05-18 22:44:21.291913000 |
+-------------------------------+-------------------------------+
```

YEAR(TIMESTAMP date)

Purpose: Returns the year field from the date portion of a `TIMESTAMP`.

Return type: `INT`

Examples:

```
select now() as right_now, year(now()) as this_year;
+-------------------------------+-----------+
| right_now                     | this_year |
+-------------------------------+-----------+
| 2016-06-01 22:46:23.647925000 | 2016      |
+-------------------------------+-----------+
```

YEARS_ADD(TIMESTAMP date, INT years), YEARS_ADD(TIMESTAMP date, BIGINT years)

Purpose: Returns the specified date and time plus some number of years.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now, years_add(now(), 1) as next_year;
+-------------------------------+-------------------------------+
| right_now                     | next_year                     |
+-------------------------------+-------------------------------+
| 2016-06-01 22:47:45.556851000 | 2017-06-01 22:47:45.556851000 |
+-------------------------------+-------------------------------+
```

The following example shows how if the equivalent date does not exist in the year of the result due to a leap year, the date is changed to the last day of the appropriate month.

```
-- Spoiler alert: there is no Feb. 29, 2017
select cast('2016-02-29' as timestamp) as feb_29_2016,
  years_add('2016-02-29', 1) as feb_29_2017;
+---------------------+---------------------+
| feb_29_2016         | feb_29_2017         |
+---------------------+---------------------+
| 2016-02-29 00:00:00 | 2017-02-28 00:00:00 |
+---------------------+---------------------+
```

YEARS_SUB(TIMESTAMP date, INT years), YEARS_SUB(TIMESTAMP date, BIGINT years)

Purpose: Returns the specified date and time minus some number of years.

Return type: `TIMESTAMP`

Examples:

```
select now() as right_now, years_sub(now(), 1) as last_year;
+-------------------------------+-------------------------------+
| right_now                     | last_year                     |
+-------------------------------+-------------------------------+
| 2016-06-01 22:48:11.851780000 | 2015-06-01 22:48:11.851780000 |
+-------------------------------+-------------------------------+
```

The following example shows how if the equivalent date does not exist in the year of the result due to a leap year, the date is changed to the last day of the appropriate month.

```
-- Spoiler alert: there is no Feb. 29, 2015
select cast('2016-02-29' as timestamp) as feb_29_2016,
  years_sub('2016-02-29', 1) as feb_29_2015;
+---------------------+---------------------+
| feb_29_2016         | feb_29_2015         |
+---------------------+---------------------+
| 2016-02-29 00:00:00 | 2015-02-28 00:00:00 |
+---------------------+---------------------+
```

Impala Conditional Functions

Impala supports the following conditional functions for testing equality, comparison operators, and nullity:

- CASE
- CASE2
- COALESCE
- DECODE
- IF
- IFNULL
- ISFALSE
- ISNOTFALSE
- ISNOTTRUE
- ISNULL
- ISTRUE
- NONNULLVALUE
- NULLIF
- NULLIFZERO
- NULLVALUE
- NVL
- NVL2
- ZEROIFNULL

CASE a WHEN b THEN c [WHEN d THEN e]... [ELSE f] END

Purpose: Compares an expression to one or more possible values, and returns a corresponding result when a match is found.

Return type: same as the initial argument value, except that integer values are promoted to `BIGINT` and floating-point values are promoted to `DOUBLE`; use `CAST()` when inserting into a smaller numeric column

Usage notes:

In this form of the `CASE` expression, the initial value `A` being evaluated for each row it typically a column reference, or an expression involving a column. This form can only compare against a set of specified values, not ranges, multi-value comparisons such as `BETWEEN` or `IN`, regular expressions, or `NULL`.

Examples:

Although this example is split across multiple lines, you can put any or all parts of a `CASE` expression on a single line, with no punctuation or other separators between the `WHEN`, `ELSE`, and `END` clauses.

```
select case x
    when 1 then 'one'
    when 2 then 'two'
    when 0 then 'zero'
    else 'out of range'
```

```
  end
    from t1;
```

CASE WHEN a THEN b [WHEN c THEN d]... [ELSE e] END

Purpose: Tests whether any of a sequence of expressions is TRUE, and returns a corresponding result for the first TRUE expression.

Return type: same as the initial argument value, except that integer values are promoted to BIGINT and floating-point values are promoted to DOUBLE; use CAST( ) when inserting into a smaller numeric column

Usage notes:

CASE expressions without an initial test value have more flexibility. For example, they can test different columns in different WHEN clauses, or use comparison operators such as BETWEEN, IN and IS NULL rather than comparing against discrete values.

CASE expressions are often the foundation of long queries that summarize and format results for easy-to-read reports. For example, you might use a CASE function call to turn values from a numeric column into category strings corresponding to integer values, or labels such as “Small”, “Medium” and “Large” based on ranges. Then subsequent parts of the query might aggregate based on the transformed values, such as how many values are classified as small, medium, or large. You can also use CASE to signal problems with out-of-bounds values, NULL values, and so on.

By using operators such as OR, IN, REGEXP, and so on in CASE expressions, you can build extensive tests and transformations into a single query. Therefore, applications that construct SQL statements often rely heavily on CASE calls in the generated SQL code.

Because this flexible form of the CASE expressions allows you to perform many comparisons and call multiple functions when evaluating each row, be careful applying elaborate CASE expressions to queries that process large amounts of data. For example, when practical, evaluate and transform values through CASE after applying operations such as aggregations that reduce the size of the result set; transform numbers to strings after performing joins with the original numeric values.

Examples:

Although this example is split across multiple lines, you can put any or all parts of a CASE expression on a single line, with no punctuation or other separators between the WHEN, ELSE, and END clauses.

```
select case
    when dayname(now()) in ('Saturday','Sunday') then 'result undefined on weekends'
    when x > y then 'x greater than y'
    when x = y then 'x and y are equal'
    when x is null or y is null then 'one of the columns is null'
    else null
  end
    from t1;
```

COALESCE(type v1, type v2, ...)

Purpose: Returns the first specified argument that is not NULL, or NULL if all arguments are NULL.

Return type: same as the initial argument value, except that integer values are promoted to BIGINT and floating-point values are promoted to DOUBLE; use CAST( ) when inserting into a smaller numeric column

DECODE(type expression, type search1, type result1 [, type search2, type result2 ...] [, type default])

Purpose: Compares the first argument, expression, to the search expressions using the IS NOT DISTINCT operator, and returns:

- The corresponding result when a match is found.
- The first corresponding result if there are more than one matching search expressions.
- The default expression if none of the search expressions matches the first argument expression.
- NULL if the final default expression is omitted and none of the search expressions matches the first argument.

Return type: Same as the first argument with the following exceptions:

- Integer values are promoted to BIGINT.
- Floating-point values are promoted to DOUBLE.
- Use CAST( ) when inserting into a smaller numeric column.

Usage notes:

- Can be used as shorthand for a CASE expression.
- The first argument, expression, and the search expressions must be of the same type or convertible types.
- The result expression can be a different type, but all result expressions must be of the same type.
- Returns a successful match if the first argument is NULL and a search expression is also NULL.
- NULL can be used as a search expression.

Examples:

The following example translates numeric day values into weekday names, such as 1 to Monday, 2 to Tuesday, etc.

```
SELECT event, DECODE(day_of_week, 1, "Monday", 2, "Tuesday", 3, "Wednesday",
  4, "Thursday", 5, "Friday", 6, "Saturday", 7, "Sunday", "Unknown day")
  FROM calendar;
```

IF(BOOLEAN condition, type ifTrue, type ifFalseOrNull)

Purpose: Tests an expression and returns a corresponding result depending on whether the result is TRUE, FALSE, or NULL.

Return type: Same as the ifTrue argument value

IFNULL(type a, type ifNull)

Purpose: Alias for the ISNULL( ) function, with the same behavior. To simplify porting SQL with vendor extensions to Impala.

Added in: Impala 1.3.0

ISFALSE(BOOLEAN expression)

Purpose: Returns TRUE if the expression is FALSE. Returns FALSE if the expression is TRUE or NULL.

Same as the IS FALSE operator.

Similar to ISNOTTRUE( ), except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Added in: CDH 5.4.0 / Impala 2.2.0

Usage notes:

In CDH 5.14 / Impala 2.11 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE( ), ISNOTTRUE( ), ISFALSE( ), and ISNOTFALSE( ).

ISNOTFALSE(BOOLEAN expression)

Purpose: Tests if a Boolean expression is not FALSE (that is, either TRUE or NULL). Returns TRUE if so. If the argument is NULL, returns TRUE.

Same as the IS NOT FALSE operator.

Similar to ISTRUE( ), except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

Added in: CDH 5.4.0 / Impala 2.2.0

Usage notes:

In CDH 5.14 / Impala 2.11 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE( ), ISNOTTRUE( ), ISFALSE( ), and ISNOTFALSE( ).

ISNOTTRUE(BOOLEAN expression)

Purpose: Tests if a Boolean expression is not TRUE (that is, either FALSE or NULL). Returns TRUE if so. If the argument is NULL, returns TRUE.

Same as the `IS NOT TRUE` operator.

Similar to `ISFALSE()`, except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Added in: CDH 5.4.0 / Impala 2.2.0

Usage notes:

In CDH 5.14 / Impala 2.11 and higher, you can use the operators `IS [NOT] TRUE` and `IS [NOT] FALSE` as equivalents for the built-in functions `ISTRUE()`, `ISNOTTRUE()`, `ISFALSE()`, and `ISNOTFALSE()`.

ISNULL(type a, type ifNull)

Purpose: Tests if an expression is NULL, and returns the expression result value if not. If the first argument is NULL, returns the second argument.

Compatibility notes: Equivalent to the `NVL()` function from Oracle Database or `IFNULL()` from MySQL. The `NVL()` and `IFNULL()` functions are also available in Impala.

Return type: Same as the first argument value

ISTRUE(BOOLEAN expression)

Purpose: Returns TRUE if the expression is TRUE. Returns FALSE if the expression is FALSE or NULL.

Same as the `IS TRUE` operator.

Similar to `ISNOTFALSE()`, except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

Added in: CDH 5.4.0 / Impala 2.2.0

Usage notes:

In CDH 5.14 / Impala 2.11 and higher, you can use the operators `IS [NOT] TRUE` and `IS [NOT] FALSE` as equivalents for the built-in functions `ISTRUE()`, `ISNOTTRUE()`, `ISFALSE()`, and `ISNOTFALSE()`.

NONNULLVALUE(type expression)

Purpose: Tests if an expression (of any type) is NULL or not. Returns FALSE if so. The converse of `nullvalue()`.

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

Added in: CDH 5.4.0 / Impala 2.2.0

NULLIF(expr1, expr2)

Purpose: Returns NULL if the two specified arguments are equal. If the specified arguments are not equal, returns the value of *expr1*. The data types of the expressions must be compatible, according to the conversion rules from Data Types on page 101. You cannot use an expression that evaluates to NULL for *expr1*; that way, you can distinguish a return value of NULL from an argument value of NULL, which would never match *expr2*.

Usage notes: This function is effectively shorthand for a CASE expression of the form:

```
CASE
  WHEN expr1 = expr2 THEN NULL
  ELSE expr1
END
```

It is commonly used in division expressions, to produce a NULL result instead of a divide-by-zero error when the divisor is equal to zero:

```
select 1.0 / nullif(c1,0) as reciprocal from t1;
```

You might also use it for compatibility with other database systems that support the same NULLIF( ) function.

Return type: same as the initial argument value, except that integer values are promoted to BIGINT and floating-point values are promoted to DOUBLE; use CAST( ) when inserting into a smaller numeric column

Added in: Impala 1.3.0

NULLIFZERO(numeric_expr)

Purpose: Returns NULL if the numeric expression evaluates to 0, otherwise returns the result of the expression.

Usage notes: Used to avoid error conditions such as divide-by-zero in numeric calculations. Serves as shorthand for a more elaborate CASE expression, to simplify porting SQL with vendor extensions to Impala.

Return type: Same type as the input argument

Added in: Impala 1.3.0

NULLVALUE(expression)

Purpose: Tests if an expression (of any type) is NULL or not. Returns TRUE if so. The converse of nonnullvalue( ).

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

Added in: CDH 5.4.0 / Impala 2.2.0

NVL(type a, type ifNull)

Purpose: Alias for the ISNULL( ) function. Returns the first argument if the first argument is not NULL. Returns the second argument if the first argument is NULL.

Equivalent to the NVL( ) function in Oracle Database or IFNULL( ) in MySQL.

Return type: Same as the first argument value

Added in: Impala 1.1

NVL2(type a, type ifNotNull, type ifNull)

Purpose: Returns the second argument, ifNotNull, if the first argument is not NULL. Returns the third argument, ifNull, if the first argument is NULL.

Equivalent to the NVL2( ) function in Oracle Database.

Return type: Same as the first argument value

Added in: CDH 5.12.0 / Impala 2.9.0

Examples:

```
SELECT NVL2(NULL, 999, 0); -- Returns 0
SELECT NVL2('ABC', 'Is Not Null', 'Is Null'); -- Returns 'Is Not Null'
```

ZEROIFNULL(numeric_expr)

Purpose: Returns 0 if the numeric expression evaluates to NULL, otherwise returns the result of the expression.

Usage notes: Used to avoid unexpected results due to unexpected propagation of NULL values in numeric calculations. Serves as shorthand for a more elaborate CASE expression, to simplify porting SQL with vendor extensions to Impala.

Return type: Same type as the input argument

Added in: Impala 1.3.0

Impala String Functions

String functions are classified as those primarily accepting or returning STRING, VARCHAR, or CHAR data types, for example to measure the length of a string or concatenate two strings together.

- All the functions that accept STRING arguments also accept the VARCHAR and CHAR types introduced in Impala 2.0.
- Whenever VARCHAR or CHAR values are passed to a function that returns a string value, the return type is normalized to STRING. For example, a call to concat( ) with a mix of STRING, VARCHAR, and CHAR arguments produces a STRING result.

Related information:

The string functions operate mainly on these data types: STRING Data Type on page 123, VARCHAR Data Type (CDH 5.2 or higher only) on page 137, and CHAR Data Type (CDH 5.2 or higher only) on page 107.

Function reference:

Impala supports the following string functions:

- ASCII
- BASE64DECODE
- BASE64ENCODE
- BTRIM
- CHAR_LENGTH
- CHR
- CONCAT
- CONCAT_WS
- FIND_IN_SET
- GROUP_CONCAT
- INITCAP
- INSTR
- LEFT
- LENGTH
- LEVENSHTEIN, LE_DST
- LOCATE
- LOWER, LCASE
- LPAD
- LTRI
- PARSE_URL
- REGEXP_ESCAPE
- REGEXP_EXTRACT
- REGEXP_LIKE
- REGEXP_REPLACE
- REPEAT
- REPLACE
- REVERSE
- RIGHT
- RPAD
- RTRIM
- SPACE
- SPLIT_PART
- STRLEFT
- STRRIGHT
- SUBSTR, SUBSTRING

- TRANSLATE
- TRIM
- UPPER, UCASE

ASCII(STRING str)

Purpose: Returns the numeric ASCII code of the first character of the argument.

Return type: `INT`

BASE64DECODE(STRING str)

Purpose:

Return type: `STRING`

Usage notes:

For general information about Base64 encoding, see Base64 article on Wikipedia.

The functions `BASE64ENCODE()` and `BASE64DECODE()` are typically used in combination, to store in an Impala table string data that is problematic to store or transmit. For example, you could use these functions to store string data that uses an encoding other than UTF-8, or to transform the values in contexts that require ASCII values, such as for partition key columns. Keep in mind that base64-encoded values produce different results for string functions such as `LENGTH()`, `MAX()`, and `MIN()` than when those functions are called with the unencoded string values.

The set of characters that can be generated as output from `BASE64ENCODE()`, or specified in the argument string to `BASE64DECODE()`, are the ASCII uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the punctuation characters +, /, and =.

All return values produced by `BASE64ENCODE()` are a multiple of 4 bytes in length. All argument values supplied to `BASE64DECODE()` must also be a multiple of 4 bytes in length. If a base64-encoded value would otherwise have a different length, it can be padded with trailing = characters to reach a length that is a multiple of 4 bytes.

If the argument string to `BASE64DECODE()` does not represent a valid base64-encoded value, subject to the constraints of the Impala implementation such as the allowed character set, the function returns `NULL`.

Examples:

The following examples show how to use `BASE64ENCODE()` and `BASE64DECODE()` together to store and retrieve string values:

```
-- An arbitrary string can be encoded in base 64.
-- The length of the output is a multiple of 4 bytes,
-- padded with trailing = characters if necessary.
select base64encode('hello world') as encoded,
  length(base64encode('hello world')) as length;
+------------------+--------+
| encoded          | length |
+------------------+--------+
| aGVsbG8gd29ybGQ= | 16     |
+------------------+--------+

-- Passing an encoded value to base64decode() produces
-- the original value.
select base64decode('aGVsbG8gd29ybGQ=') as decoded;
+-------------+
| decoded     |
+-------------+
| hello world |
+-------------+
```

These examples demonstrate incorrect encoded values that produce `NULL` return values when decoded:

```
-- The input value to base64decode() must be a multiple of 4 bytes.
-- In this case, leaving off the trailing = padding character
-- produces a NULL return value.
```

```
select base64decode('aGVsbG8gd29ybGQ') as decoded;
+---------+
| decoded |
+---------+
| NULL    |
+---------+
WARNINGS: UDF WARNING: Invalid base64 string; input length is 15,
  which is not a multiple of 4.

-- The input to base64decode() can only contain certain characters.
-- The $ character in this case causes a NULL return value.
select base64decode('abc$');
+----------------------+
| base64decode('abc$') |
+----------------------+
| NULL                 |
+----------------------+
WARNINGS: UDF WARNING: Could not base64 decode input in space 4; actual output length
0
```

These examples demonstrate “round-tripping” of an original string to an encoded string, and back again. This technique is applicable if the original source is in an unknown encoding, or if some intermediate processing stage might cause national characters to be misrepresented:

```
select 'circumflex accents: â, ê, î, ô, û' as original,
  base64encode('circumflex accents: â, ê, î, ô, û') as encoded;
+---------------------------------+-----------------------------------------------------+
| original                        | encoded
     |
+---------------------------------+-----------------------------------------------------+
| circumflex accents: â, ê, î, ô, û | Y2lyY3VtZmxleCBhY2NlbnRzOiDDoiwgw6osIMOuLCDDtCwgw7s=
 |
+---------------------------------+-----------------------------------------------------+

select base64encode('circumflex accents: â, ê, î, ô, û') as encoded,
  base64decode(base64encode('circumflex accents: â, ê, î, ô, û')) as decoded;
+-----------------------------------------------------+----------------------------------+
| encoded                                             | decoded
     |
+-----------------------------------------------------+----------------------------------+
| Y2lyY3VtZmxleCBhY2NlbnRzOiDDoiwgw6osIMOuLCDDtCwgw7s= | circumflex accents: â, ê, î,
ô, û |
+-----------------------------------------------------+----------------------------------+
```

BASE64ENCODE(STRING str)

Purpose:

Return type: `STRING`

Usage notes:

For general information about Base64 encoding, see Base64 article on Wikipedia.

The functions `BASE64ENCODE()` and `BASE64DECODE()` are typically used in combination, to store in an Impala table string data that is problematic to store or transmit. For example, you could use these functions to store string data that uses an encoding other than UTF-8, or to transform the values in contexts that require ASCII values, such as for partition key columns. Keep in mind that base64-encoded values produce different results for string functions such as `LENGTH()`, `MAX()`, and `MIN()` than when those functions are called with the unencoded string values.

The set of characters that can be generated as output from `BASE64ENCODE()`, or specified in the argument string to `BASE64DECODE()`, are the ASCII uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the punctuation characters +, /, and =.

All return values produced by `BASE64ENCODE()` are a multiple of 4 bytes in length. All argument values supplied to `BASE64DECODE()` must also be a multiple of 4 bytes in length. If a base64-encoded value would otherwise have a different length, it can be padded with trailing = characters to reach a length that is a multiple of 4 bytes.

Examples:

The following examples show how to use BASE64ENCODE( ) and BASE64DECODE( ) together to store and retrieve string values:

```
-- An arbitrary string can be encoded in base 64.
-- The length of the output is a multiple of 4 bytes,
-- padded with trailing = characters if necessary.
select base64encode('hello world') as encoded,
  length(base64encode('hello world')) as length;
+------------------+--------+
| encoded          | length |
+------------------+--------+
| aGVsbG8gd29ybGQ= | 16     |
+------------------+--------+

-- Passing an encoded value to base64decode() produces
-- the original value.
select base64decode('aGVsbG8gd29ybGQ=') as decoded;
+-------------+
| decoded     |
+-------------+
| hello world |
+-------------+
```

These examples demonstrate incorrect encoded values that produce NULL return values when decoded:

```
-- The input value to base64decode() must be a multiple of 4 bytes.
-- In this case, leaving off the trailing = padding character
-- produces a NULL return value.
select base64decode('aGVsbG8gd29ybGQ') as decoded;
+---------+
| decoded |
+---------+
| NULL    |
+---------+
WARNINGS: UDF WARNING: Invalid base64 string; input length is 15,
  which is not a multiple of 4.

-- The input to base64decode() can only contain certain characters.
-- The $ character in this case causes a NULL return value.
select base64decode('abc$');
+----------------------+
| base64decode('abc$') |
+----------------------+
| NULL                 |
+----------------------+
WARNINGS: UDF WARNING: Could not base64 decode input in space 4; actual output length
0
```

These examples demonstrate “round-tripping” of an original string to an encoded string, and back again. This technique is applicable if the original source is in an unknown encoding, or if some intermediate processing stage might cause national characters to be misrepresented:

```
select 'circumflex accents: â, ê, î, ô, û' as original,
  base64encode('circumflex accents: â, ê, î, ô, û') as encoded;
+---------------------------------+-----------------------------------------------------+
| original                        | encoded
     |
+---------------------------------+-----------------------------------------------------+
| circumflex accents: â, ê, î, ô, û | Y2lyY3VtZmxleCBhY2NlbnRzOiDDoiwgw6osIMOuLCDDtCwgw7s=
 |
+---------------------------------+-----------------------------------------------------+

select base64encode('circumflex accents: â, ê, î, ô, û') as encoded,
  base64decode(base64encode('circumflex accents: â, ê, î, ô, û')) as decoded;
+-----------------------------------------------------+---------------------------------+
| encoded                                             | decoded
```

```
     |
+-----------------------------------------------------+----------------------------------+
| Y2lyY3VtZmxleCBhY2NlbnRzOiDDoiwgw6osIMOuLCDDtCwgw7s= | circumflex accents: â, ê, î,
ô, û |
+-----------------------------------------------------+----------------------------------+
```

BTRIM(STRING a), BTRIM(STRING a, STRING chars_to_trim)

Purpose: Removes all instances of one or more characters from the start and end of a `STRING` value. By default, removes only spaces. If a non-`NULL` optional second argument is specified, the function removes all occurrences of characters in that second argument from the beginning and end of the string.

Return type: `STRING`

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

The following examples show the default `btrim()` behavior, and what changes when you specify the optional second argument. All the examples bracket the output value with `[ ]` so that you can see any leading or trailing spaces in the `btrim()` result. By default, the function removes and number of both leading and trailing spaces. When the second argument is specified, any number of occurrences of any character in the second argument are removed from the start and end of the input string; in this case, spaces are not removed (unless they are part of the second argument) and any instances of the characters are not removed if they do not come right at the beginning or end of the string.

```
-- Remove multiple spaces before and one space after.
select concat('[',btrim('    hello '),']');
+--------------------------------------+
| concat('[', btrim('    hello '), ']') |
+--------------------------------------+
| [hello]                              |
+--------------------------------------+

-- Remove any instances of x or y or z at beginning or end. Leave spaces alone.
select concat('[',btrim('xy    hello zyzzxx','xyz'),']');
+-----------------------------------------------------+
| concat('[', btrim('xy    hello zyzzxx', 'xyz'), ']') |
+-----------------------------------------------------+
| [    hello ]                                        |
+-----------------------------------------------------+

-- Remove any instances of x or y or z at beginning or end.
-- Leave x, y, z alone in the middle of the string.
select concat('[',btrim('xyhelxyzlozyzzxx','xyz'),']');
+-----------------------------------------------------+
| concat('[', btrim('xyhelxyzlozyzzxx', 'xyz'), ']') |
+-----------------------------------------------------+
| [helxyzlo]                                          |
+-----------------------------------------------------+
```

CHAR_LENGTH(STRING a), CHARACTER_LENGTH(STRING a)

Purpose: Returns the length in characters of the argument string. Aliases for the `length()` function.

Return type: `INT`

CHR(INT character_code)

Purpose: Returns a character specified by a decimal code point value. The interpretation and display of the resulting character depends on your system locale. Because consistent processing of Impala string values is only guaranteed for values within the ASCII range, only use this function for values corresponding to ASCII characters. In particular, parameter values greater than 255 return an empty string.

Return type: `STRING`

Usage notes: Can be used as the inverse of the `ascii()` function, which converts a character to its numeric ASCII code.

Added in: CDH 5.5.0 / Impala 2.3.0

Examples:

```
SELECT chr(65);
+---------+
| chr(65) |
+---------+
| A       |
+---------+

SELECT chr(97);
+---------+
| chr(97) |
+---------+
| a       |
+---------+
```

CONCAT(STRING a, STRING b...)

Purpose: Returns a single string representing all the argument values joined together.

Return type: `STRING`

Usage notes: `concat()` and `concat_ws()` are appropriate for concatenating the values of multiple columns within the same row, while `group_concat()` joins together values from different rows.

CONCAT_WS(STRING sep, STRING a, STRING b...)

Purpose: Returns a single string representing the second and following argument values joined together, delimited by a specified separator.

Return type: `STRING`

Usage notes: `concat()` and `concat_ws()` are appropriate for concatenating the values of multiple columns within the same row, while `group_concat()` joins together values from different rows.

FIND_IN_SET(STRING str, STRING strList)

Purpose: Returns the position (starting from 1) of the first occurrence of a specified string within a comma-separated string. Returns `NULL` if either argument is `NULL`, 0 if the search string is not found, or 0 if the search string contains a comma.

Return type: `INT`

GROUP_CONCAT(STRING s [, STRING sep])

Purpose: Returns a single string representing the argument value concatenated together for each row of the result set. If the optional separator string is specified, the separator is added between each pair of concatenated values.

Return type: `STRING`

Usage notes: `concat()` and `concat_ws()` are appropriate for concatenating the values of multiple columns within the same row, while `group_concat()` joins together values from different rows.

By default, returns a single string covering the whole result set. To include other columns or values in the result set, or to produce multiple concatenated strings for subsets of rows, include a `GROUP BY` clause in the query.

Strictly speaking, `group_concat()` is an aggregate function, not a scalar function like the others in this list. For additional details and examples, see GROUP_CONCAT Function on page 489.

INITCAP(STRING str)

Purpose: Returns the input string with the first letter of each word capitalized and all other letters in lowercase.

Return type: `STRING`

Example:

`INITCAP("i gOt mY ChiCkeNs in tHe yard.")` returns `"I Got My Chickens In The Yard."`.

INSTR(STRING str, STRING substr [, BIGINT position [, BIGINT occurrence]])

Purpose: Returns the position (starting from 1) of the first occurrence of a *substr* within a longer string.

Return type: `INT`

Usage notes:

If the *substr* is not present in *str*, the function returns 0.

The optional third and fourth arguments let you find instances of the *substr* other than the first instance starting from the left.

- The third argument, *position*, lets you specify a starting point within the *str* other than 1.

```
-- Restricting the search to positions 7..end,
-- the first occurrence of 'b' is at position 9.
select instr('foo bar bletch', 'b', 7);
+---------------------------------+
| instr('foo bar bletch', 'b', 7) |
+---------------------------------+
| 9                               |
+---------------------------------+
```

- If there are no more occurrences after the specified position, the result is 0.
- If *position* is negative, the search works right-to-left starting that many characters from the right. The return value still represents the position starting from the left side of *str*.

```
-- Scanning right to left, the first occurrence of 'o'
-- is at position 8. (8th character from the left.)
select instr('hello world','o',-1);
+-------------------------------+
| instr('hello world', 'o', -1) |
+-------------------------------+
| 8                             |
+-------------------------------+
```

- The fourth argument, *occurrence*, lets you specify an occurrence other than the first.

```
-- 2nd occurrence of 'b' is at position 9.
select instr('foo bar bletch', 'b', 1, 2);
+------------------------------------+
| instr('foo bar bletch', 'b', 1, 2) |
+------------------------------------+
| 9                                  |
+------------------------------------+
```

- If *occurrence* is greater than the number of matching occurrences, the function returns 0.
- *occurrence* cannot be negative or zero. A non-positive value for this argument causes an error.
- If either of the optional arguments, *position* or *occurrence*, is `NULL`, the function also returns `NULL`.

LEFT(STRING a, INT num_chars)

See the `STRLEFT( )` function.

LENGTH(STRING a)

Purpose: Returns the length in characters of the argument string.

Return type: `INT`

LEVENSHTEIN(STRING str1, STRING str2), LE_DST(STRING str1, STRING str2)

Purpose: Returns the Levenshtein distance between two input strings. The Levenshtein distance between two strings is the minimum number of single-character edits required to transform one string to other. The function indicates how different the input strings are.

Return type: `INT`

Usage notes:

If input strings are equal, the function returns 0.

If either input exceeds 255 characters, the function returns an error.

If either input string is `NULL`, the function returns `NULL`.

If the length of one input string is zero, the function returns the length of the other string.

Example:

`LEVENSHTEIN ('welcome', 'We come')` returns 2, first change to replace 'w' to 'W', and then to replace 'l' to a space character.

LOCATE(STRING substr, STRING str[, INT pos])

Purpose: Returns the position (starting from 1) of the first occurrence of a substring within a longer string, optionally after a particular position.

Return type: `INT`

LOWER(STRING a), LCASE(STRING a)

Purpose: Returns the argument string converted to all-lowercase.

Return type: `STRING`

Usage notes:

In CDH 5.7 / Impala 2.5 and higher, you can simplify queries that use many `UPPER()` and `LOWER()` calls to do case-insensitive comparisons, by using the `ILIKE` or `IREGEXP` operators instead. See ILIKE Operator on page 179 and IREGEXP Operator on page 182 for details.

LPAD(STRING str, INT len, STRING pad)

Purpose: Returns a string of a specified length, based on the first argument string. If the specified string is too short, it is padded on the left with a repeating sequence of the characters from the pad string. If the specified string is too long, it is truncated on the right.

Return type: `STRING`

LTRIM(STRING a [, STRING chars_to_trim])

Purpose: Returns the argument string with all occurrences of characters specified by the second argument removed from the left side. Removes spaces if the second argument is not specified.

Return type: `STRING`

PARSE_URL(STRING urlString, STRING partToExtract [, STRING keyToExtract])

Purpose: Returns the portion of a URL corresponding to a specified part. The part argument can be `'PROTOCOL'`, `'HOST'`, `'PATH'`, `'REF'`, `'AUTHORITY'`, `'FILE'`, `'USERINFO'`, or `'QUERY'`. Uppercase is required for these literal values. When requesting the `QUERY` portion of the URL, you can optionally specify a key to retrieve just the associated value from the key-value pairs in the query string.

Return type: `STRING`

Usage notes: This function is important for the traditional Hadoop use case of interpreting web logs. For example, if the web traffic data features raw URLs not divided into separate table columns, you can count visitors to a particular page by extracting the `'PATH'` or `'FILE'` field, or analyze search terms by extracting the corresponding key from the `'QUERY'` field.

REGEXP_ESCAPE(STRING source)

Purpose: The REGEXP_ESCAPE function returns a string escaped for the special character in RE2 library so that the special characters are interpreted literally rather than as special characters. The following special characters are escaped by the function:

```
.\+*?[^]$(){}=!<>|:-
```

Return type: `string`

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including `.*?` for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the `\` character for escaping, use `\\` to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as `[[:digit:]]` instead of `\d` which you would have to escape as `\\d`.

Examples:

This example shows escaping one of special characters in RE2.

```
+-------------------------------------------------------+
| regexp_escape('Hello.world')                          |
+-------------------------------------------------------+
| Hello\.world                                          |
+-------------------------------------------------------+
```

This example shows escaping all the special characters in RE2.

```
+-------------------------------------------------------------+
| regexp_escape('a.b\\c+d*e?f[g]h$i(j)k{l}m=n!o<p>q|r:s-t')  |
+-------------------------------------------------------------+
| a\.b\\c\+d\*e\?f\[g\]h\$i\(j\)k\{l\}m\=n\!o\<p\>q\|r\:s\-t |
+-------------------------------------------------------------+
```

REGEXP_EXTRACT(STRING subject, STRING pattern, INT index)

Purpose: Returns the specified () group from a string based on a regular expression pattern. Group 0 refers to the entire extracted string, while group 1, 2, and so on refers to the first, second, and so on `(...)` portion.

Return type: `STRING`

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including `.*?` for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the `\` character for escaping, use `\\` to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as `[[:digit:]]` instead of `\d` which you would have to escape as `\\d`.

Examples:

This example shows how group 0 matches the full pattern string, including the portion outside any ( ) group:

```
[localhost:21000] > select regexp_extract('abcdef123ghi456jkl','.*?(\\d+)',0);
+-----------------------------------------------------+
| regexp_extract('abcdef123ghi456jkl', '.*?(\\d+)', 0) |
+-----------------------------------------------------+
| abcdef123ghi456                                     |
+-----------------------------------------------------+
Returned 1 row(s) in 0.11s
```

This example shows how group 1 matches just the contents inside the first ( ) group in the pattern string:

```
[localhost:21000] > select regexp_extract('abcdef123ghi456jkl','.*?(\\d+)',1);
+-----------------------------------------------------+
| regexp_extract('abcdef123ghi456jkl', '.*?(\\d+)', 1) |
+-----------------------------------------------------+
| 456                                                 |
+-----------------------------------------------------+
Returned 1 row(s) in 0.11s
```

Unlike in earlier Impala releases, the regular expression library used in Impala 2.0 and later supports the `.*?` idiom for non-greedy matches. This example shows how a pattern string starting with `.*?` matches the shortest possible portion of the source string, returning the rightmost set of lowercase letters. A pattern string both starting and ending with `.*?` finds two potential matches of equal length, and returns the first one found (the leftmost set of lowercase letters).

```
[localhost:21000] > select regexp_extract('AbcdBCdefGHI','.*?([[:lower:]]+)',1);
+-----------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*?([[:lower:]]+)', 1) |
+-----------------------------------------------------+
| def                                                 |
+-----------------------------------------------------+
[localhost:21000] > select regexp_extract('AbcdBCdefGHI','.*?([[:lower:]]+).*?',1);
+-------------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*?([[:lower:]]+).*?', 1) |
+-------------------------------------------------------+
| bcd                                                   |
+-------------------------------------------------------+
```

REGEXP_LIKE(STRING source, STRING pattern[, STRING options])

Purpose: Returns `true` or `false` to indicate whether the source string contains anywhere inside it the regular expression given by the pattern. The optional third argument consists of letter flags that change how the match is performed, such as `i` for case-insensitive matching.

Syntax:

The flags that you can include in the optional third argument are:

- `c`: Case-sensitive matching (the default).
- `i`: Case-insensitive matching. If multiple instances of `c` and `i` are included in the third argument, the last such option takes precedence.
- `m`: Multi-line matching. The `^` and `$` operators match the start or end of any line within the source string, not the start and end of the entire string.
- `n`: Newline matching. The `.` operator can match the newline character. A repetition operator such as `.*` can match a portion of the source string that spans multiple lines.

Return type: BOOLEAN

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including `.*?` for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the `\` character for escaping, use `\\` to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as `[[:digit:]]` instead of `\d` which you would have to escape as `\\d`.

Examples:

This example shows how `REGEXP_LIKE()` can test for the existence of various kinds of regular expression patterns within a source string:

```
-- Matches because the 'f' appears somewhere in 'foo'.
select regexp_like('foo','f');
+-------------------------+
| regexp_like('foo', 'f') |
+-------------------------+
| true                    |
+-------------------------+

-- Does not match because the comparison is case-sensitive by default.
select regexp_like('foo','F');
+-------------------------+
| regexp_like('foo', 'f') |
+-------------------------+
| false                   |
+-------------------------+

-- The 3rd argument can change the matching logic, such as 'i' meaning case-insensitive.
select regexp_like('foo','F','i');
+------------------------------+
| regexp_like('foo', 'f', 'i') |
+------------------------------+
| true                         |
+------------------------------+

-- The familiar regular expression notations work, such as ^ and $ anchors...
select regexp_like('foo','f$');
+--------------------------+
| regexp_like('foo', 'f$') |
+--------------------------+
| false                    |
+--------------------------+

select regexp_like('foo','o$');
+--------------------------+
| regexp_like('foo', 'o$') |
+--------------------------+
| true                     |
+--------------------------+

-- ...and repetition operators such as * and +
select regexp_like('fooooooobar','fo+b');
+-----------------------------------+
| regexp_like('fooooooobar', 'fo+b') |
+-----------------------------------+
| true                              |
+-----------------------------------+

select regexp_like('fooooooobar','fx*y*o*b');
+--------------------------------------+
| regexp_like('fooooooobar', 'fx*y*o*b') |
+--------------------------------------+
| true                                 |
+--------------------------------------+
```

REGEXP_REPLACE(STRING initial, STRING pattern, STRING replacement)

Purpose: Returns the initial argument with the regular expression pattern replaced by the final argument string.

Return type: `STRING`

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including `.*?` for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the `\` character for escaping, use `\\` to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as `[[:digit:]]` instead of `\d` which you would have to escape as `\\d`.

Examples:

These examples show how you can replace parts of a string matching a pattern with replacement text, which can include backreferences to any `()` groups in the pattern string. The backreference numbers start at 1, and any `\` characters must be escaped as `\\`.

Replace a character pattern with new text:

```
[localhost:21000] > select regexp_replace('aaabbbaaa','b+','xyz');
+------------------------------------------+
| regexp_replace('aaabbbaaa', 'b+', 'xyz') |
+------------------------------------------+
| aaaxyzaaa                                |
+------------------------------------------+
Returned 1 row(s) in 0.11s
```

Replace a character pattern with substitution text that includes the original matching text:

```
[localhost:21000] > select regexp_replace('aaabbbaaa','(b+)','<\\1>');
+--------------------------------------------+
| regexp_replace('aaabbbaaa', '(b+)', '<\\1>') |
+--------------------------------------------+
| aaa<bbb>aaa                                |
+--------------------------------------------+
Returned 1 row(s) in 0.11s
```

Remove all characters that are not digits:

```
[localhost:21000] > select regexp_replace('123-456-789','[^[:digit:]]','');
+----------------------------------------------------+
| regexp_replace('123-456-789', '[^[:digit:]]', '') |
+----------------------------------------------------+
| 123456789                                          |
+----------------------------------------------------+
Returned 1 row(s) in 0.12s
```

REPEAT(STRING str, INT n)

Purpose: Returns the argument string repeated a specified number of times.

Return type: `STRING`

REPLACE(STRING initial, STRING target, STRING replacement)

Purpose: Returns the initial argument with all occurrences of the target string replaced by the replacement string.

Return type: `STRING`

Usage notes:

Because this function does not use any regular expression patterns, it is typically faster than `REGEXP_REPLACE()` for simple string substitutions.

If any argument is `NULL`, the return value is `NULL`.

Matching is case-sensitive.

If the replacement string contains another instance of the target string, the expansion is only performed once, instead of applying again to the newly constructed string.

Added in: CDH 5.12.0 / Impala 2.9.0

Examples:

```
-- Replace one string with another.
select replace('hello world','world','earth');
+-----------------------------------------+
| replace('hello world', 'world', 'earth') |
+-----------------------------------------+
| hello earth                              |
+-----------------------------------------+

-- All occurrences of the target string are replaced.
select replace('hello world','o','0');
+---------------------------------+
| replace('hello world', 'o', '0') |
+---------------------------------+
| hell0 w0rld                      |
+---------------------------------+

-- If no match is found, the original string is returned unchanged.
select replace('hello world','xyz','abc');
+-----------------------------------+
| replace('hello world', 'xyz', 'abc') |
+-----------------------------------+
| hello world                       |
+-----------------------------------+
```

REVERSE(STRING a)

Purpose: Returns the argument string with characters in reversed order.

Return type: `STRING`

RIGHT(STRING a, INT num_chars)

See the `STRRIGHT` function.

RPAD(STRING str, INT len, STRING pad)

Purpose: Returns a string of a specified length, based on the first argument string. If the specified string is too short, it is padded on the right with a repeating sequence of the characters from the pad string. If the specified string is too long, it is truncated on the right.

Return type: `STRING`

RTRIM(STRING a [, STRING chars_to_trim])

Purpose: Returns the argument string with all occurrences of characters specified by the second argument removed from the right side. Removes spaces if the second argument is not specified.

Return type: `STRING`

SPACE(INT n)

Purpose: Returns a concatenated string of the specified number of spaces. Shorthand for `repeat(' ',n)`.

Return type: `STRING`

SPLIT_PART(STRING source, STRING delimiter, BIGINT n)

Purpose: Returns the nth field within a delimited string. The fields are numbered starting from 1. The delimiter can consist of multiple characters, not just a single character. All matching of the delimiter is done exactly, not using any regular expression patterns.

Return type: `STRING`

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the RE2 documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on, including `.*?` for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the `\` character for escaping, use `\\` to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as `[[:digit:]]` instead of `\d` which you would have to escape as `\\d`.

Examples:

These examples show how to retrieve the nth field from a delimited string:

```
select split_part('x,y,z',',',1);
+----------------------------+
| split_part('x,y,z', ',', 1) |
+----------------------------+
| x                          |
+----------------------------+

select split_part('x,y,z',',',2);
+----------------------------+
| split_part('x,y,z', ',', 2) |
+----------------------------+
| y                          |
+----------------------------+

select split_part('x,y,z',',',3);
+----------------------------+
| split_part('x,y,z', ',', 3) |
+----------------------------+
| z                          |
+----------------------------+
```

These examples show what happens for out-of-range field positions. Specifying a value less than 1 produces an error. Specifying a value greater than the number of fields returns a zero-length string (which is not the same as `NULL`).

```
select split_part('x,y,z',',',0);
ERROR: Invalid field position: 0

with t1 as (select split_part('x,y,z',',',4) nonexistent_field)
  select
      nonexistent_field
    , concat('[',nonexistent_field,']')
    , length(nonexistent_field);
from t1
+-------------------+-----------------------------------+--------------------------+
| nonexistent_field | concat('[', nonexistent_field, ']') | length(nonexistent_field)
|
+-------------------+-----------------------------------+--------------------------+
|                   | []                                | 0
|
+-------------------+-----------------------------------+--------------------------+
```

These examples show how the delimiter can be a multi-character value:

```
select split_part('one***two***three','***',2);
+-----------------------------------------+
| split_part('one***two***three', '***', 2) |
+-----------------------------------------+
| two                                     |
+-----------------------------------------+
```

```
select split_part('one\|/two\|/three','\|/',3);
+-----------------------------------------+
| split_part('one\|/two\|/three', '\|/', 3) |
+-----------------------------------------+
| three                                   |
+-----------------------------------------+
```

STRLEFT(STRING a, INT num_chars)

Purpose: Returns the leftmost characters of the string. Shorthand for a call to `substr()` with 2 arguments.

Return type: `STRING`

STRRIGHT(STRING a, INT num_chars)

Purpose: Returns the rightmost characters of the string. Shorthand for a call to `substr()` with 2 arguments.

Return type: `STRING`

SUBSTR(STRING a, INT start [, INT len]), SUBSTRING(STRING a, INT start [, INT len])

Purpose: Returns the portion of the string starting at a specified point, optionally with a specified maximum length. The characters in the string are indexed starting at 1.

Return type: `STRING`

TRANSLATE(STRING input, STRING from, STRING to)

Purpose: Returns the `input` string with each character in the `from` argument replaced with the corresponding character in the `to` argument. The characters are matched in the order they appear in `from` and `to`.

For example: `translate ('hello world','world','earth')` returns `'hetta earth'`.

Return type: `STRING`

Usage notes:

If `from` contains more characters than `to`, the `from` characters that are beyond the length of `to` are removed in the result.

For example:

`translate('abcdedg', 'bcd', '1')` returns `'a1eg'`.

`translate('Unit Number#2', '# ', '_')` returns `'UnitNumber_2'`.

If `from` is `NULL`, the function returns `NULL`.

If `to` contains more characters than `from`, the extra characters in `to` are ignored.

If `from` contains duplicate characters, the duplicate character is replaced with the first matching character in `to`.

For example: `translate ('hello','ll','67')` returns `'he66o'`.

TRIM(STRING a)

Purpose: Returns the input string with both leading and trailing spaces removed. The same as passing the string through both `LTRIM()` and `RTRIM()`.

Usage notes: Often used during data cleansing operations during the ETL cycle, if input values might still have surrounding spaces. For a more general-purpose function that can remove other leading and trailing characters besides spaces, see `BTRIM()`.

Return type: `STRING`

UPPER(STRING a), UCASE(STRING a)

Purpose: Returns the argument string converted to all-uppercase.

Return type: `STRING`

Usage notes:

In CDH 5.7 / Impala 2.5 and higher, you can simplify queries that use many `UPPER()` and `LOWER()` calls to do case-insensitive comparisons, by using the `ILIKE` or `IREGEXP` operators instead. See ILIKE Operator on page 179 and IREGEXP Operator on page 182 for details.

Impala Miscellaneous Functions

Impala supports the following utility functions that do not operate on a particular column or data type:

- CURRENT_DATABASE
- EFFECTIVE_USER
- GET_JSON_OBJECT
- LOGGED_IN_USER
- PID
- SLEEP
- USER
- UUID
- VERSION

CURREN_DATABASE()

Purpose: Returns the database that the session is currently using, either `default` if no database has been selected, or whatever database the session switched to through a `USE` statement or the `impalad -d` option.

Return type: `STRING`

EFFECTIVE_USER()

Purpose: Typically returns the same value as `USER()`, except if delegation is enabled, in which case it returns the ID of the delegated user.

Return type: `STRING`

Added in: CDH 5.4.5 / Impala 2.2.5

GET_JSON_OBJECT(STRING json_str, STRING selector)

Purpose: Extracts JSON object from the *json_str* based on the *selector* JSON path and returns the string of the extracted JSON object.

The function returns `NULL` if the input *json_str* is invalid or if nothing is selected based on the *selector* JSON path.

The following characters are supported in the *selector* JSON path:

- $: Denotes the root object
- . : Denotes the child operator
- [] : Denotes the subscript operator for array
- * : Denotes the wildcard for [] or .

Return type: `STRING`

Examples:

```
---- QUERY
SELECT GET_JSON_OBJECT ('{"a":true, "b":false, "c":true}', '$.*');
---- RESULTS
[true,false,true]
```

```
---- QUERY
SELECT GET_JSON_OBJECT(t.json, '$.a.b.c') FROM (VALUES (
 ('{"a": {"b": {"c": 1}}}' AS json),
 ('{"a": {"b": {"c": 2}}}'),
 ('{"a": {"b": {"c": 3}}}')
)) t
---- RESULTS
'1'
```

```
'2'
'3'
```

```
---- QUERY
SELECT GET_JSON_OBJECT(t.json, '$.a'),
 GET_JSON_OBJECT(t.json, '$.b'),
 GET_JSON_OBJECT(t.json, '$.c')
FROM (VALUES (
 ('{"a":1, "b":2, "c":3}' AS json),
 ('{"b":2, "c":3}'),
 ('{"c":3}')
)) t
---- RESULTS
'1','2','3'
'NULL','2','3'
'NULL','NULL','3'
```

```
---- QUERY
SELECT GET_JSON_OBJECT(t.json, '$[1]'),
 GET_JSON_OBJECT(t.json, '$[*]')
FROM (VALUES (
 ('["a", "b", "c"]' AS json),
 ('["a", "b"]'),
 ('["a"]')
)) t
---- RESULTS
'b','["a","b","c"]'
'b','["a","b"]'
'NULL','a'
```

Added in: CDH 6.1

LOGGED_IN_USER()

Purpose: Typically returns the same value as `USER()`. If delegation is enabled, it returns the ID of the delegated user.

`LOGGED_IN_USER()` is an alias of `EFFECTIVE_USER()`.

Return type: `STRING`

Added in: CDH 6.1

PID()

Purpose: Returns the process ID of the `impalad` daemon that the session is connected to. You can use it during low-level debugging, to issue Linux commands that trace, show the arguments, and so on the `impalad` process.

Return type: `INT`

SLEEP(INT ms)

Purpose: Pauses the query for a specified number of milliseconds. For slowing down queries with small result sets enough to monitor runtime execution, memory usage, or other factors that otherwise would be difficult to capture during the brief interval of query execution. When used in the `SELECT` list, it is called once for each row in the result set; adjust the number of milliseconds accordingly. For example, a query `SELECT *, sleep(5) FROM table_with_1000_rows` would take at least 5 seconds to complete (5 milliseconds * 1000 rows in result set). To avoid an excessive number of concurrent queries, use this function for troubleshooting on test and development systems, not for production queries.

Return type: N/A

USER()

Purpose: Returns the username of the Linux user who is connected to the `impalad` daemon. Typically called a single time, in a query without any `FROM` clause, to understand how authorization settings apply in a security context; once you know the logged-in username, you can check which groups that user belongs to, and from the list of groups you can check which roles are available to those groups through the authorization policy file.

Impala 2.0 and later, `USER()` returns the full Kerberos principal string, such as `user@example.com`, in a Kerberized environment.

When delegation is enabled, consider calling the `effective_user()` function instead.

Return type: `STRING`

UUID()

Purpose: Returns a universal unique identifier, a 128-bit value encoded as a string with groups of hexadecimal digits separated by dashes.

Each call to `UUID()` produces a new arbitrary value.

If you get a UUID for each row of a result set, you can use it as a unique identifier within a table, or even a unique ID across tables.

Return type: `STRING`

Added in: CDH 5.7.0 / Impala 2.5.0

Usage notes:

Ascending numeric sequences of type `BIGINT` are often used as identifiers within a table, and as join keys across multiple tables. The `uuid()` value is a convenient alternative that does not require storing or querying the highest sequence number. For example, you can use it to quickly construct new unique identifiers during a data import job, or to combine data from different tables without the likelihood of ID collisions.

VERSION()

Purpose: Returns information such as the precise version number and build date for the `impalad` daemon that you are currently connected to. Typically used to confirm that you are connected to the expected level of Impala to use a particular feature, or to connect to several nodes and confirm they are all running the same level of `impalad`.

Return type: `STRING` (with one or more embedded newlines)

COORDINATOR()

Purpose: Returns the name of the host which is running the `impalad` daemon that is acting as the `coordinator` for the current query.

Return type: `STRING`

Added in: CDH 6.1

Impala Aggregate Functions

Aggregate functions are a special category with different rules. These functions calculate a return value across all the items in a result set, so they require a `FROM` clause in the query:

```
select count(product_id) from product_catalog;
select max(height), avg(height) from census_data where age > 20;
```

Aggregate functions also ignore `NULL` values rather than returning a `NULL` result. For example, if some rows have `NULL` for a particular column, those rows are ignored when computing the `AVG()` for that column. Likewise, specifying `COUNT(col_name)` in a query counts only those rows where *col_name* contains a non-`NULL` value.

APPX_MEDIAN Function

An aggregate function that returns a value that is approximately the median (midpoint) of values in the set of input values.

Syntax:

```
APPX_MEDIAN([DISTINCT | ALL] expression)
```

This function works with any input type, because the only requirement is that the type supports less-than and greater-than comparison operators.

Usage notes:

Because the return value represents the estimated midpoint, it might not reflect the precise midpoint value, especially if the cardinality of the input values is very high. If the cardinality is low (up to approximately 20,000), the result is more accurate because the sampling considers all or almost all of the different values.

Return type: Same as the input value, except for CHAR and VARCHAR arguments which produce a STRING result

The return value is always the same as one of the input values, not an "in-between" value produced by averaging.

Restrictions:

This function cannot be used in an analytic context. That is, the OVER( ) clause is not allowed at all with this function.

The APPX_MEDIAN function returns only the first 10 characters for string values (string, varchar, char). Additional characters are truncated.

Examples:

The following example uses a table of a million random floating-point numbers ranging up to approximately 50,000. The average is approximately 25,000. Because of the random distribution, we would expect the median to be close to this same number. Computing the precise median is a more intensive operation than computing the average, because it requires keeping track of every distinct value and how many times each occurs. The APPX_MEDIAN( ) function uses a sampling algorithm to return an approximate result, which in this case is close to the expected value. To make sure that the value is not substantially out of range due to a skewed distribution, subsequent queries confirm that there are approximately 500,000 values higher than the APPX_MEDIAN( ) value, and approximately 500,000 values lower than the APPX_MEDIAN( ) value.

```
[localhost:21000] > select min(x), max(x), avg(x) from million_numbers;
+-------------------+-------------------+-------------------+
| min(x)            | max(x)            | avg(x)            |
+-------------------+-------------------+-------------------+
| 4.725693727250069 | 49994.56852674231 | 24945.38563793553 |
+-------------------+-------------------+-------------------+
[localhost:21000] > select appx_median(x) from million_numbers;
+----------------+
| appx_median(x) |
+----------------+
| 24721.6        |
+----------------+
[localhost:21000] > select count(x) as higher from million_numbers where x > (select
appx_median(x) from million_numbers);
+--------+
| higher |
+--------+
| 502013 |
+--------+
[localhost:21000] > select count(x) as lower from million_numbers where x < (select
appx_median(x) from million_numbers);
+--------+
| lower  |
+--------+
| 497987 |
+--------+
```

The following example computes the approximate median using a subset of the values from the table, and then confirms that the result is a reasonable estimate for the midpoint.

```
[localhost:21000] > select appx_median(x) from million_numbers where x between 1000 and
 5000;
+-------------------+
| appx_median(x)    |
+-------------------+
| 3013.107787358159 |
+-------------------+
```

```
[localhost:21000] > select count(x) as higher from million_numbers where x between 1000
 and 5000 and x > 3013.107787358159;
+--------+
| higher |
+--------+
| 37692  |
+--------+
[localhost:21000] > select count(x) as lower from million_numbers where x between 1000
 and 5000 and x < 3013.107787358159;
+-------+
| lower |
+-------+
| 37089 |
+-------+
```

AVG Function

An aggregate function that returns the average value from a set of numbers or TIMESTAMP values. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to AVG are NULL, AVG returns NULL.

Syntax:

```
AVG([DISTINCT | ALL] expression) [OVER (analytic_clause)]
```

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Return type: DOUBLE for numeric values; TIMESTAMP for TIMESTAMP values

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM( ) and AVG( ) are computed using the numeric R_NATIONKEY field, and the general-purpose MAX( ) and MIN( ) values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
```

```
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+---------------+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+------+-----------+----------------+---------------+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5             |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5             |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5             |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5             |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5             |
+-------------+-------+-----+------+-----------+----------------+---------------+
```

Examples:

```
-- Average all the non-NULL values in a column.
insert overwrite avg_t values (2),(4),(6),(null),(null);
-- The average of the above values is 4: (2+4+6) / 3. The 2 NULL values are ignored.
select avg(x) from avg_t;
-- Average only certain values from the column.
select avg(x) from t1 where month = 'January' and year = '2013';
-- Apply a calculation to the value of the column before averaging.
select avg(x/3) from t1;
-- Apply a function to the value of the column before averaging.
-- Here we are substituting a value of 0 for all NULLs in the column,
-- so that those rows do factor into the return value.
select avg(isnull(x,0)) from t1;
-- Apply some number-returning function to a string column and average the results.
-- If column s contains any NULLs, length(s) also returns NULL and those rows are ignored.
select avg(length(s)) from t1;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, avg(page_visits) from web_stats group by month, year;
-- Filter the input to eliminate duplicates before performing the calculation.
select avg(distinct x) from t1;
-- Filter the output after performing the calculation.
select avg(x) from t1 group by y having avg(x) between 1 and 20;
```

The following examples show how to use AVG( ) in an analytic context. They use a table containing integers from 1 to 10. Notice how the AVG( ) is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, avg(x) over (partition by property) as avg from int_t where property
 in ('odd','even');
+----+----------+-----+
| x  | property | avg |
+----+----------+-----+
| 2  | even     | 6   |
| 4  | even     | 6   |
| 6  | even     | 6   |
| 8  | even     | 6   |
| 10 | even     | 6   |
| 1  | odd      | 5   |
| 3  | odd      | 5   |
| 5  | odd      | 5   |
| 7  | odd      | 5   |
| 9  | odd      | 5   |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the "window"). The following examples use AVG( ) in an analytic context (that is, with an OVER( ) clause) to produce a running average of all the even values, then a running average of all the odd values. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property,
  avg(x) over (partition by property order by x) as 'cumulative average'
  from int_t where property in ('odd','even');
+----+----------+--------------------+
| x  | property | cumulative average |
+----+----------+--------------------+
| 2  | even     | 2                  |
| 4  | even     | 3                  |
| 6  | even     | 4                  |
| 8  | even     | 5                  |
| 10 | even     | 6                  |
| 1  | odd      | 1                  |
| 3  | odd      | 2                  |
| 5  | odd      | 3                  |
| 7  | odd      | 4                  |
| 9  | odd      | 5                  |
+----+----------+--------------------+

select x, property,
  avg(x) over
  (
    partition by property
    order by x
    range between unbounded preceding and current row
  ) as 'cumulative average'
from int_t where property in ('odd','even');
+----+----------+--------------------+
| x  | property | cumulative average |
+----+----------+--------------------+
| 2  | even     | 2                  |
| 4  | even     | 3                  |
| 6  | even     | 4                  |
| 8  | even     | 5                  |
| 10 | even     | 6                  |
| 1  | odd      | 1                  |
| 3  | odd      | 2                  |
| 5  | odd      | 3                  |
| 7  | odd      | 4                  |
| 9  | odd      | 5                  |
+----+----------+--------------------+

select x, property,
```

```
  avg(x) over
  (
    partition by property
    order by x
    rows between unbounded preceding and current row
  ) as 'cumulative average'
  from int_t where property in ('odd','even');
+----+----------+--------------------+
| x  | property | cumulative average |
+----+----------+--------------------+
| 2  | even     | 2                  |
| 4  | even     | 3                  |
| 6  | even     | 4                  |
| 8  | even     | 5                  |
| 10 | even     | 6                  |
| 1  | odd      | 1                  |
| 3  | odd      | 2                  |
| 5  | odd      | 3                  |
| 7  | odd      | 4                  |
| 9  | odd      | 5                  |
+----+----------+--------------------+
```

The following examples show how to construct a moving window, with a running average taking into account 1 row before and 1 row after the current row, within the same partition (all the even values or all the odd values). Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause:

```
select x, property,
  avg(x) over
  (
    partition by property
    order by x
    rows between 1 preceding and 1 following
  ) as 'moving average'
  from int_t where property in ('odd','even');
+----+----------+----------------+
| x  | property | moving average |
+----+----------+----------------+
| 2  | even     | 3              |
| 4  | even     | 4              |
| 6  | even     | 6              |
| 8  | even     | 8              |
| 10 | even     | 9              |
| 1  | odd      | 2              |
| 3  | odd      | 3              |
| 5  | odd      | 5              |
| 7  | odd      | 7              |
| 9  | odd      | 8              |
+----+----------+----------------+

-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  avg(x) over
  (
    partition by property
    order by x
    range between 1 preceding and 1 following
  ) as 'moving average'
from int_t where property in ('odd','even');
ERROR: AnalysisException: RANGE is only supported with both the lower and upper bounds
 UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM( ) and AVG( ) for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

Related information:

Impala Analytic Functions on page 506, MAX Function on page 490, MIN Function on page 493

COUNT Function

An aggregate function that returns the number of rows, or the number of non-NULL rows.

Syntax:

```
COUNT([DISTINCT | ALL] expression) [OVER (analytic_clause)]
```

Depending on the argument, COUNT( ) considers rows that meet certain conditions:

- The notation COUNT(*) includes NULL values in the total.
- The notation COUNT(*column_name*) only considers rows where the column contains a non-NULL value.
- You can also combine COUNT with the DISTINCT operator to eliminate duplicates before counting, and to count the combinations of values across multiple columns.

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Return type: BIGINT

Usage notes:

If you frequently run aggregate functions such as MIN( ), MAX( ), and COUNT(DISTINCT) on partition key columns, consider enabling the OPTIMIZE_PARTITION_KEY_SCANS query option, which optimizes such queries. This feature is available in CDH 5.7 / Impala 2.5 and higher. See OPTIMIZE_PARTITION_KEY_SCANS Query Option (CDH 5.7 or higher only) on page 348 for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM( ) and AVG( ) are computed using the numeric R_NATIONKEY field, and the general-purpose MAX( ) and MIN( ) values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
```

```
| AFRICA      | 15              |
| AFRICA      | 16              |
| AMERICA     | 1               |
| AMERICA     | 2               |
| AMERICA     | 3               |
| AMERICA     | 17              |
| AMERICA     | 24              |
| ASIA        | 8               |
| ASIA        | 9               |
| ASIA        | 12              |
| ASIA        | 18              |
| ASIA        | 21              |
| EUROPE      | 6               |
| EUROPE      | 7               |
| EUROPE      | 19              |
| EUROPE      | 22              |
| EUROPE      | 23              |
| MIDDLE EAST | 4               |
| MIDDLE EAST | 10              |
| MIDDLE EAST | 11              |
| MIDDLE EAST | 13              |
| MIDDLE EAST | 20              |
+-------------+-----------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+---------------+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+------+-----------+----------------+---------------+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5             |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5             |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5             |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5             |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5             |
+-------------+-------+-----+------+-----------+----------------+---------------+
```

Examples:

```
-- How many rows total are in the table, regardless of NULL values?
select count(*) from t1;
-- How many rows are in the table with non-NULL values for a column?
select count(c1) from t1;
-- Count the rows that meet certain conditions.
-- Again, * includes NULLs, so COUNT(*) might be greater than COUNT(col).
select count(*) from t1 where x > 10;
select count(c1) from t1 where x > 10;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Combine COUNT and DISTINCT to find the number of unique values.
-- Must use column names rather than * with COUNT(DISTINCT ...) syntax.
-- Rows with NULL values are not counted.
select count(distinct c1) from t1;
-- Rows with a NULL value in _either_ column are not counted.
select count(distinct c1, c2) from t1;
-- Return more than one result.
select month, year, count(distinct visitor_id) from web_stats group by month, year;
```

The following examples show how to use COUNT( ) in an analytic context. They use a table containing integers from 1 to 10. Notice how the COUNT( ) is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, count(x) over (partition by property) as count from int_t where
property in ('odd','even');
+----+----------+-------+
| x  | property | count |
+----+----------+-------+
| 2  | even     | 5     |
| 4  | even     | 5     |
| 6  | even     | 5     |
| 8  | even     | 5     |
| 10 | even     | 5     |
| 1  | odd      | 5     |
| 3  | odd      | 5     |
| 5  | odd      | 5     |
| 7  | odd      | 5     |
| 9  | odd      | 5     |
+----+----------+-------+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the "window"). The following examples use COUNT( ) in an analytic context (that is, with an OVER( ) clause) to produce a running count of all the even values, then a running count of all the odd values. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property,
  count(x) over (partition by property order by x) as 'cumulative count'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative count |
+----+----------+------------------+
| 2  | even     | 1                |
| 4  | even     | 2                |
| 6  | even     | 3                |
| 8  | even     | 4                |
| 10 | even     | 5                |
| 1  | odd      | 1                |
| 3  | odd      | 2                |
| 5  | odd      | 3                |
| 7  | odd      | 4                |
| 9  | odd      | 5                |
+----+----------+------------------+

select x, property,
  count(x) over
  (
    partition by property
    order by x
    range between unbounded preceding and current row
  ) as 'cumulative total'
from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative count |
+----+----------+------------------+
| 2  | even     | 1                |
| 4  | even     | 2                |
| 6  | even     | 3                |
| 8  | even     | 4                |
| 10 | even     | 5                |
| 1  | odd      | 1                |
| 3  | odd      | 2                |
| 5  | odd      | 3                |
| 7  | odd      | 4                |
| 9  | odd      | 5                |
+----+----------+------------------+

select x, property,
```

```
  count(x) over
  (
    partition by property
    order by x
    rows between unbounded preceding and current row
  ) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative count |
+----+----------+------------------+
| 2  | even     | 1                |
| 4  | even     | 2                |
| 6  | even     | 3                |
| 8  | even     | 4                |
| 10 | even     | 5                |
| 1  | odd      | 1                |
| 3  | odd      | 2                |
| 5  | odd      | 3                |
| 7  | odd      | 4                |
| 9  | odd      | 5                |
+----+----------+------------------+
```

The following examples show how to construct a moving window, with a running count taking into account 1 row before and 1 row after the current row, within the same partition (all the even values or all the odd values). Therefore, the count is consistently 3 for rows in the middle of the window, and 2 for rows near the ends of the window, where there is no preceding or no following row in the partition. Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause:

```
select x, property,
  count(x) over
  (
    partition by property
    order by x
    rows between 1 preceding and 1 following
  ) as 'moving total'
  from int_t where property in ('odd','even');
+----+----------+--------------+
| x  | property | moving total |
+----+----------+--------------+
| 2  | even     | 2            |
| 4  | even     | 3            |
| 6  | even     | 3            |
| 8  | even     | 3            |
| 10 | even     | 2            |
| 1  | odd      | 2            |
| 3  | odd      | 3            |
| 5  | odd      | 3            |
| 7  | odd      | 3            |
| 9  | odd      | 2            |
+----+----------+--------------+

-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  count(x) over
  (
    partition by property
    order by x
    range between 1 preceding and 1 following
  ) as 'moving total'
from int_t where property in ('odd','even');
ERROR: AnalysisException: RANGE is only supported with both the lower and upper bounds
 UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Related information:

Impala Analytic Functions on page 506

GROUP_CONCAT Function

An aggregate function that returns a single string representing the argument value concatenated together for each row of the result set. If the optional separator string is specified, the separator is added between each pair of concatenated values. The default separator is a comma followed by a space.

Syntax:

```
GROUP_CONCAT([ALL | DISTINCT] expression [, separator])
```

Usage notes: `concat()` and `concat_ws()` are appropriate for concatenating the values of multiple columns within the same row, while `group_concat()` joins together values from different rows.

By default, returns a single string covering the whole result set. To include other columns or values in the result set, or to produce multiple concatenated strings for subsets of rows, include a `GROUP BY` clause in the query.

Return type: `STRING`

This function cannot be used in an analytic context. That is, the `OVER()` clause is not allowed at all with this function.

Currently, Impala returns an error if the result value grows larger than 1 GiB.

Examples:

The following examples illustrate various aspects of the `GROUP_CONCAT()` function.

You can call the function directly on a `STRING` column. To use it with a numeric column, cast the value to `STRING`.

```
[localhost:21000] > create table t1 (x int, s string);
[localhost:21000] > insert into t1 values (1, "one"), (3, "three"), (2, "two"), (1,
"one");
[localhost:21000] > select group_concat(s) from t1;
+----------------------+
| group_concat(s)      |
+----------------------+
| one, three, two, one |
+----------------------+
[localhost:21000] > select group_concat(cast(x as string)) from t1;
+---------------------------------+
| group_concat(cast(x as string)) |
+---------------------------------+
| 1, 3, 2, 1                      |
+---------------------------------+
```

Specify the `DISTINCT` keyword to eliminate duplicate values from the concatenated result:

```
[localhost:21000] > select group_concat(distinct s) from t1;
+--------------------------+
| group_concat(distinct s) |
+--------------------------+
| three, two, one          |
+--------------------------+
```

The optional separator lets you format the result in flexible ways. The separator can be an arbitrary string expression, not just a single character.

```
[localhost:21000] > select group_concat(s,"|") from t1;
+----------------------+
| group_concat(s, '|') |
+----------------------+
| one|three|two|one    |
+----------------------+
[localhost:21000] > select group_concat(s,'---') from t1;
+-------------------------+
| group_concat(s, '---')  |
+-------------------------+
```

```
| one---three---two---one |
+-------------------------+
```

The default separator is a comma followed by a space. To get a comma-delimited result without extra spaces, specify a delimiter character that is only a comma.

```
[localhost:21000] > select group_concat(s,',') from t1;
+----------------------+
| group_concat(s, ',') |
+----------------------+
| one,three,two,one    |
+----------------------+
```

Including a GROUP BY clause lets you produce a different concatenated result for each group in the result set. In this example, the only X value that occurs more than once is 1, so that is the only row in the result set where GROUP_CONCAT( ) returns a delimited value. For groups containing a single value, GROUP_CONCAT( ) returns the original value of its STRING argument.

```
[localhost:21000] > select x, group_concat(s) from t1 group by x;
+---+-----------------+
| x | group_concat(s) |
+---+-----------------+
| 2 | two             |
| 3 | three           |
| 1 | one, one        |
+---+-----------------+
```

MAX Function

An aggregate function that returns the maximum value from a set of numbers. Opposite of the MIN function. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to MAX are NULL, MAX returns NULL.

Syntax:

```
MAX([DISTINCT | ALL] expression) [OVER (analytic_clause)]
```

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Restrictions: In Impala 2.0 and higher, this function can be used as an analytic function, but with restrictions on any window clause. For MAX( ) and MIN( ), the window clause is only allowed if the start bound is UNBOUNDED PRECEDING.

Return type: Same as the input value, except for CHAR and VARCHAR arguments which produce a STRING result

Usage notes:

If you frequently run aggregate functions such as MIN( ), MAX( ), and COUNT(DISTINCT) on partition key columns, consider enabling the OPTIMIZE_PARTITION_KEY_SCANS query option, which optimizes such queries. This feature is available in CDH 5.7 / Impala 2.5 and higher. See OPTIMIZE_PARTITION_KEY_SCANS Query Option (CDH 5.7 or higher only) on page 348 for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are

referenced using dot notation. Numeric values such as SUM( ) and AVG( ) are computed using the numeric R_NATIONKEY field, and the general-purpose MAX( ) and MIN( ) values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+---------------+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+------+-----------+----------------+---------------+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5             |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5             |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5             |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5             |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5             |
+-------------+-------+-----+------+-----------+----------------+---------------+
```

Examples:

```
-- Find the largest value for this column in the table.
select max(c1) from t1;
-- Find the largest value for this column from a subset of the table.
select max(c1) from t1 where month = 'January' and year = '2013';
-- Find the largest value from a set of numeric function results.
select max(length(s)) from t1;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, max(purchase_price) from store_stats group by month, year;
-- Filter the input to eliminate duplicates before performing the calculation.
select max(distinct x) from t1;
```

The following examples show how to use `MAX()` in an analytic context. They use a table containing integers from 1 to 10. Notice how the `MAX()` is reported for each input value, as opposed to the `GROUP BY` clause which condenses the result set.

```
select x, property, max(x) over (partition by property) as max from int_t where property
 in ('odd','even');
+----+----------+-----+
| x  | property | max |
+----+----------+-----+
| 2  | even     | 10  |
| 4  | even     | 10  |
| 6  | even     | 10  |
| 8  | even     | 10  |
| 10 | even     | 10  |
| 1  | odd      | 9   |
| 3  | odd      | 9   |
| 5  | odd      | 9   |
| 7  | odd      | 9   |
| 9  | odd      | 9   |
+----+----------+-----+
```

Adding an `ORDER BY` clause lets you experiment with results that are cumulative or apply to a moving set of rows (the "window"). The following examples use `MAX()` in an analytic context (that is, with an `OVER()` clause) to display the smallest value of `X` encountered up to each row in the result set. The examples use two columns in the `ORDER BY` clause to produce a sequence of values that rises and falls, to illustrate how the `MAX()` result only increases or stays the same throughout each partition within the result set. The basic `ORDER BY x` clause implicitly activates a window clause of `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, which is effectively the same as `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, therefore all of these examples produce the same results:

```
select x, property,
  max(x) over (order by property, x desc) as 'maximum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | maximum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 7                     |
| 3 | prime    | 7                     |
| 2 | prime    | 7                     |
| 9 | square   | 9                     |
| 4 | square   | 9                     |
| 1 | square   | 9                     |
+---+----------+-----------------------+

select x, property,
  max(x) over
  (
    order by property, x desc
    rows between unbounded preceding and current row
  ) as 'maximum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | maximum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
```

```
| 5 | prime    | 7                    |
| 3 | prime    | 7                    |
| 2 | prime    | 7                    |
| 9 | square   | 9                    |
| 4 | square   | 9                    |
| 1 | square   | 9                    |
+---+----------+----------------------+

select x, property,
  max(x) over
  (
    order by property, x desc
    range between unbounded preceding and current row
  ) as 'maximum to this point'
from int_t where property in ('prime','square');
+---+----------+----------------------+
| x | property | maximum to this point |
+---+----------+----------------------+
| 7 | prime    | 7                    |
| 5 | prime    | 7                    |
| 3 | prime    | 7                    |
| 2 | prime    | 7                    |
| 9 | square   | 9                    |
| 4 | square   | 9                    |
| 1 | square   | 9                    |
+---+----------+----------------------+
```

The following examples show how to construct a moving window, with a running maximum taking into account all rows before and 1 row after the current row. Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause. Because of an extra Impala restriction on the MAX( ) and MIN( ) functions in an analytic context, the lower bound must be UNBOUNDED PRECEDING.

```
select x, property,
  max(x) over
  (
    order by property, x
    rows between unbounded preceding and 1 following
  ) as 'local maximum'
from int_t where property in ('prime','square');
+---+----------+---------------+
| x | property | local maximum |
+---+----------+---------------+
| 2 | prime    | 3             |
| 3 | prime    | 5             |
| 5 | prime    | 7             |
| 7 | prime    | 7             |
| 1 | square   | 7             |
| 4 | square   | 9             |
| 9 | square   | 9             |
+---+----------+---------------+

-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  max(x) over
  (
    order by property, x
    range between unbounded preceding and 1 following
  ) as 'local maximum'
from int_t where property in ('prime','square');
ERROR: AnalysisException: RANGE is only supported with both the lower and upper bounds
 UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Related information:

Impala Analytic Functions on page 506, MIN Function on page 493, AVG Function on page 481

MIN Function

An aggregate function that returns the minimum value from a set of numbers. Opposite of the MAX function. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows

with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to MIN are NULL, MIN returns NULL.

Syntax:

```
MIN([DISTINCT | ALL] expression) [OVER (analytic_clause)]
```

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Restrictions: In Impala 2.0 and higher, this function can be used as an analytic function, but with restrictions on any window clause. For MAX( ) and MIN( ), the window clause is only allowed if the start bound is UNBOUNDED PRECEDING.

Return type: Same as the input value, except for CHAR and VARCHAR arguments which produce a STRING result

Usage notes:

If you frequently run aggregate functions such as MIN( ), MAX( ), and COUNT(DISTINCT) on partition key columns, consider enabling the OPTIMIZE_PARTITION_KEY_SCANS query option, which optimizes such queries. This feature is available in CDH 5.7 / Impala 2.5 and higher. See OPTIMIZE_PARTITION_KEY_SCANS Query Option (CDH 5.7 or higher only) on page 348 for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM( ) and AVG( ) are computed using the numeric R_NATIONKEY field, and the general-purpose MAX( ) and MIN( ) values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
```

```
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+-----+-----------+----------------+---------------+
| r_name      | count | sum | avg | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+-----+-----------+----------------+---------------+
| AFRICA      | 5     | 50  | 10  | ALGERIA   | MOZAMBIQUE     | 5             |
| AMERICA     | 5     | 47  | 9.4 | ARGENTINA | UNITED STATES  | 5             |
| ASIA        | 5     | 68  | 13.6| CHINA     | VIETNAM        | 5             |
| EUROPE      | 5     | 77  | 15.4| FRANCE    | UNITED KINGDOM | 5             |
| MIDDLE EAST | 5     | 58  | 11.6| EGYPT     | SAUDI ARABIA   | 5             |
+-------------+-------+-----+-----+-----------+----------------+---------------+
```

Examples:

```
-- Find the smallest value for this column in the table.
select min(c1) from t1;
-- Find the smallest value for this column from a subset of the table.
select min(c1) from t1 where month = 'January' and year = '2013';
-- Find the smallest value from a set of numeric function results.
select min(length(s)) from t1;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, min(purchase_price) from store_stats group by month, year;
-- Filter the input to eliminate duplicates before performing the calculation.
select min(distinct x) from t1;
```

The following examples show how to use `MIN()` in an analytic context. They use a table containing integers from 1 to 10. Notice how the `MIN()` is reported for each input value, as opposed to the `GROUP BY` clause which condenses the result set.

```
select x, property, min(x) over (partition by property) as min from int_t where property
 in ('odd','even');
+----+----------+-----+
| x  | property | min |
+----+----------+-----+
| 2  | even     | 2   |
| 4  | even     | 2   |
| 6  | even     | 2   |
| 8  | even     | 2   |
| 10 | even     | 2   |
| 1  | odd      | 1   |
| 3  | odd      | 1   |
| 5  | odd      | 1   |
| 7  | odd      | 1   |
| 9  | odd      | 1   |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the "window"). The following examples use MIN() in an analytic context (that is, with an OVER() clause) to display the smallest value of X encountered up to each row in the result set. The examples use two columns in the ORDER BY clause to produce a sequence of values that rises and falls, to illustrate how the MIN() result only decreases or stays the same throughout each partition within the result set. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property, min(x) over (order by property, x desc) as 'minimum to this point'
  from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | minimum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 5                     |
| 3 | prime    | 3                     |
| 2 | prime    | 2                     |
| 9 | square   | 2                     |
| 4 | square   | 2                     |
| 1 | square   | 1                     |
+---+----------+-----------------------+

select x, property,
  min(x) over
  (
    order by property, x desc
    range between unbounded preceding and current row
  ) as 'minimum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | minimum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 5                     |
| 3 | prime    | 3                     |
| 2 | prime    | 2                     |
| 9 | square   | 2                     |
| 4 | square   | 2                     |
| 1 | square   | 1                     |
+---+----------+-----------------------+

select x, property,
  min(x) over
  (
    order by property, x desc
    rows between unbounded preceding and current row
  ) as 'minimum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | minimum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 5                     |
| 3 | prime    | 3                     |
| 2 | prime    | 2                     |
| 9 | square   | 2                     |
| 4 | square   | 2                     |
| 1 | square   | 1                     |
+---+----------+-----------------------+
```

The following examples show how to construct a moving window, with a running minimum taking into account all rows before and 1 row after the current row. Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause. Because of an extra Impala restriction on the MAX() and MIN() functions in an analytic context, the lower bound must be UNBOUNDED PRECEDING.

```
select x, property,
  min(x) over
  (
    order by property, x desc
    rows between unbounded preceding and 1 following
```

```
  ) as 'local minimum'
from int_t where property in ('prime','square');
+---+----------+---------------+
| x | property | local minimum |
+---+----------+---------------+
| 7 | prime    | 5             |
| 5 | prime    | 3             |
| 3 | prime    | 2             |
| 2 | prime    | 2             |
| 9 | square   | 2             |
| 4 | square   | 1             |
| 1 | square   | 1             |
+---+----------+---------------+

-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  min(x) over
  (
    order by property, x desc
    range between unbounded preceding and 1 following
  ) as 'local minimum'
from int_t where property in ('prime','square');
ERROR: AnalysisException: RANGE is only supported with both the lower and upper bounds
 UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Related information:

Impala Analytic Functions on page 506, MAX Function on page 490, AVG Function on page 481

NDV Function

An aggregate function that returns an approximate value similar to the result of `COUNT(DISTINCT col)`, the "number of distinct values". It is much faster than the combination of `COUNT` and `DISTINCT`, and uses a constant amount of memory and thus is less memory-intensive for columns with high cardinality.

Syntax:

```
NDV([DISTINCT | ALL] expression)
```

Usage notes:

This is the mechanism used internally by the `COMPUTE STATS` statement for computing the number of distinct values in a column.

Because this number is an estimate, it might not reflect the precise number of different values in the column, especially if the cardinality is very low or very high. If the estimated number is higher than the number of rows in the table, Impala adjusts the value internally during query planning.

Return type: `DOUBLE` in Impala 2.0 and higher; `STRING` in earlier releases

Complex type considerations:

To access a column with a complex type (`ARRAY`, `STRUCT`, or `MAP`) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an `ARRAY` of `STRUCT` items). The array is unpacked inside the query using join notation. The array elements are referenced using the `ITEM` pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as `SUM()` and `AVG()` are computed using the numeric `R_NATIONKEY` field, and the general-purpose `MAX()` and `MIN()` values are computed from the string `N_NAME` field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
```

```
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+--------------+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+------+-----------+----------------+--------------+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5            |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5            |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5            |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5            |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5            |
+-------------+-------+-----+------+-----------+----------------+--------------+
```

Restrictions:

This function cannot be used in an analytic context. That is, the `OVER( )` clause is not allowed at all with this function.

Examples:

The following example queries a billion-row table to illustrate the relative performance of `COUNT(DISTINCT)` and `NDV()`. It shows how `COUNT(DISTINCT)` gives a precise answer, but is inefficient for large-scale data where an approximate result is sufficient. The `NDV()` function gives an approximate result but is much faster.

```
select count(distinct col1) from sample_data;
+---------------------+
| count(distinct col1)|
+---------------------+
| 100000              |
+---------------------+
Fetched 1 row(s) in 20.13s

select cast(ndv(col1) as bigint) as col1 from sample_data;
+----------+
| col1     |
+----------+
| 139017   |
+----------+
Fetched 1 row(s) in 8.91s
```

The following example shows how you can code multiple `NDV()` calls in a single query, to easily learn which columns have substantially more or fewer distinct values. This technique is faster than running a sequence of queries with `COUNT(DISTINCT)` calls.

```
select cast(ndv(col1) as bigint) as col1, cast(ndv(col2) as bigint) as col2,
    cast(ndv(col3) as bigint) as col3, cast(ndv(col4) as bigint) as col4
  from sample_data;
+----------+-----------+------------+-----------+
| col1     | col2      | col3       | col4      |
+----------+-----------+------------+-----------+
| 139017   | 282       | 46         | 145636240 |
+----------+-----------+------------+-----------+
Fetched 1 row(s) in 34.97s

select count(distinct col1) from sample_data;
+---------------------+
| count(distinct col1)|
+---------------------+
| 100000              |
+---------------------+
Fetched 1 row(s) in 20.13s

select count(distinct col2) from sample_data;
+---------------------+
| count(distinct col2) |
+---------------------+
| 278                 |
+---------------------+
Fetched 1 row(s) in 20.09s

select count(distinct col3) from sample_data;
+----------------------+
| count(distinct col3)  |
+----------------------+
| 46                   |
+----------------------+
Fetched 1 row(s) in 19.12s

select count(distinct col4) from sample_data;
+---------------------+
| count(distinct col4) |
+---------------------+
| 147135880           |
+---------------------+
Fetched 1 row(s) in 266.95s
```

STDDEV, STDDEV_SAMP, STDDEV_POP Functions

An aggregate function that returns the standard deviation of a set of numbers.

Syntax:

```
{ STDDEV | STDDEV_SAMP | STDDEV_POP } ([DISTINCT | ALL] expression)
```

This function works with any numeric data type.

Return type: DOUBLE in Impala 2.0 and higher; STRING in earlier releases

This function is typically used in mathematical formulas related to probability distributions.

The STDDEV_POP( ) and STDDEV_SAMP( ) functions compute the population standard deviation and sample standard deviation, respectively, of the input values. (STDDEV( ) is an alias for STDDEV_SAMP( ).) Both functions evaluate all input rows matched by the query. The difference is that STDDEV_SAMP( ) is scaled by 1/(N-1) while STDDEV_POP( ) is scaled by 1/N.

If no input rows match the query, the result of any of these functions is NULL. If a single input row matches the query, the result of any of these functions is "0.0".

Examples:

This example demonstrates how STDDEV( ) and STDDEV_SAMP( ) return the same result, while STDDEV_POP( ) uses a slightly different calculation to reflect that the input data is considered part of a larger “population”.

```
[localhost:21000] > select stddev(score) from test_scores;
+---------------+
| stddev(score) |
+---------------+
| 28.5          |
+---------------+
[localhost:21000] > select stddev_samp(score) from test_scores;
+--------------------+
| stddev_samp(score) |
+--------------------+
| 28.5               |
+--------------------+
[localhost:21000] > select stddev_pop(score) from test_scores;
+-------------------+
| stddev_pop(score) |
+-------------------+
| 28.4858           |
+-------------------+
```

This example demonstrates that, because the return value of these aggregate functions is a STRING, you must currently convert the result with CAST.

```
[localhost:21000] > create table score_stats as select cast(stddev(score) as decimal(7,4))
  `standard_deviation`, cast(variance(score) as decimal(7,4)) `variance` from test_scores;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc score_stats;
+--------------------+--------------+---------+
| name               | type         | comment |
+--------------------+--------------+---------+
| standard_deviation | decimal(7,4) |         |
| variance           | decimal(7,4) |         |
+--------------------+--------------+---------+
```

Restrictions:

This function cannot be used in an analytic context. That is, the OVER( ) clause is not allowed at all with this function.

Related information:

The STDDEV( ), STDDEV_POP( ), and STDDEV_SAMP( ) functions compute the standard deviation (square root of the variance) based on the results of VARIANCE( ), VARIANCE_POP( ), and VARIANCE_SAMP( ) respectively. See VARIANCE, VARIANCE_SAMP, VARIANCE_POP, VAR_SAMP, VAR_POP Functions on page 505 for details about the variance property.

SUM Function

An aggregate function that returns the sum of a set of numbers. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to MIN are NULL, SUM returns NULL.

Syntax:

```
SUM([DISTINCT | ALL] expression) [OVER (analytic_clause)]
```

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Return type: BIGINT for integer arguments, DOUBLE for floating-point arguments

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex Types (CDH 5.5 or higher only) on page 139 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM( ) and AVG( ) are computed using the numeric R_NATIONKEY field, and the general-purpose MAX( ) and MIN( ) values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
```

```
| MIDDLE EAST | 11              |
| MIDDLE EAST | 13              |
| MIDDLE EAST | 20              |
+-------------+-----------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+---------------+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct_vals |
+-------------+-------+-----+------+-----------+----------------+---------------+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5             |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5             |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5             |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5             |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5             |
+-------------+-------+-----+------+-----------+----------------+---------------+
```

Examples:

The following example shows how to use SUM( ) to compute the total for all the values in the table, a subset of values, or the sum for each combination of values in the GROUP BY clause:

```
-- Total all the values for this column in the table.
select sum(c1) from t1;
-- Find the total for this column from a subset of the table.
select sum(c1) from t1 where month = 'January' and year = '2013';
-- Find the total from a set of numeric function results.
select sum(length(s)) from t1;
-- Often used with functions that return predefined values to compute a score.
select sum(case when grade = 'A' then 1.0 when grade = 'B' then 0.75 else 0) as
class_honors from test_scores;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, sum(purchase_price) from store_stats group by month, year;
-- Filter the input to eliminate duplicates before performing the calculation.
select sum(distinct x) from t1;
```

The following examples show how to use SUM( ) in an analytic context. They use a table containing integers from 1 to 10. Notice how the SUM( ) is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, sum(x) over (partition by property) as sum from int_t where property
 in ('odd','even');
+----+----------+-----+
| x  | property | sum |
+----+----------+-----+
| 2  | even     | 30  |
| 4  | even     | 30  |
| 6  | even     | 30  |
| 8  | even     | 30  |
| 10 | even     | 30  |
| 1  | odd      | 25  |
| 3  | odd      | 25  |
| 5  | odd      | 25  |
| 7  | odd      | 25  |
| 9  | odd      | 25  |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the "window"). The following examples use SUM( ) in an analytic context (that is, with an OVER( ) clause) to produce a

running total of all the even values, then a running total of all the odd values. The basic `ORDER BY x` clause implicitly activates a window clause of `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, which is effectively the same as `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, therefore all of these examples produce the same results:

```
select x, property,
  sum(x) over (partition by property order by x) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 2  | even     | 2                |
| 4  | even     | 6                |
| 6  | even     | 12               |
| 8  | even     | 20               |
| 10 | even     | 30               |
| 1  | odd      | 1                |
| 3  | odd      | 4                |
| 5  | odd      | 9                |
| 7  | odd      | 16               |
| 9  | odd      | 25               |
+----+----------+------------------+

select x, property,
  sum(x) over
  (
    partition by property
    order by x
    range between unbounded preceding and current row
  ) as 'cumulative total'
from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 2  | even     | 2                |
| 4  | even     | 6                |
| 6  | even     | 12               |
| 8  | even     | 20               |
| 10 | even     | 30               |
| 1  | odd      | 1                |
| 3  | odd      | 4                |
| 5  | odd      | 9                |
| 7  | odd      | 16               |
| 9  | odd      | 25               |
+----+----------+------------------+

select x, property,
  sum(x) over
  (
    partition by property
    order by x
    rows between unbounded preceding and current row
  ) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 2  | even     | 2                |
| 4  | even     | 6                |
| 6  | even     | 12               |
| 8  | even     | 20               |
| 10 | even     | 30               |
| 1  | odd      | 1                |
| 3  | odd      | 4                |
| 5  | odd      | 9                |
| 7  | odd      | 16               |
| 9  | odd      | 25               |
+----+----------+------------------+
```

Changing the direction of the ORDER BY clause causes the intermediate results of the cumulative total to be calculated in a different order:

```
select sum(x) over (partition by property order by x desc) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 10 | even     | 10               |
| 8  | even     | 18               |
| 6  | even     | 24               |
| 4  | even     | 28               |
| 2  | even     | 30               |
| 9  | odd      | 9                |
| 7  | odd      | 16               |
| 5  | odd      | 21               |
| 3  | odd      | 24               |
| 1  | odd      | 25               |
+----+----------+------------------+
```

The following examples show how to construct a moving window, with a running total taking into account 1 row before and 1 row after the current row, within the same partition (all the even values or all the odd values). Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause:

```
select x, property,
  sum(x) over
  (
    partition by property
    order by x
    rows between 1 preceding and 1 following
  ) as 'moving total'
  from int_t where property in ('odd','even');
+----+----------+--------------+
| x  | property | moving total |
+----+----------+--------------+
| 2  | even     | 6            |
| 4  | even     | 12           |
| 6  | even     | 18           |
| 8  | even     | 24           |
| 10 | even     | 18           |
| 1  | odd      | 4            |
| 3  | odd      | 9            |
| 5  | odd      | 15           |
| 7  | odd      | 21           |
| 9  | odd      | 16           |
+----+----------+--------------+

-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  sum(x) over
  (
    partition by property
    order by x
    range between 1 preceding and 1 following
  ) as 'moving total'
from int_t where property in ('odd','even');
ERROR: AnalysisException: RANGE is only supported with both the lower and upper bounds
 UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM( ) and AVG( ) for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

Related information:

Impala Analytic Functions on page 506

VARIANCE, VARIANCE_SAMP, VARIANCE_POP, VAR_SAMP, VAR_POP Functions

An aggregate function that returns the variance of a set of numbers. This is a mathematical property that signifies how far the values spread apart from the mean. The return value can be zero (if the input is a single value, or a set of identical values), or a positive number otherwise.

Syntax:

```
{ VARIANCE | VAR[IANCE]_SAMP | VAR[IANCE]_POP } ([DISTINCT | ALL] expression)
```

This function works with any numeric data type.

Return type: DOUBLE in Impala 2.0 and higher; STRING in earlier releases

This function is typically used in mathematical formulas related to probability distributions.

The VARIANCE_SAMP() and VARIANCE_POP() functions compute the sample variance and population variance, respectively, of the input values. (VARIANCE() is an alias for VARIANCE_SAMP().) Both functions evaluate all input rows matched by the query. The difference is that STDDEV_SAMP() is scaled by 1/(N-1) while STDDEV_POP() is scaled by 1/N.

The functions VAR_SAMP() and VAR_POP() are the same as VARIANCE_SAMP() and VARIANCE_POP(), respectively. These aliases are available in Impala 2.0 and later.

If no input rows match the query, the result of any of these functions is NULL. If a single input row matches the query, the result of any of these functions is "0.0".

Examples:

This example demonstrates how VARIANCE() and VARIANCE_SAMP() return the same result, while VARIANCE_POP() uses a slightly different calculation to reflect that the input data is considered part of a larger “population”.

```
[localhost:21000] > select variance(score) from test_scores;
+-----------------+
| variance(score) |
+-----------------+
| 812.25          |
+-----------------+
[localhost:21000] > select variance_samp(score) from test_scores;
+----------------------+
| variance_samp(score) |
+----------------------+
| 812.25               |
+----------------------+
[localhost:21000] > select variance_pop(score) from test_scores;
+---------------------+
| variance_pop(score) |
+---------------------+
| 811.438             |
+---------------------+
```

This example demonstrates that, because the return value of these aggregate functions is a STRING, you convert the result with CAST if you need to do further calculations as a numeric value.

```
[localhost:21000] > create table score_stats as select cast(stddev(score) as decimal(7,4))
  `standard_deviation`, cast(variance(score) as decimal(7,4)) `variance` from test_scores;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc score_stats;
+--------------------+--------------+---------+
| name               | type         | comment |
+--------------------+--------------+---------+
| standard_deviation | decimal(7,4) |         |
```

```
| variance           | decimal(7,4) |         |
+--------------------+--------------+---------+
```

Restrictions:

This function cannot be used in an analytic context. That is, the OVER( ) clause is not allowed at all with this function.

Related information:

The STDDEV( ), STDDEV_POP( ), and STDDEV_SAMP( ) functions compute the standard deviation (square root of the variance) based on the results of VARIANCE( ), VARIANCE_POP( ), and VARIANCE_SAMP( ) respectively. See STDDEV, STDDEV_SAMP, STDDEV_POP Functions on page 499 for details about the standard deviation property.

Impala Analytic Functions

Analytic functions (also known as window functions) are a special category of built-in functions. Like aggregate functions, they examine the contents of multiple input rows to compute each output value. However, rather than being limited to one result value per GROUP BY group, they operate on ***windows*** where the input rows are ordered and grouped using flexible conditions expressed through an OVER( ) clause.

Added in: CDH 5.2.0 / Impala 2.0.0

Some functions, such as LAG( ) and RANK( ), can only be used in this analytic context. Some aggregate functions do double duty: when you call the aggregation functions such as MAX( ), SUM( ), AVG( ), and so on with an OVER( ) clause, they produce an output value for each row, based on computations across other rows in the window.

Although analytic functions often compute the same value you would see from an aggregate function in a GROUP BY query, the analytic functions produce a value for each row in the result set rather than a single value for each group. This flexibility lets you include additional columns in the SELECT list, offering more opportunities for organizing and filtering the result set.

Analytic function calls are only allowed in the SELECT list and in the outermost ORDER BY clause of the query. During query processing, analytic functions are evaluated after other query stages such as joins, WHERE, and GROUP BY,

The rows that are part of each partition are analyzed by computations across an ordered or unordered set of rows. For example, COUNT( ) and SUM( ) might be applied to all the rows in the partition, in which case the order of analysis does not matter. The ORDER BY clause might be used inside the OVER( ) clause to defines the ordering that applies to functions such as LAG( ) and FIRST_VALUE( ).

Analytic functions are frequently used in fields such as finance and science to provide trend, outlier, and bucketed analysis for large data sets. You might also see the term “window functions” in database literature, referring to the sequence of rows (the “window”) that the function call applies to, particularly when the OVER clause includes a ROWS or RANGE keyword.

The following sections describe the analytic query clauses and the pure analytic functions provided by Impala. For usage information about aggregate functions in an analytic context, see Impala Aggregate Functions on page 479.

OVER Clause

The OVER clause is required for calls to pure analytic functions such as LEAD( ), RANK( ), and FIRST_VALUE( ). When you include an OVER clause with calls to aggregate functions such as MAX( ), COUNT( ), or SUM( ), they operate as analytic functions.

Syntax:

```
function(args) OVER([partition_by_clause] [order_by_clause [window_clause]])

partition_by_clause ::= PARTITION BY expr [, expr ...]
order_by_clause ::= ORDER BY expr  [ASC | DESC] [NULLS FIRST | NULLS LAST] [, expr [ASC
  | DESC] [NULLS FIRST | NULLS LAST] ...]
window_clause: See Window Clause
```

PARTITION BY clause:

The PARTITION BY clause acts much like the GROUP BY clause in the outermost block of a query. It divides the rows into groups containing identical values in one or more columns. These logical groups are known as ***partitions***. Throughout the discussion of analytic functions, "partitions" refers to the groups produced by the PARTITION BY clause, not to partitioned tables. However, note the following limitation that applies specifically to analytic function calls involving partitioned tables.

In queries involving both analytic functions and partitioned tables, partition pruning only occurs for columns named in the PARTITION BY clause of the analytic function call. For example, if an analytic function query has a clause such as WHERE year=2016, the way to make the query prune all other YEAR partitions is to include PARTITION BY year in the analytic function call; for example, OVER (PARTITION BY year,*other_columns other_analytic_clauses*).

The sequence of results from an analytic function "resets" for each new partition in the result set. That is, the set of preceding or following rows considered by the analytic function always come from a single partition. Any MAX(), SUM(), ROW_NUMBER(), and so on apply to each partition independently. Omit the PARTITION BY clause to apply the analytic operation to all the rows in the table.

ORDER BY clause:

The ORDER BY clause works much like the ORDER BY clause in the outermost block of a query. It defines the order in which rows are evaluated for the entire input set, or for each group produced by a PARTITION BY clause. You can order by one or multiple expressions, and for each expression optionally choose ascending or descending order and whether nulls come first or last in the sort order. Because this ORDER BY clause only defines the order in which rows are evaluated, if you want the results to be output in a specific order, also include an ORDER BY clause in the outer block of the query.

When the ORDER BY clause is omitted, the analytic function applies to all items in the group produced by the PARTITION BY clause. When the ORDER BY clause is included, the analysis can apply to all or a subset of the items in the group, depending on the optional window clause.

The order in which the rows are analyzed is only defined for those columns specified in ORDER BY clauses.

One difference between the analytic and outer uses of the ORDER BY clause: inside the OVER clause, ORDER BY 1 or other integer value is interpreted as a constant sort value (effectively a no-op) rather than referring to column 1.

Window clause:

The window clause is only allowed in combination with an ORDER BY clause. If the ORDER BY clause is specified but the window clause is not, the default window is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. See Window Clause on page 508 for full details.

HBase considerations:

Because HBase tables are optimized for single-row lookups rather than full scans, analytic functions using the OVER() clause are not recommended for HBase tables. Although such queries work, their performance is lower than on comparable tables using HDFS data files.

Parquet considerations:

Analytic functions are very efficient for Parquet tables. The data that is examined during evaluation of the OVER() clause comes from a specified set of columns, and the values for each column are arranged sequentially within each data file.

Text table considerations:

Analytic functions are convenient to use with text tables for exploratory business intelligence. When the volume of data is substantial, prefer to use Parquet tables for performance-critical analytic queries.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example shows how to synthesize a numeric sequence corresponding to all the rows in a table. The new table has the same columns as the old one, plus an additional column `ID` containing the integers 1, 2, 3, and so on, corresponding to the order of a `TIMESTAMP` column in the original table.

```
CREATE TABLE events_with_id AS
  SELECT
    row_number() OVER (ORDER BY date_and_time) AS id,
    c1, c2, c3, c4
  FROM events;
```

The following example shows how to determine the number of rows containing each value for a column. Unlike a corresponding `GROUP BY` query, this one can analyze a single column and still return all values (not just the distinct ones) from the other columns.

```
SELECT x, y, z,
  count() OVER (PARTITION BY x) AS how_many_x
FROM t1;
```

Restrictions:

You cannot directly combine the `DISTINCT` operator with analytic function calls. You can put the analytic function call in a `WITH` clause or an inline view, and apply the `DISTINCT` operator to its result set.

```
WITH t1 AS (SELECT x, sum(x) OVER (PARTITION BY x) AS total FROM t1)
  SELECT DISTINCT x, total FROM t1;
```

Window Clause

Certain analytic functions accept an optional ***window clause***, which makes the function analyze only certain rows "around" the current row rather than all rows in the partition. For example, you can get a moving average by specifying some number of preceding and following rows, or a running count or running total by specifying all rows up to the current position. This clause can result in different analytic results for rows within the same partition.

The window clause is supported with the `AVG()`, `COUNT()`, `FIRST_VALUE()`, `LAST_VALUE()`, and `SUM()` functions. For `MAX()` and `MIN()`, the window clause only allowed if the start bound is `UNBOUNDED PRECEDING`

Syntax:

```
ROWS BETWEEN [ { m | UNBOUNDED } PRECEDING | CURRENT ROW] [ AND [CURRENT ROW | { UNBOUNDED
  | n } FOLLOWING] ]
RANGE BETWEEN [ {m | UNBOUNDED } PRECEDING | CURRENT ROW] [ AND [CURRENT ROW | { UNBOUNDED
  | n } FOLLOWING] ]
```

`ROWS BETWEEN` defines the size of the window in terms of the indexes of the rows in the result set. The size of the window is predictable based on the clauses the position within the result set.

`RANGE BETWEEN` does not currently support numeric arguments to define a variable-size sliding window.

Currently, Impala supports only some combinations of arguments to the `RANGE` clause:

- `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (the default when `ORDER BY` is specified and the window clause is omitted)
- `RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`
- `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`

When `RANGE` is used, `CURRENT ROW` includes not just the current row but all rows that are tied with the current row based on the `ORDER BY` expressions.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following examples show financial data for a fictional stock symbol JDR. The closing price moves up and down each day.

```
create table stock_ticker (stock_symbol string, closing_price decimal(8,2), closing_date
 timestamp);
...load some data...
select * from stock_ticker order by stock_symbol, closing_date
+--------------+---------------+---------------------+
| stock_symbol | closing_price | closing_date        |
+--------------+---------------+---------------------+
| JDR          | 12.86         | 2014-10-02 00:00:00 |
| JDR          | 12.89         | 2014-10-03 00:00:00 |
| JDR          | 12.94         | 2014-10-04 00:00:00 |
| JDR          | 12.55         | 2014-10-05 00:00:00 |
| JDR          | 14.03         | 2014-10-06 00:00:00 |
| JDR          | 14.75         | 2014-10-07 00:00:00 |
| JDR          | 13.98         | 2014-10-08 00:00:00 |
+--------------+---------------+---------------------+
```

The queries use analytic functions with window clauses to compute moving averages of the closing price. For example, ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING produces an average of the value from a 3-day span, producing a different value for each row. The first row, which has no preceding row, only gets averaged with the row following it. If the table contained more than one stock symbol, the PARTITION BY clause would limit the window for the moving average to only consider the prices for a single stock.

```
select stock_symbol, closing_date, closing_price,
  avg(closing_price) over (partition by stock_symbol order by closing_date
    rows between 1 preceding and 1 following) as moving_average
  from stock_ticker;
+--------------+---------------------+---------------+----------------+
| stock_symbol | closing_date        | closing_price | moving_average |
+--------------+---------------------+---------------+----------------+
| JDR          | 2014-10-02 00:00:00 | 12.86         | 12.87          |
| JDR          | 2014-10-03 00:00:00 | 12.89         | 12.89          |
| JDR          | 2014-10-04 00:00:00 | 12.94         | 12.79          |
| JDR          | 2014-10-05 00:00:00 | 12.55         | 13.17          |
| JDR          | 2014-10-06 00:00:00 | 14.03         | 13.77          |
| JDR          | 2014-10-07 00:00:00 | 14.75         | 14.25          |
| JDR          | 2014-10-08 00:00:00 | 13.98         | 14.36          |
+--------------+---------------------+---------------+----------------+
```

The clause ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW produces a cumulative moving average, from the earliest data up to the value for each day.

```
select stock_symbol, closing_date, closing_price,
  avg(closing_price) over (partition by stock_symbol order by closing_date
    rows between unbounded preceding and current row) as moving_average
  from stock_ticker;
+--------------+---------------------+---------------+----------------+
| stock_symbol | closing_date        | closing_price | moving_average |
+--------------+---------------------+---------------+----------------+
| JDR          | 2014-10-02 00:00:00 | 12.86         | 12.86          |
| JDR          | 2014-10-03 00:00:00 | 12.89         | 12.87          |
| JDR          | 2014-10-04 00:00:00 | 12.94         | 12.89          |
| JDR          | 2014-10-05 00:00:00 | 12.55         | 12.81          |
| JDR          | 2014-10-06 00:00:00 | 14.03         | 13.05          |
| JDR          | 2014-10-07 00:00:00 | 14.75         | 13.33          |
| JDR          | 2014-10-08 00:00:00 | 13.98         | 13.42          |
+--------------+---------------------+---------------+----------------+
```

AVG Function - Analytic Context

You can include an OVER clause with a call to this function to use it as an analytic function. See AVG Function on page 481 for details and examples.

COUNT Function - Analytic Context

You can include an OVER clause with a call to this function to use it as an analytic function. See COUNT Function on page 485 for details and examples.

CUME_DIST Function (CDH 5.5 or higher only)

Returns the cumulative distribution of a value. The value for each row in the result set is greater than 0 and less than or equal to 1.

Syntax:

```
CUME_DIST (expr)
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Within each partition of the result set, the CUME_DIST() value represents an ascending sequence that ends at 1. Each value represents the proportion of rows in the partition whose values are less than or equal to the value in the current row.

If the sequence of input values contains ties, the CUME_DIST() results are identical for the tied values.

Impala only supports the CUME_DIST() function in an analytic context, not as a regular aggregate function.

Examples:

This example uses a table with 9 rows. The CUME_DIST() function evaluates the entire table because there is no PARTITION BY clause, with the rows ordered by the weight of the animal. the sequence of values shows that 1/9 of the values are less than or equal to the lightest animal (mouse), 2/9 of the values are less than or equal to the second-lightest animal, and so on up to the heaviest animal (elephant), where 9/9 of the rows are less than or equal to its weight.

```
create table animals (name string, kind string, kilos decimal(9,3));
insert into animals values
  ('Elephant', 'Mammal', 4000), ('Giraffe', 'Mammal', 1200), ('Mouse', 'Mammal', 0.020),

  ('Condor', 'Bird', 15), ('Horse', 'Mammal', 500), ('Owl', 'Bird', 2.5),
  ('Ostrich', 'Bird', 145), ('Polar bear', 'Mammal', 700), ('Housecat', 'Mammal', 5);

select name, cume_dist() over (order by kilos) from animals;
+------------+----------------------+
| name       | cume_dist() OVER(...) |
+------------+----------------------+
| Elephant   | 1                    |
| Giraffe    | 0.8888888888888888   |
| Polar bear | 0.7777777777777778   |
| Horse      | 0.6666666666666666   |
| Ostrich    | 0.5555555555555556   |
| Condor     | 0.4444444444444444   |
| Housecat   | 0.3333333333333333   |
| Owl        | 0.2222222222222222   |
| Mouse      | 0.1111111111111111   |
+------------+----------------------+
```

Using a PARTITION BY clause produces a separate sequence for each partition group, in this case one for mammals and one for birds. Because there are 3 birds and 6 mammals, the sequence illustrates how 1/3 of the “Bird” rows have a kilos value that is less than or equal to the lightest bird, 1/6 of the “Mammal” rows have a kilos value that is less than or equal to the lightest mammal, and so on until both the heaviest bird and heaviest mammal have a CUME_DIST() value of 1.

```
select name, kind, cume_dist() over (partition by kind order by kilos) from animals
+------------+--------+----------------------+
| name       | kind   | cume_dist() OVER(...) |
+------------+--------+----------------------+
```

```
| Ostrich    | Bird   | 1                    |
| Condor     | Bird   | 0.6666666666666666   |
| Owl        | Bird   | 0.3333333333333333   |
| Elephant   | Mammal | 1                    |
| Giraffe    | Mammal | 0.8333333333333334   |
| Polar bear | Mammal | 0.6666666666666666   |
| Horse      | Mammal | 0.5                  |
| Housecat   | Mammal | 0.3333333333333333   |
| Mouse      | Mammal | 0.1666666666666667   |
+------------+--------+----------------------+
```

We can reverse the ordering within each partition group by using an `ORDER BY ... DESC` clause within the `OVER()` clause. Now the lightest (smallest value of `kilos`) animal of each kind has a `CUME_DIST()` value of 1.

```
select name, kind, cume_dist() over (partition by kind order by kilos desc) from animals
+------------+--------+----------------------+
| name       | kind   | cume_dist() OVER(...) |
+------------+--------+----------------------+
| Owl        | Bird   | 1                    |
| Condor     | Bird   | 0.6666666666666666   |
| Ostrich    | Bird   | 0.3333333333333333   |
| Mouse      | Mammal | 1                    |
| Housecat   | Mammal | 0.8333333333333334   |
| Horse      | Mammal | 0.6666666666666666   |
| Polar bear | Mammal | 0.5                  |
| Giraffe    | Mammal | 0.3333333333333333   |
| Elephant   | Mammal | 0.1666666666666667   |
+------------+--------+----------------------+
```

The following example manufactures some rows with identical values in the `kilos` column, to demonstrate how the results look in case of tie values. For simplicity, it only shows the `CUME_DIST()` sequence for the “Bird” rows. Now with 3 rows all with a value of 15, all of those rows have the same `CUME_DIST()` value. 4/5 of the rows have a value for `kilos` that is less than or equal to 15.

```
insert into animals values ('California Condor', 'Bird', 15), ('Andean Condor', 'Bird',
 15)

select name, kind, cume_dist() over (order by kilos) from animals where kind = 'Bird';
+------------------+------+----------------------+
| name             | kind | cume_dist() OVER(...) |
+------------------+------+----------------------+
| Ostrich          | Bird | 1                    |
| Condor           | Bird | 0.8                  |
| California Condor | Bird | 0.8                  |
| Andean Condor    | Bird | 0.8                  |
| Owl              | Bird | 0.2                  |
+------------------+------+----------------------+
```

The following example shows how to use an `ORDER BY` clause in the outer block to order the result set in case of ties. Here, all the “Bird” rows are together, then in descending order by the result of the `CUME_DIST()` function, and all tied `CUME_DIST()` values are ordered by the animal name.

```
select name, kind, cume_dist() over (partition by kind order by kilos) as ordering
  from animals
where
  kind = 'Bird'
order by kind, ordering desc, name;
+------------------+------+----------+
| name             | kind | ordering |
+------------------+------+----------+
| Ostrich          | Bird | 1        |
| Andean Condor    | Bird | 0.8      |
| California Condor | Bird | 0.8      |
| Condor           | Bird | 0.8      |
| Owl              | Bird | 0.2      |
+------------------+------+----------+
```

DENSE_RANK Function

Returns an ascending sequence of integers, starting with 1. The output sequence produces duplicate integers for duplicate values of the ORDER BY expressions. After generating duplicate output values for the "tied" input values, the function continues the sequence with the next higher integer. Therefore, the sequence contains duplicates but no gaps when the input contains duplicates. Starts the sequence over for each group produced by the PARTITIONED BY clause.

Syntax:

```
DENSE_RANK() OVER([partition_by_clause] order_by_clause)
```

The PARTITION BY clause is optional. The ORDER BY clause is required. The window clause is not allowed.

Usage notes:

Often used for top-N and bottom-N queries. For example, it could produce a "top 10" report including all the items with the 10 highest values, even if several items tied for 1st place.

Similar to ROW_NUMBER and RANK. These functions differ in how they treat duplicate combinations of values.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example demonstrates how the DENSE_RANK() function identifies where each value "places" in the result set, producing the same result for duplicate values, but with a strict sequence from 1 to the number of groups. For example, when results are ordered by the X column, both 1 values are tied for first; both 2 values are tied for second; and so on.

```
select x, dense_rank() over(order by x) as rank, property from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | square   |
| 1  | 1    | odd      |
| 2  | 2    | even     |
| 2  | 2    | prime    |
| 3  | 3    | prime    |
| 3  | 3    | odd      |
| 4  | 4    | even     |
| 4  | 4    | square   |
| 5  | 5    | odd      |
| 5  | 5    | prime    |
| 6  | 6    | even     |
| 6  | 6    | perfect  |
| 7  | 7    | lucky    |
| 7  | 7    | lucky    |
| 7  | 7    | lucky    |
| 7  | 7    | odd      |
| 7  | 7    | prime    |
| 8  | 8    | even     |
| 9  | 9    | square   |
| 9  | 9    | odd      |
| 10 | 10   | round    |
| 10 | 10   | even     |
+----+------+----------+
```

The following examples show how the DENSE_RANK() function is affected by the PARTITION property within the ORDER BY clause.

Partitioning by the PROPERTY column groups all the even, odd, and so on values together, and DENSE_RANK() returns the place of each value within the group, producing several ascending sequences.

```
select x, dense_rank() over(partition by property order by x) as rank, property from
int_t;
+----+------+----------+
| x  | rank | property |
```

```
+----+------+----------+
| 2  | 1    | even     |
| 4  | 2    | even     |
| 6  | 3    | even     |
| 8  | 4    | even     |
| 10 | 5    | even     |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 1  | 1    | odd      |
| 3  | 2    | odd      |
| 5  | 3    | odd      |
| 7  | 4    | odd      |
| 9  | 5    | odd      |
| 6  | 1    | perfect  |
| 2  | 1    | prime    |
| 3  | 2    | prime    |
| 5  | 3    | prime    |
| 7  | 4    | prime    |
| 10 | 1    | round    |
| 1  | 1    | square   |
| 4  | 2    | square   |
| 9  | 3    | square   |
+----+------+----------+
```

Partitioning by the X column groups all the duplicate numbers together and returns the place each value within the group; because each value occurs only 1 or 2 times, DENSE_RANK( ) designates each X value as either first or second within its group.

```
select x, dense_rank() over(partition by x order by property) as rank, property from
int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | odd      |
| 1  | 2    | square   |
| 2  | 1    | even     |
| 2  | 2    | prime    |
| 3  | 1    | odd      |
| 3  | 2    | prime    |
| 4  | 1    | even     |
| 4  | 2    | square   |
| 5  | 1    | odd      |
| 5  | 2    | prime    |
| 6  | 1    | even     |
| 6  | 2    | perfect  |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 2    | odd      |
| 7  | 3    | prime    |
| 8  | 1    | even     |
| 9  | 1    | odd      |
| 9  | 2    | square   |
| 10 | 1    | even     |
| 10 | 2    | round    |
+----+------+----------+
```

The following example shows how DENSE_RANK( ) produces a continuous sequence while still allowing for ties. In this case, Croesus and Midas both have the second largest fortune, while Crassus has the third largest. (In RANK Function on page 521, you see a similar query with the RANK( ) function that shows that while Crassus has the third largest fortune, he is the fourth richest person.)

```
select dense_rank() over (order by net_worth desc) as placement, name, net_worth from
wealth order by placement, name;
+-----------+---------+---------------+
| placement | name    | net_worth     |
+-----------+---------+---------------+
| 1         | Solomon | 2000000000.00 |
| 2         | Croesus | 1000000000.00 |
```

```
| 2         | Midas   | 1000000000.00 |
| 3         | Crassus | 500000000.00  |
| 4         | Scrooge | 80000000.00   |
+-----------+---------+---------------+
```

Related information:

RANK Function on page 521, ROW_NUMBER Function on page 523

FIRST_VALUE Function

Returns the expression value from the first row in the window. If your table has null values, you can use the `IGNORE NULLS` clause to return the first non-null value from the window. This same value is repeated for all result rows for the group. The return value is `NULL` if the input expression is `NULL`.

Syntax:

```
FIRST_VALUE(expr [IGNORE NULLS]) OVER([partition_by_clause] order_by_clause
[window_clause])
```

The `PARTITION BY` clause is optional. The `ORDER BY` clause is required. The window clause is optional.

Usage notes:

If any duplicate values occur in the tuples evaluated by the `ORDER BY` clause, the result of this function is not deterministic. Consider adding additional `ORDER BY` columns to ensure consistent ordering.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example shows a table with a wide variety of country-appropriate greetings. For consistency, we want to standardize on a single greeting for each country. The `FIRST_VALUE()` function helps to produce a mail merge report where every person from the same country is addressed with the same greeting.

```
select name, country, greeting from mail_merge;
+---------+---------+--------------+
| name    | country | greeting     |
+---------+---------+--------------+
| Pete    | USA     | Hello        |
| John    | USA     | Hi           |
| Boris   | Germany | Guten tag    |
| Michael | Germany | Guten morgen |
| Bjorn   | Sweden  | Hej          |
| Mats    | Sweden  | Tja          |
+---------+---------+--------------+

select country, name,
  first_value(greeting)
    over (partition by country order by name, greeting) as greeting
  from mail_merge;
+---------+---------+-----------+
| country | name    | greeting  |
+---------+---------+-----------+
| Germany | Boris   | Guten tag |
| Germany | Michael | Guten tag |
| Sweden  | Bjorn   | Hej       |
| Sweden  | Mats    | Hej       |
| USA     | John    | Hi        |
| USA     | Pete    | Hi        |
+---------+---------+-----------+
```

Changing the order in which the names are evaluated changes which greeting is applied to each group.

```
select country, name,
  first_value(greeting)
    over (partition by country order by name desc, greeting) as greeting
  from mail_merge;
```

```
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Germany | Michael | Guten morgen |
| Germany | Boris   | Guten morgen |
| Sweden  | Mats    | Tja          |
| Sweden  | Bjorn   | Tja          |
| USA     | Pete    | Hello        |
| USA     | John    | Hello        |
+---------+---------+--------------+
```

If you introduce null values in the `mail_merge` table, the `FIRST_VALUE()` function will produce a different result with the `IGNORE NULLS` clause.

```
select * from mail_merge;
+---------+---------+--------------+
| name    | country | greeting     |
+---------+---------+--------------+
| Boris   | Germany | Guten tag    |
| Peng    | China   | Nihao        |
| Mats    | Sweden  | Tja          |
| Bjorn   | Sweden  | Hej          |
| Kei     | Japan   | NULL         |
| Li      | China   | NULL         |
| John    | USA     | Hi           |
| Pete    | USA     | Hello        |
| Michael | Germany | Guten morgen |
+---------+---------+--------------+

select country, name,
  first_value(greeting ignore nulls)
    over (partition by country order by name,greeting) as greeting
  from mail_merge;
+---------+---------+-----------+
| country | name    | greeting  |
+---------+---------+-----------+
| Japan   | Kei     | NULL      |
| Germany | Boris   | Guten tag |
| Germany | Michael | Guten tag |
| China   | Li      | NULL      |
| China   | Peng    | Nihao     |
| Sweden  | Bjorn   | Hej       |
| Sweden  | Mats    | Hej       |
| USA     | John    | Hi        |
| USA     | Pete    | Hi        |
+---------+---------+-----------+
```

Changing the order in which the names are evaluated changes the result because null values are now encountered in a different order.

```
select country, name,
  first_value(greeting ignore nulls)
    over (partition by country order by name desc, greeting) as greeting
  from mail_merge
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Japan   | Kei     | NULL         |
| China   | Peng    | Nihao        |
| China   | Li      | Nihao        |
| Sweden  | Mats    | Tja          |
| Sweden  | Bjorn   | Tja          |
| USA     | Pete    | Hello        |
| USA     | John    | Hello        |
| Germany | Michael | Guten morgen |
| Germany | Boris   | Guten morgen |
+---------+---------+--------------+
```

Related information:

LAST_VALUE Function on page 517

LAG Function

This function returns the value of an expression using column values from a preceding row. You specify an integer offset, which designates a row position some number of rows previous to the current row. Any column references in the expression argument refer to column values from that prior row. Typically, the table contains a time sequence or numeric sequence column that clearly distinguishes the ordering of the rows.

Syntax:

```
LAG (expr [, offset] [, default])
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Sometimes used an an alternative to doing a self-join.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example uses the same stock data created in Window Clause on page 508. For each day, the query prints the closing price alongside the previous day's closing price. The first row for each stock symbol has no previous row, so that LAG( ) value is NULL.

```
select stock_symbol, closing_date, closing_price,
    lag(closing_price,1) over (partition by stock_symbol order by closing_date) as
"yesterday closing"
  from stock_ticker
    order by closing_date;
+--------------+---------------------+---------------+-------------------+
| stock_symbol | closing_date        | closing_price | yesterday closing |
+--------------+---------------------+---------------+-------------------+
| JDR          | 2014-09-13 00:00:00 | 12.86         | NULL              |
| JDR          | 2014-09-14 00:00:00 | 12.89         | 12.86             |
| JDR          | 2014-09-15 00:00:00 | 12.94         | 12.89             |
| JDR          | 2014-09-16 00:00:00 | 12.55         | 12.94             |
| JDR          | 2014-09-17 00:00:00 | 14.03         | 12.55             |
| JDR          | 2014-09-18 00:00:00 | 14.75         | 14.03             |
| JDR          | 2014-09-19 00:00:00 | 13.98         | 14.75             |
+--------------+---------------------+---------------+-------------------+
```

The following example does an arithmetic operation between the current row and a value from the previous row, to produce a delta value for each day. This example also demonstrates how ORDER BY works independently in the different parts of the query. The ORDER BY closing_date in the OVER clause makes the query analyze the rows in chronological order. Then the outer query block uses ORDER BY closing_date DESC to present the results with the most recent date first.

```
select stock_symbol, closing_date, closing_price,
    cast(
      closing_price - lag(closing_price,1) over
        (partition by stock_symbol order by closing_date)
      as decimal(8,2)
    )
    as "change from yesterday"
  from stock_ticker
    order by closing_date desc;
+--------------+---------------------+---------------+-----------------------+
| stock_symbol | closing_date        | closing_price | change from yesterday |
+--------------+---------------------+---------------+-----------------------+
| JDR          | 2014-09-19 00:00:00 | 13.98         | -0.76                 |
| JDR          | 2014-09-18 00:00:00 | 14.75         | 0.72                  |
| JDR          | 2014-09-17 00:00:00 | 14.03         | 1.47                  |
| JDR          | 2014-09-16 00:00:00 | 12.55         | -0.38                 |
| JDR          | 2014-09-15 00:00:00 | 12.94         | 0.04                  |
| JDR          | 2014-09-14 00:00:00 | 12.89         | 0.03                  |
```

```
| JDR          | 2014-09-13 00:00:00 | 12.86         | NULL                   |
+--------------+---------------------+---------------+------------------------+
```

Related information:

This function is the converse of LEAD Function on page 518.

LAST_VALUE Function

Returns the expression value from the last row in the window. If your table has null values, you can use the `IGNORE NULLS` clause to return the last non-null value from the window. This same value is repeated for all result rows for the group. The return value is `NULL` if the input expression is `NULL`.

Syntax:

```
LAST_VALUE(expr [IGNORE NULLS]) OVER([partition_by_clause] order_by_clause
[window_clause])
```

The `PARTITION BY` clause is optional. The `ORDER BY` clause is required. The window clause is optional.

Usage notes:

If any duplicate values occur in the tuples evaluated by the `ORDER BY` clause, the result of this function is not deterministic. Consider adding additional `ORDER BY` columns to ensure consistent ordering.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example uses the same `MAIL_MERGE` table as in the example for FIRST_VALUE Function on page 514. Because the default window when `ORDER BY` is used is `BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, the query requires the `UNBOUNDED FOLLOWING` to look ahead to subsequent rows and find the last value for each country.

```
select country, name,
  last_value(greeting) over (
    partition by country order by name, greeting
    rows between unbounded preceding and unbounded following
  ) as greeting
  from mail_merge
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Germany | Boris   | Guten morgen |
| Germany | Michael | Guten morgen |
| Sweden  | Bjorn   | Tja          |
| Sweden  | Mats    | Tja          |
| USA     | John    | Hello        |
| USA     | Pete    | Hello        |
+---------+---------+--------------+
```

Introducing null values into the `MAIL_MERGE` table as in the example for FIRST_VALUE Function on page 514, the result set changes when you use the `IGNORE NULLS` clause.

```
select * from mail_merge;
+---------+---------+--------------+
| name    | country | greeting     |
+---------+---------+--------------+
| Kei     | Japan   | NULL         |
| Boris   | Germany | Guten tag    |
| Li      | China   | NULL         |
| Michael | Germany | Guten morgen |
| Bjorn   | Sweden  | Hej          |
| Peng    | China   | Nihao        |
| Pete    | USA     | Hello        |
| Mats    | Sweden  | Tja          |
| John    | USA     | Hi           |
+---------+---------+--------------+
```

```
select country, name,
  last_value(greeting ignore nulls) over (
    partition by country order by name, greeting
    rows between unbounded preceding and unbounded following
  ) as greeting
  from mail_merge;
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Japan   | Kei     | NULL         |
| Germany | Boris   | Guten morgen |
| Germany | Michael | Guten morgen |
| China   | Li      | Nihao        |
| China   | Peng    | Nihao        |
| Sweden  | Bjorn   | Tja          |
| Sweden  | Mats    | Tja          |
| USA     | John    | Hello        |
| USA     | Pete    | Hello        |
+---------+---------+--------------+
```

Related information:

FIRST_VALUE Function on page 514

LEAD Function

This function returns the value of an expression using column values from a following row. You specify an integer offset, which designates a row position some number of rows after to the current row. Any column references in the expression argument refer to column values from that later row. Typically, the table contains a time sequence or numeric sequence column that clearly distinguishes the ordering of the rows.

Syntax:

```
LEAD (expr [, offset] [, default])
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Sometimes used an an alternative to doing a self-join.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example uses the same stock data created in Window Clause on page 508. The query analyzes the closing price for a stock symbol, and for each day evaluates if the closing price for the following day is higher or lower.

```
select stock_symbol, closing_date, closing_price,
  case
    (lead(closing_price,1)
      over (partition by stock_symbol order by closing_date)
        - closing_price) > 0
    when true then "higher"
    when false then "flat or lower"
  end as "trending"
from stock_ticker
  order by closing_date;
+-------------+---------------------+---------------+---------------+
| stock_symbol | closing_date        | closing_price | trending      |
+-------------+---------------------+---------------+---------------+
| JDR          | 2014-09-13 00:00:00 | 12.86         | higher        |
| JDR          | 2014-09-14 00:00:00 | 12.89         | higher        |
| JDR          | 2014-09-15 00:00:00 | 12.94         | flat or lower |
| JDR          | 2014-09-16 00:00:00 | 12.55         | higher        |
| JDR          | 2014-09-17 00:00:00 | 14.03         | higher        |
| JDR          | 2014-09-18 00:00:00 | 14.75         | flat or lower |
```

```
| JDR          | 2014-09-19 00:00:00 | 13.98         | NULL          |
+--------------+---------------------+---------------+---------------+
```

Related information:

This function is the converse of LAG Function on page 516.

MAX Function - Analytic Context

You can include an OVER clause with a call to this function to use it as an analytic function. See MAX Function on page 490 for details and examples.

MIN Function - Analytic Context

You can include an OVER clause with a call to this function to use it as an analytic function. See MIN Function on page 493 for details and examples.

NTILE Function (CDH 5.5 or higher only)

Returns the “bucket number” associated with each row, between 1 and the value of an expression. For example, creating 100 buckets puts the lowest 1% of values in the first bucket, while creating 10 buckets puts the lowest 10% of values in the first bucket. Each partition can have a different number of buckets.

Syntax:

```
NTILE (expr [, offset ...]
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

The “ntile” name is derived from the practice of dividing result sets into fourths (quartile), tenths (decile), and so on. The NTILE( ) function divides the result set based on an arbitrary percentile value.

The number of buckets must be a positive integer.

The number of items in each bucket is identical or almost so, varying by at most 1. If the number of items does not divide evenly between the buckets, the remaining N items are divided evenly among the first N buckets.

If the number of buckets N is greater than the number of input rows in the partition, then the first N buckets each contain one item, and the remaining buckets are empty.

Examples:

The following example shows divides groups of animals into 4 buckets based on their weight. The ORDER BY ... DESC clause in the OVER( ) clause means that the heaviest 25% are in the first group, and the lightest 25% are in the fourth group. (The ORDER BY in the outermost part of the query shows how you can order the final result set independently from the order in which the rows are evaluated by the OVER( ) clause.) Because there are 9 rows in the group, divided into 4 buckets, the first bucket receives the extra item.

```
create table animals (name string, kind string, kilos decimal(9,3));

insert into animals values
  ('Elephant', 'Mammal', 4000), ('Giraffe', 'Mammal', 1200), ('Mouse', 'Mammal', 0.020),

  ('Condor', 'Bird', 15), ('Horse', 'Mammal', 500), ('Owl', 'Bird', 2.5),
  ('Ostrich', 'Bird', 145), ('Polar bear', 'Mammal', 700), ('Housecat', 'Mammal', 5);

select name, ntile(4) over (order by kilos desc) as quarter
  from animals
order by quarter desc;
+------------+---------+
| name       | quarter |
+------------+---------+
| Owl        | 4       |
```

```
| Mouse      | 4       |
| Condor     | 3       |
| Housecat   | 3       |
| Horse      | 2       |
| Ostrich    | 2       |
| Elephant   | 1       |
| Giraffe    | 1       |
| Polar bear | 1       |
+------------+---------+
```

The following examples show how the PARTITION clause works for the NTILE( ) function. Here, we divide each kind of animal (mammal or bird) into 2 buckets, the heavier half and the lighter half.

```
select name, kind, ntile(2) over (partition by kind order by kilos desc) as half
  from animals
order by kind;
+------------+--------+------+
| name       | kind   | half |
+------------+--------+------+
| Ostrich    | Bird   | 1    |
| Condor     | Bird   | 1    |
| Owl        | Bird   | 2    |
| Elephant   | Mammal | 1    |
| Giraffe    | Mammal | 1    |
| Polar bear | Mammal | 1    |
| Horse      | Mammal | 2    |
| Housecat   | Mammal | 2    |
| Mouse      | Mammal | 2    |
+------------+--------+------+
```

Again, the result set can be ordered independently from the analytic evaluation. This next example lists all the animals heaviest to lightest, showing that elephant and giraffe are in the “top half” of mammals by weight, while housecat and mouse are in the “bottom half”.

```
select name, kind, ntile(2) over (partition by kind order by kilos desc) as half
  from animals
order by kilos desc;
+------------+--------+------+
| name       | kind   | half |
+------------+--------+------+
| Elephant   | Mammal | 1    |
| Giraffe    | Mammal | 1    |
| Polar bear | Mammal | 1    |
| Horse      | Mammal | 2    |
| Ostrich    | Bird   | 1    |
| Condor     | Bird   | 1    |
| Housecat   | Mammal | 2    |
| Owl        | Bird   | 2    |
| Mouse      | Mammal | 2    |
+------------+--------+------+
```

PERCENT_RANK Function (CDH 5.5 or higher only)

Syntax:

```
PERCENT_RANK (expr)
  OVER ([partition_by_clause] order_by_clause)
```

Calculates the rank, expressed as a percentage, of each row within a group of rows. If rank is the value for that same row from the RANK( ) function (from 1 to the total number of rows in the partition group), then the PERCENT_RANK( ) value is calculated as (*rank* - 1) / (*rows_in_group* - 1) . If there is only a single item in the partition group, its PERCENT_RANK( ) value is 0.

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

This function is similar to the RANK and CUME_DIST( ) functions: it returns an ascending sequence representing the position of each row within the rows of the same partition group. The actual numeric sequence is calculated differently, and the handling of duplicate (tied) values is different.

The return values range from 0 to 1 inclusive. The first row in each partition group always has the value 0. A NULL value is considered the lowest possible value. In the case of duplicate input values, all the corresponding rows in the result set have an identical value: the lowest PERCENT_RANK( ) value of those tied rows. (In contrast to CUME_DIST( ), where all tied rows have the highest CUME_DIST( ) value.)

Examples:

The following example uses the same ANIMALS table as the examples for CUME_DIST( ) and NTILE( ), with a few additional rows to illustrate the results where some values are NULL or there is only a single row in a partition group.

```
insert into animals values ('Komodo dragon', 'Reptile', 70);
insert into animals values ('Unicorn', 'Mythical', NULL);
insert into animals values ('Fire-breathing dragon', 'Mythical', NULL);
```

As with CUME_DIST( ), there is an ascending sequence for each kind of animal. For example, the “Birds” and “Mammals” rows each have a PERCENT_RANK( ) sequence that ranges from 0 to 1. The “Reptile” row has a PERCENT_RANK( ) of 0 because that partition group contains only a single item. Both “Mythical” animals have a PERCENT_RANK( ) of 0 because a NULL is considered the lowest value within its partition group.

```
select name, kind, percent_rank() over (partition by kind order by kilos) from animals;
+----------------------+----------+--------------------------+
| name                 | kind     | percent_rank() OVER(...) |
+----------------------+----------+--------------------------+
| Mouse                | Mammal   | 0                        |
| Housecat             | Mammal   | 0.2                      |
| Horse                | Mammal   | 0.4                      |
| Polar bear           | Mammal   | 0.6                      |
| Giraffe              | Mammal   | 0.8                      |
| Elephant             | Mammal   | 1                        |
| Komodo dragon        | Reptile  | 0                        |
| Owl                  | Bird     | 0                        |
| California Condor    | Bird     | 0.25                     |
| Andean Condor        | Bird     | 0.25                     |
| Condor               | Bird     | 0.25                     |
| Ostrich              | Bird     | 1                        |
| Fire-breathing dragon | Mythical | 0                        |
| Unicorn              | Mythical | 0                        |
+----------------------+----------+--------------------------+
```

RANK Function

Returns an ascending sequence of integers, starting with 1. The output sequence produces duplicate integers for duplicate values of the ORDER BY expressions. After generating duplicate output values for the “tied” input values, the function increments the sequence by the number of tied values. Therefore, the sequence contains both duplicates and gaps when the input contains duplicates. Starts the sequence over for each group produced by the PARTITIONED BY clause.

Syntax:

```
RANK() OVER([partition_by_clause] order_by_clause)
```

The PARTITION BY clause is optional. The ORDER BY clause is required. The window clause is not allowed.

Usage notes:

Often used for top-N and bottom-N queries. For example, it could produce a “top 10” report including several items that were tied for 10th place.

Similar to ROW_NUMBER and DENSE_RANK. These functions differ in how they treat duplicate combinations of values.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example demonstrates how the RANK( ) function identifies where each value “places” in the result set, producing the same result for duplicate values, and skipping values in the sequence to account for the number of duplicates. For example, when results are ordered by the X column, both 1 values are tied for first; both 2 values are tied for third; and so on.

```
select x, rank() over(order by x) as rank, property from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | square   |
| 1  | 1    | odd      |
| 2  | 3    | even     |
| 2  | 3    | prime    |
| 3  | 5    | prime    |
| 3  | 5    | odd      |
| 4  | 7    | even     |
| 4  | 7    | square   |
| 5  | 9    | odd      |
| 5  | 9    | prime    |
| 6  | 11   | even     |
| 6  | 11   | perfect  |
| 7  | 13   | lucky    |
| 7  | 13   | lucky    |
| 7  | 13   | lucky    |
| 7  | 13   | odd      |
| 7  | 13   | prime    |
| 8  | 18   | even     |
| 9  | 19   | square   |
| 9  | 19   | odd      |
| 10 | 21   | round    |
| 10 | 21   | even     |
+----+------+----------+
```

The following examples show how the RANK( ) function is affected by the PARTITION property within the ORDER BY clause.

Partitioning by the PROPERTY column groups all the even, odd, and so on values together, and RANK( ) returns the place of each value within the group, producing several ascending sequences.

```
select x, rank() over(partition by property order by x) as rank, property from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 2  | 1    | even     |
| 4  | 2    | even     |
| 6  | 3    | even     |
| 8  | 4    | even     |
| 10 | 5    | even     |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 1  | 1    | odd      |
| 3  | 2    | odd      |
| 5  | 3    | odd      |
| 7  | 4    | odd      |
| 9  | 5    | odd      |
| 6  | 1    | perfect  |
| 2  | 1    | prime    |
| 3  | 2    | prime    |
| 5  | 3    | prime    |
| 7  | 4    | prime    |
| 10 | 1    | round    |
| 1  | 1    | square   |
| 4  | 2    | square   |
| 9  | 3    | square   |
+----+------+----------+
```

Partitioning by the X column groups all the duplicate numbers together and returns the place each value within the group; because each value occurs only 1 or 2 times, RANK( ) designates each X value as either first or second within its group.

```
select x, rank() over(partition by x order by property) as rank, property from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | odd      |
| 1  | 2    | square   |
| 2  | 1    | even     |
| 2  | 2    | prime    |
| 3  | 1    | odd      |
| 3  | 2    | prime    |
| 4  | 1    | even     |
| 4  | 2    | square   |
| 5  | 1    | odd      |
| 5  | 2    | prime    |
| 6  | 1    | even     |
| 6  | 2    | perfect  |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 4    | odd      |
| 7  | 5    | prime    |
| 8  | 1    | even     |
| 9  | 1    | odd      |
| 9  | 2    | square   |
| 10 | 1    | even     |
| 10 | 2    | round    |
+----+------+----------+
```

The following example shows how a magazine might prepare a list of history's wealthiest people. Croesus and Midas are tied for second, then Crassus is fourth.

```
select rank() over (order by net_worth desc) as rank, name, net_worth from wealth order
 by rank, name;
+------+---------+---------------+
| rank | name    | net_worth     |
+------+---------+---------------+
| 1    | Solomon | 2000000000.00 |
| 2    | Croesus | 1000000000.00 |
| 2    | Midas   | 1000000000.00 |
| 4    | Crassus | 500000000.00  |
| 5    | Scrooge | 80000000.00   |
+------+---------+---------------+
```

Related information:

DENSE_RANK Function on page 512, ROW_NUMBER Function on page 523

ROW_NUMBER Function

Returns an ascending sequence of integers, starting with 1. Starts the sequence over for each group produced by the PARTITIONED BY clause. The output sequence includes different values for duplicate input values. Therefore, the sequence never contains any duplicates or gaps, regardless of duplicate input values.

Syntax:

```
ROW_NUMBER() OVER([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Often used for top-N and bottom-N queries where the input values are known to be unique, or precisely N rows are needed regardless of duplicate values.

Because its result value is different for each row in the result set (when used without a PARTITION BY clause), ROW_NUMBER( ) can be used to synthesize unique numeric ID values, for example for result sets involving unique values or tuples.

Similar to RANK and DENSE_RANK. These functions differ in how they treat duplicate combinations of values.

Added in: CDH 5.2.0 / Impala 2.0.0

Examples:

The following example demonstrates how ROW_NUMBER( ) produces a continuous numeric sequence, even though some values of X are repeated.

```
select x, row_number() over(order by x, property) as row_number, property from int_t;
+----+------------+----------+
| x  | row_number | property |
+----+------------+----------+
| 1  | 1          | odd      |
| 1  | 2          | square   |
| 2  | 3          | even     |
| 2  | 4          | prime    |
| 3  | 5          | odd      |
| 3  | 6          | prime    |
| 4  | 7          | even     |
| 4  | 8          | square   |
| 5  | 9          | odd      |
| 5  | 10         | prime    |
| 6  | 11         | even     |
| 6  | 12         | perfect  |
| 7  | 13         | lucky    |
| 7  | 14         | lucky    |
| 7  | 15         | lucky    |
| 7  | 16         | odd      |
| 7  | 17         | prime    |
| 8  | 18         | even     |
| 9  | 19         | odd      |
| 9  | 20         | square   |
| 10 | 21         | even     |
| 10 | 22         | round    |
+----+------------+----------+
```

The following example shows how a financial institution might assign customer IDs to some of history's wealthiest figures. Although two of the people have identical net worth figures, unique IDs are required for this purpose. ROW_NUMBER( ) produces a sequence of five different values for the five input rows.

```
select row_number() over (order by net_worth desc) as account_id, name, net_worth
  from wealth order by account_id, name;
+------------+---------+---------------+
| account_id | name    | net_worth     |
+------------+---------+---------------+
| 1          | Solomon | 2000000000.00 |
| 2          | Croesus | 1000000000.00 |
| 3          | Midas   | 1000000000.00 |
| 4          | Crassus | 500000000.00  |
| 5          | Scrooge | 80000000.00   |
+------------+---------+---------------+
```

Related information:

RANK Function on page 521, DENSE_RANK Function on page 512

SUM Function - Analytic Context

You can include an OVER clause with a call to this function to use it as an analytic function. See SUM Function on page 501 for details and examples.

User-Defined Functions (UDFs)

User-defined functions (frequently abbreviated as UDFs) let you code your own application logic for processing column values during an Impala query. For example, a UDF could perform calculations using an external math library, combine several column values into one, do geospatial calculations, or other kinds of tests and transformations that are outside the scope of the built-in SQL operators and functions.

You can use UDFs to simplify query logic when producing reports, or to transform data in flexible ways when copying from one table to another with the `INSERT ... SELECT` syntax.

You might be familiar with this feature from other database products, under names such as stored functions or stored routines.

Impala support for UDFs is available in Impala 1.2 and higher:

- In Impala 1.1, using UDFs in a query required using the Hive shell. (Because Impala and Hive share the same metastore database, you could switch to Hive to run just those queries requiring UDFs, then switch back to Impala.)
- Starting in Impala 1.2, Impala can run both high-performance native code UDFs written in C++, and Java-based Hive UDFs that you might already have written.
- Impala can run scalar UDFs that return a single value for each row of the result set, and user-defined aggregate functions (UDAFs) that return a value based on a set of rows. Currently, Impala does not support user-defined table functions (UDTFs) or window functions.

UDF Concepts

Depending on your use case, you might write all-new functions, reuse Java UDFs that you have already written for Hive, or port Hive Java UDF code to higher-performance native Impala UDFs in C++. You can code either scalar functions for producing results one row at a time, or more complex aggregate functions for doing analysis across. The following sections discuss these different aspects of working with UDFs.

UDFs and UDAFs

Depending on your use case, the user-defined functions (UDFs) you write might accept or produce different numbers of input and output values:

- The most general kind of user-defined function (the one typically referred to by the abbreviation UDF) takes a single input value and produces a single output value. When used in a query, it is called once for each row in the result set. For example:

```
select customer_name, is_frequent_customer(customer_id) from customers;
select obfuscate(sensitive_column) from sensitive_data;
```

- A user-defined aggregate function (UDAF) accepts a group of values and returns a single value. You use UDAFs to summarize and condense sets of rows, in the same style as the built-in `COUNT`, `MAX()`, `SUM()`, and `AVG()` functions. When called in a query that uses the `GROUP BY` clause, the function is called once for each combination of `GROUP BY` values. For example:

```
-- Evaluates multiple rows but returns a single value.
select closest_restaurant(latitude, longitude) from places;

-- Evaluates batches of rows and returns a separate value for each batch.
select most_profitable_location(store_id, sales, expenses, tax_rate, depreciation) from
 franchise_data group by year;
```

- Currently, Impala does not support other categories of user-defined functions, such as user-defined table functions (UDTFs) or window functions.

Native Impala UDFs

Impala supports UDFs written in C++, in addition to supporting existing Hive UDFs written in Java. Cloudera recommends using C++ UDFs because the compiled native code can yield higher performance, with UDF execution time often 10x faster for a C++ UDF than the equivalent Java UDF.

Using Hive UDFs with Impala

Impala can run Java-based user-defined functions (UDFs), originally written for Hive, with no changes, subject to the following conditions:

- The parameters and return value must all use scalar data types supported by Impala. For example, complex or nested types are not supported.
- Hive/Java UDFs must extend `org.apache.hadoop.hive.ql.exec.UDF` class.
- Currently, Hive UDFs that accept or return the `TIMESTAMP` type are not supported.
- Prior to CDH 5.7 / Impala 2.5 the return type must be a "Writable" type such as `Text` or `IntWritable`, rather than a Java primitive type such as `String` or `int`. Otherwise, the UDF returns `NULL`. In CDH 5.7 / Impala 2.5 and higher, this restriction is lifted, and both UDF arguments and return values can be Java primitive types.
- Hive UDAFs and UDTFs are not supported.
- Typically, a Java UDF will execute several times slower in Impala than the equivalent native UDF written in C++.
- In CDH 5.7 / Impala 2.5 and higher, you can transparently call Hive Java UDFs through Impala, or call Impala Java UDFs through Hive. This feature does not apply to built-in Hive functions. Any Impala Java UDFs created with older versions must be re-created using new `CREATE FUNCTION` syntax, without any signature for arguments or the return value.

To take full advantage of the Impala architecture and performance features, you can also write Impala-specific UDFs in C++.

For background about Java-based Hive UDFs, see the Hive documentation for UDFs. For examples or tutorials for writing such UDFs, search the web for related blog posts.

The ideal way to understand how to reuse Java-based UDFs (originally written for Hive) with Impala is to take some of the Hive built-in functions (implemented as Java UDFs) and take the applicable JAR files through the UDF deployment process for Impala, creating new UDFs with different names:

1. Take a copy of the Hive JAR file containing the Hive built-in functions. For example, the path might be like `/usr/lib/hive/lib/hive-exec-0.10.0-cdh4.2.0.jar`, with different version numbers corresponding to your specific level of CDH.
2. Use `jar tf` *`jar_file`* to see a list of the classes inside the JAR. You will see names like `org/apache/hadoop/hive/ql/udf/UDFLower.class` and `org/apache/hadoop/hive/ql/udf/UDFOPNegative.class`. Make a note of the names of the functions you want to experiment with. When you specify the entry points for the Impala `CREATE FUNCTION` statement, change the slash characters to dots and strip off the `.class` suffix, for example `org.apache.hadoop.hive.ql.udf.UDFLower` and `org.apache.hadoop.hive.ql.udf.UDFOPNegative`.
3. Copy that file to an HDFS location that Impala can read. (In the examples here, we renamed the file to `hive-builtins.jar` in HDFS for simplicity.)
4. For each Java-based UDF that you want to call through Impala, issue a `CREATE FUNCTION` statement, with a `LOCATION` clause containing the full HDFS path of the JAR file, and a `SYMBOL` clause with the fully qualified name of the class, using dots as separators and without the `.class` extension. Remember that user-defined functions are associated with a particular database, so issue a `USE` statement for the appropriate database first, or specify the SQL function name as *`db_name`*`.`*`function_name`*. Use completely new names for the SQL functions, because Impala UDFs cannot have the same name as Impala built-in functions.
5. Call the function from your queries, passing arguments of the correct type to match the function signature. These arguments could be references to columns, arithmetic or other kinds of expressions, the results of `CAST` functions to ensure correct data types, and so on.

Note:

In CDH 5.12 / Impala 2.9 and higher, you can refresh the user-defined functions (UDFs) that Impala recognizes, at the database level, by running the REFRESH FUNCTIONS statement with the database name as an argument. Java-based UDFs can be added to the metastore database through Hive CREATE FUNCTION statements, and made visible to Impala by subsequently running REFRESH FUNCTIONS. For example:

```
CREATE DATABASE shared_udfs;
USE shared_udfs;
...use CREATE FUNCTION statements in Hive to create some Java-based UDFs

   that Impala is not initially aware of...
REFRESH FUNCTIONS shared_udfs;
SELECT udf_created_by_hive(c1) FROM ...
```

Java UDF Example: Reusing lower() Function

For example, the following `impala-shell` session creates an Impala UDF `my_lower()` that reuses the Java code for the Hive `lower()`: built-in function. We cannot call it `lower()` because Impala does not allow UDFs to have the same name as built-in functions. From SQL, we call the function in a basic way (in a query with no WHERE clause), directly on a column, and on the results of a string expression:

```
[localhost:21000] > create database udfs;
[localhost:21000] > use udfs;
localhost:21000] > create function lower(string) returns string location
'/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hive.ql.udf.UDFLower';
ERROR: AnalysisException: Function cannot have the same name as a builtin: lower
[localhost:21000] > create function my_lower(string) returns string location
'/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hive.ql.udf.UDFLower';
[localhost:21000] > select my_lower('Some String NOT ALREADY LOWERCASE');
+----------------------------------------------------+
| udfs.my_lower('some string not already lowercase') |
+----------------------------------------------------+
| some string not already lowercase                  |
+----------------------------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create table t2 (s string);
[localhost:21000] > insert into t2 values ('lower'),('UPPER'),('Init cap'),('CamelCase');
Inserted 4 rows in 2.28s
[localhost:21000] > select * from t2;
+-----------+
| s         |
+-----------+
| lower     |
| UPPER     |
| Init cap  |
| CamelCase |
+-----------+
Returned 4 row(s) in 0.47s
[localhost:21000] > select my_lower(s) from t2;
+------------------+
| udfs.my_lower(s) |
+------------------+
| lower            |
| upper            |
| init cap         |
| camelcase        |
+------------------+
Returned 4 row(s) in 0.54s
[localhost:21000] > select my_lower(concat('ABC ',s,' XYZ')) from t2;
+----------------------------------------+
| udfs.my_lower(concat('abc ', s, ' xyz')) |
+----------------------------------------+
| abc lower xyz                          |
| abc upper xyz                          |
| abc init cap xyz                       |
```

```
| abc camelcase xyz                       |
+-----------------------------------------+
Returned 4 row(s) in 0.22s
```

Java UDF Example: Reusing negative() Function

Here is an example that reuses the Hive Java code for the `negative()` built-in function. This example demonstrates how the data types of the arguments must match precisely with the function signature. At first, we create an Impala SQL function that can only accept an integer argument. Impala cannot find a matching function when the query passes a floating-point argument, although we can call the integer version of the function by casting the argument. Then we overload the same function name to also accept a floating-point argument.

```
[localhost:21000] > create table t (x int);
[localhost:21000] > insert into t values (1), (2), (4), (100);
Inserted 4 rows in 1.43s
[localhost:21000] > create function my_neg(bigint) returns bigint location
'/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hive.ql.udf.UDFOPNegative';
[localhost:21000] > select my_neg(4);
+----------------+
| udfs.my_neg(4) |
+----------------+
| -4             |
+----------------+
[localhost:21000] > select my_neg(x) from t;
+----------------+
| udfs.my_neg(x) |
+----------------+
| -2             |
| -4             |
| -100           |
+----------------+
Returned 3 row(s) in 0.60s
[localhost:21000] > select my_neg(4.0);
ERROR: AnalysisException: No matching function with signature: udfs.my_neg(FLOAT).
[localhost:21000] > select my_neg(cast(4.0 as int));
+------------------------------+
| udfs.my_neg(cast(4.0 as int)) |
+------------------------------+
| -4                           |
+------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create function my_neg(double) returns double location
'/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hive.ql.udf.UDFOPNegative';
[localhost:21000] > select my_neg(4.0);
+------------------+
| udfs.my_neg(4.0) |
+------------------+
| -4               |
+------------------+
Returned 1 row(s) in 0.11s
```

You can find the sample files mentioned here in the Impala github repo.

Runtime Environment for UDFs

By default, Impala copies UDFs into `/tmp`, and you can configure this location through the `--local_library_dir` startup flag for the `impalad` daemon.

Installing the UDF Development Package

To develop UDFs for Impala, download and install the `impala-udf-devel` package (RHEL-based distributions) or `impala-udf-dev` (Ubuntu and Debian). This package contains header files, sample source, and build configuration files.

1. Start at:
   - https://archive.cloudera.com/cdh5/ for CDH 5
   - https://archive.cloudera.com/cdh6/ for CDH 6

2. Locate the appropriate `.repo` or list file for your operating system version, such as the `.repo` file for CDH 5 on RHEL 7.
3. Use the `yum`, `zypper`, or `apt-get` commands depending on your operating system. For the package name, specify `impala-udf-devel` (RHEL-based distributions) or `impala-udf-dev` (Ubuntu and Debian).



Note: The UDF development code does not rely on Impala being installed on the same machine. You can write and compile UDFs on a minimal development system, then deploy them on a different one for use with Impala. If you develop UDFs on a server managed by Cloudera Manager through the parcel mechanism, you still install the UDF development kit through the package mechanism; this small standalone package does not interfere with the parcels containing the main Impala code.

When you are ready to start writing your own UDFs, download the sample code and build scripts from the Cloudera sample UDF github. Then see Writing User-Defined Functions (UDFs) on page 529 for how to code UDFs, and Examples of Creating and Using UDFs on page 535 for how to build and run UDFs.

Writing User-Defined Functions (UDFs)

Before starting UDF development, make sure to install the development package and download the UDF code samples, as described in Installing the UDF Development Package on page 528.

When writing UDFs:

- Keep in mind the data type differences as you transfer values from the high-level SQL to your lower-level UDF code. For example, in the UDF code you might be much more aware of how many bytes different kinds of integers require.
- Use best practices for function-oriented programming: choose arguments carefully, avoid side effects, make each function do a single thing, and so on.

Getting Started with UDF Coding

To understand the layout and member variables and functions of the predefined UDF data types, examine the header file `/usr/include/impala_udf/udf.h`:

```
// This is the only Impala header required to develop UDFs and UDAs. This header
// contains the types that need to be used and the FunctionContext object. The context
// object serves as the interface object between the UDF/UDA and the impala process.
```

For the basic declarations needed to write a scalar UDF, see the header file `udf-sample.h` within the sample build environment, which defines a simple function named `AddUdf()`:

```
#ifndef IMPALA_UDF_SAMPLE_UDF_H
#define IMPALA_UDF_SAMPLE_UDF_H

#include <impala_udf/udf.h>

using namespace impala_udf;

IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const IntVal& arg2);

#endif
```

For sample C++ code for a simple function named `AddUdf()`, see the source file `udf-sample.cc` within the sample build environment:

```
#include "udf-sample.h"

// In this sample we are declaring a UDF that adds two ints and returns an int.
IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const IntVal& arg2) {
  if (arg1.is_null || arg2.is_null) return IntVal::null();
  return IntVal(arg1.val + arg2.val);
}
```

```
// Multiple UDFs can be defined in the same file
```

Data Types for Function Arguments and Return Values

Each value that a user-defined function can accept as an argument or return as a result value must map to a SQL data type that you could specify for a table column.

Currently, Impala UDFs cannot accept arguments or return values of the Impala complex types (STRUCT, ARRAY, or MAP).

Each data type has a corresponding structure defined in the C++ and Java header files, with two member fields and some predefined comparison operators and constructors:

- `is_null` indicates whether the value is NULL or not. `val` holds the actual argument or return value when it is non-NULL.
- Each struct also defines a `null()` member function that constructs an instance of the struct with the `is_null` flag set.
- The built-in SQL comparison operators and clauses such as <, >=, BETWEEN, and ORDER BY all work automatically based on the SQL return type of each UDF. For example, Impala knows how to evaluate `BETWEEN 1 AND udf_returning_int(col1)` or `ORDER BY udf_returning_string(col2)` without you declaring any comparison operators within the UDF itself.

  For convenience within your UDF code, each struct defines == and != operators for comparisons with other structs of the same type. These are for typical C++ comparisons within your own code, not necessarily reproducing SQL semantics. For example, if the `is_null` flag is set in both structs, they compare as equal. That behavior of `null` comparisons is different from SQL (where `NULL == NULL` is NULL rather than `true`), but more in line with typical C++ behavior.
- Each kind of struct has one or more constructors that define a filled-in instance of the struct, optionally with default values.
- Impala cannot process UDFs that accept the composite or nested types as arguments or return them as result values. This limitation applies both to Impala UDFs written in C++ and Java-based Hive UDFs.
- You can overload functions by creating multiple functions with the same SQL name but different argument types. For overloaded functions, you must use different C++ or Java entry point names in the underlying functions.

The data types defined on the C++ side (in `/usr/include/impala_udf/udf.h`) are:

- `IntVal` represents an INT column.
- `BigIntVal` represents a BIGINT column. Even if you do not need the full range of a BIGINT value, it can be useful to code your function arguments as `BigIntVal` to make it convenient to call the function with different kinds of integer columns and expressions as arguments. Impala automatically casts smaller integer types to larger ones when appropriate, but does not implicitly cast large integer types to smaller ones.
- `SmallIntVal` represents a SMALLINT column.
- `TinyIntVal` represents a TINYINT column.
- `StringVal` represents a STRING column. It has a `len` field representing the length of the string, and a `ptr` field pointing to the string data. It has constructors that create a new `StringVal` struct based on a null-terminated C-style string, or a pointer plus a length; these new structs still refer to the original string data rather than allocating a new buffer for the data. It also has a constructor that takes a pointer to a `FunctionContext` struct and a length, that does allocate space for a new copy of the string data, for use in UDFs that return string values.
- `BooleanVal` represents a BOOLEAN column.
- `FloatVal` represents a FLOAT column.

- `DoubleVal` represents a `DOUBLE` column.
- `TimestampVal` represents a `TIMESTAMP` column. It has a `date` field, a 32-bit integer representing the Gregorian date, that is, the days past the epoch date. It also has a `time_of_day` field, a 64-bit integer representing the current time of day in nanoseconds.

Variable-Length Argument Lists

UDFs typically take a fixed number of arguments, with each one named explicitly in the signature of your C++ function. Your function can also accept additional optional arguments, all of the same type. For example, you can concatenate two strings, three strings, four strings, and so on. Or you can compare two numbers, three numbers, four numbers, and so on.

To accept a variable-length argument list, code the signature of your function like this:

```
StringVal Concat(FunctionContext* context, const StringVal& separator,
  int num_var_args, const StringVal* args);
```

In the `CREATE FUNCTION` statement, after the type of the first optional argument, include `...` to indicate it could be followed by more arguments of the same type. For example, the following function accepts a `STRING` argument, followed by one or more additional `STRING` arguments:

```
[localhost:21000] > create function my_concat(string, string ...) returns string location
 '/user/test_user/udfs/sample.so' symbol='Concat';
```

The call from the SQL query must pass at least one argument to the variable-length portion of the argument list.

When Impala calls the function, it fills in the initial set of required arguments, then passes the number of extra arguments and a pointer to the first of those optional arguments.

Handling NULL Values

For correctness, performance, and reliability, it is important for each UDF to handle all situations where any `NULL` values are passed to your function. For example, when passed a `NULL`, UDFs typically also return `NULL`. In an aggregate function, which could be passed a combination of real and `NULL` values, you might make the final value into a `NULL` (as in `CONCAT()`), ignore the `NULL` value (as in `AVG()`), or treat it the same as a numeric zero or empty string.

Each parameter type, such as `IntVal` or `StringVal`, has an `is_null` Boolean member. Test this flag immediately for each argument to your function, and if it is set, do not refer to the `val` field of the argument structure. The `val` field is undefined when the argument is `NULL`, so your function could go into an infinite loop or produce incorrect results if you skip the special handling for `NULL`.

If your function returns `NULL` when passed a `NULL` value, or in other cases such as when a search string is not found, you can construct a null instance of the return type by using its `null()` member function.

Memory Allocation for UDFs

By default, memory allocated within a UDF is deallocated when the function exits, which could be before the query is finished. The input arguments remain allocated for the lifetime of the function, so you can refer to them in the expressions for your return values. If you use temporary variables to construct all-new string values, use the `StringVal()` constructor that takes an initial `FunctionContext*` argument followed by a length, and copy the data into the newly allocated memory buffer.

Thread-Safe Work Area for UDFs

One way to improve performance of UDFs is to specify the optional `PREPARE_FN` and `CLOSE_FN` clauses on the `CREATE FUNCTION` statement. The “prepare” function sets up a thread-safe data structure in memory that you can use as a work area. The “close” function deallocates that memory. Each subsequent call to the UDF within the same thread can access that same memory area. There might be several such memory areas allocated on the same host, as UDFs are parallelized using multiple threads.

Within this work area, you can set up predefined lookup tables, or record the results of complex operations on data types such as STRING or TIMESTAMP. Saving the results of previous computations rather than repeating the computation each time is an optimization known as http://en.wikipedia.org/wiki/Memoization. For example, if your UDF performs a regular expression match or date manipulation on a column that repeats the same value over and over, you could store the last-computed value or a hash table of already-computed values, and do a fast lookup to find the result for subsequent iterations of the UDF.

Each such function must have the signature:

```
void function_name(impala_udf::FunctionContext*,
impala_udf::FunctionContext::FunctionScope)
```

Currently, only THREAD_SCOPE is implemented, not FRAGMENT_SCOPE. See udf.h for details about the scope values.

Error Handling for UDFs

To handle errors in UDFs, you call functions that are members of the initial FunctionContext* argument passed to your function.

A UDF can record one or more warnings, for conditions that indicate minor, recoverable problems that do not cause the query to stop. The signature for this function is:

```
bool AddWarning(const char* warning_msg);
```

For a serious problem that requires cancelling the query, a UDF can set an error flag that prevents the query from returning any results. The signature for this function is:

```
void SetError(const char* error_msg);
```

Writing User-Defined Aggregate Functions (UDAFs)

User-defined aggregate functions (UDAFs or UDAs) are a powerful and flexible category of user-defined functions. If a query processes N rows, calling a UDAF during the query condenses the result set, anywhere from a single value (such as with the SUM or MAX functions), or some number less than or equal to N (as in queries using the GROUP BY or HAVING clause).

The Underlying Functions for a UDA

A UDAF must maintain a state value across subsequent calls, so that it can accumulate a result across a set of calls, rather than derive it purely from one set of arguments. For that reason, a UDAF is represented by multiple underlying functions:

- An initialization function that sets any counters to zero, creates empty buffers, and does any other one-time setup for a query.
- An update function that processes the arguments for each row in the query result set and accumulates an intermediate result for each node. For example, this function might increment a counter, append to a string buffer, or set flags.
- A merge function that combines the intermediate results from two different nodes.
- A serialize function that flattens any intermediate values containing pointers, and frees any memory allocated during the init, update, and merge phases.
- A finalize function that either passes through the combined result unchanged, or does one final transformation.

In the SQL syntax, you create a UDAF by using the statement CREATE AGGREGATE FUNCTION. You specify the entry points of the underlying C++ functions using the clauses INIT_FN, UPDATE_FN, MERGE_FN, SERIALIZE_FN, and FINALIZE_FN.

For convenience, you can use a naming convention for the underlying functions and Impala automatically recognizes those entry points. Specify the UPDATE_FN clause, using an entry point name containing the string update or Update. When you omit the other _FN clauses from the SQL statement, Impala looks for entry points with names formed by substituting the update or Update portion of the specified name.

`uda-sample.h`:

See this file online at: uda-sample.h

`uda-sample.cc`:

See this file online at: uda-sample.cc

Intermediate Results for UDAs

A user-defined aggregate function might produce and combine intermediate results during some phases of processing, using a different data type than the final return value. For example, if you implement a function similar to the built-in `AVG()` function, it must keep track of two values, the number of values counted and the sum of those values. Or, you might accumulate a string value over the course of a UDA, then in the end return a numeric or Boolean result.

In such a case, specify the data type of the intermediate results using the optional `INTERMEDIATE` *type_name* clause of the `CREATE AGGREGATE FUNCTION` statement. If the intermediate data is a typeless byte array (for example, to represent a C++ struct or array), specify the type name as `CHAR(`*n*`)`, with *n* representing the number of bytes in the intermediate result buffer.

For an example of this technique, see the `trunc_sum()` aggregate function, which accumulates intermediate results of type `DOUBLE` and returns `BIGINT` at the end. View the appropriate `CREATE FUNCTION` statement and the implementation of the underlying TruncSum*() functions on Github.

Building and Deploying UDFs

This section explains the steps to compile Impala UDFs from C++ source code, and deploy the resulting libraries for use in Impala queries.

Impala UDF development package ships with a sample build environment for UDFs, that you can study, experiment with, and adapt for your own use.

To build the sample environment:

1. Install the Impala UDF development package as described in Installing the UDF Development Package on page 528
2. Run the following commands:

```
cmake .
make
```

The `cmake` configuration command reads the file `CMakeLists.txt` and generates a `Makefile` customized for your particular directory paths. Then the `make` command runs the actual build steps based on the rules in the `Makefile`.

Impala loads the shared library from an HDFS location. After building a shared library containing one or more UDFs, use `hdfs dfs` or `hadoop fs` commands to copy the binary file to an HDFS location readable by Impala.

The final step in deployment is to issue a `CREATE FUNCTION` statement in the `impala-shell` interpreter to make Impala aware of the new function. See CREATE FUNCTION Statement on page 228 for syntax details. Because each function is associated with a particular database, always issue a `USE` statement to the appropriate database before creating a function, or specify a fully qualified name, that is, `CREATE FUNCTION` *db_name*`.`*function_name*.

As you update the UDF code and redeploy updated versions of a shared library, use `DROP FUNCTION` and `CREATE FUNCTION` to let Impala pick up the latest version of the code.



Note:

In CDH 5.7 / Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new CREATE FUNCTION syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old CREATE FUNCTION syntax do not persist across restarts because they are held in the memory of the catalogd daemon. Until you re-create such Java UDFs using the new CREATE FUNCTION syntax, you must reload those Java-based UDFs by running the original CREATE FUNCTION statements again each time you restart the catalogd daemon. Prior to CDH 5.7 / Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

See CREATE FUNCTION Statement on page 228 and DROP FUNCTION Statement on page 263 for the new syntax for the persistent Java UDFs.

Prerequisites for the build environment are:

1. Install the packages using the appropriate package installation command for your Linux distribution.

```
sudo yum install gcc-c++ cmake boost-devel
sudo yum install impala-udf-devel
# The package name on Ubuntu and Debian is impala-udf-dev.
```

2. Download the UDF sample code:

```
git clone https://github.com/cloudera/impala-udf-samples
cd impala-udf-samples &amp;amp;&amp;amp; cmake . &amp;amp;&amp;amp; make
```

3. Unpack the sample code in udf_samples.tar.gz and use that as a template to set up your build environment.

To build the original samples:

```
# Process CMakeLists.txt and set up appropriate Makefiles.
cmake .
# Generate shared libraries from UDF and UDAF sample code,
# udf_samples/libudfsample.so and udf_samples/libudasample.so
make
```

The sample code to examine, experiment with, and adapt is in these files:

- udf-sample.h: Header file that declares the signature for a scalar UDF (AddUDF).
- udf-sample.cc: Sample source for a simple UDF that adds two integers. Because Impala can reference multiple function entry points from the same shared library, you could add other UDF functions in this file and add their signatures to the corresponding header file.
- udf-sample-test.cc: Basic unit tests for the sample UDF.
- uda-sample.h: Header file that declares the signature for sample aggregate functions. The SQL functions will be called COUNT, AVG, and STRINGCONCAT. Because aggregate functions require more elaborate coding to handle the processing for multiple phases, there are several underlying C++ functions such as CountInit, AvgUpdate, and StringConcatFinalize.
- uda-sample.cc: Sample source for simple UDAFs that demonstrate how to manage the state transitions as the underlying functions are called during the different phases of query processing.
  - The UDAF that imitates the COUNT function keeps track of a single incrementing number; the merge functions combine the intermediate count values from each Impala node, and the combined number is returned verbatim by the finalize function.
  - The UDAF that imitates the AVG function keeps track of two numbers, a count of rows processed and the sum of values for a column. These numbers are updated and merged as with COUNT, then the finalize function divides them to produce and return the final average value.

  - The UDAF that concatenates string values into a comma-separated list demonstrates how to manage storage for a string that increases in length as the function is called for multiple rows.

- `uda-sample-test.cc`: basic unit tests for the sample UDAFs.

Performance Considerations for UDFs

Because a UDF typically processes each row of a table, potentially being called billions of times, the performance of each UDF is a critical factor in the speed of the overall ETL or ELT pipeline. Tiny optimizations you can make within the function body can pay off in a big way when the function is called over and over when processing a huge result set.

Examples of Creating and Using UDFs

This section demonstrates how to create and use all kinds of user-defined functions (UDFs).

For downloadable examples that you can experiment with, adapt, and use as templates for your own functions, see the Cloudera sample UDF github. You must have already installed the appropriate header files, as explained in Installing the UDF Development Package on page 528.

Sample C++ UDFs: HasVowels, CountVowels, StripVowels

This example shows 3 separate UDFs that operate on strings and return different data types. In the C++ code, the functions are `HasVowels()` (checks if a string contains any vowels), `CountVowels()` (returns the number of vowels in a string), and `StripVowels()` (returns a new string with vowels removed).

First, we add the signatures for these functions to `udf-sample.h` in the demo build environment:

```
BooleanVal HasVowels(FunctionContext* context, const StringVal& input);
IntVal CountVowels(FunctionContext* context, const StringVal& arg1);
StringVal StripVowels(FunctionContext* context, const StringVal& arg1);
```

Then, we add the bodies of these functions to `udf-sample.cc`:

```
BooleanVal HasVowels(FunctionContext* context, const StringVal& input)
{
        if (input.is_null) return BooleanVal::null();

        int index;
        uint8_t *ptr;

        for (ptr = input.ptr, index = 0; index <= input.len; index++, ptr++)
        {
                uint8_t c = tolower(*ptr);
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
                {
                        return BooleanVal(true);
                }
        }
        return BooleanVal(false);
}

IntVal CountVowels(FunctionContext* context, const StringVal& arg1)
{
        if (arg1.is_null) return IntVal::null();

        int count;
        int index;
        uint8_t *ptr;

        for (ptr = arg1.ptr, count = 0, index = 0; index <= arg1.len; index++, ptr++)
        {
                uint8_t c = tolower(*ptr);
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
                {
                        count++;
                }
        }
        return IntVal(count);
```

```
}

StringVal StripVowels(FunctionContext* context, const StringVal& arg1)
{
        if (arg1.is_null) return StringVal::null();

        int index;
        std::string original((const char *)arg1.ptr,arg1.len);
        std::string shorter("");

        for (index = 0; index < original.length(); index++)
        {
                uint8_t c = original[index];
                uint8_t l = tolower(c);

                if (l == 'a' || l == 'e' || l == 'i' || l == 'o' || l == 'u')
                {
                        ;
                }
                else
                {
                    shorter.append(1, (char)c);
                }
        }
// The modified string is stored in 'shorter', which is destroyed when this function
ends. We need to make a string val
// and copy the contents.
       StringVal result(context, shorter.size()); // Only the version of the ctor that
 takes a context object allocates new memory
        memcpy(result.ptr, shorter.c_str(), shorter.size());
        return result;
}
```

We build a shared library, `libudfsample.so`, and put the library file into HDFS where Impala can read it:

```
$ make
[  0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
[ 33%] Built target udasample
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
Scanning dependencies of target udfsample
[ 83%] Building CXX object CMakeFiles/udfsample.dir/udf-sample.o
Linking CXX shared library udf_samples/libudfsample.so
[ 83%] Built target udfsample
Linking CXX executable udf_samples/udf-sample-test
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudfsample.so /user/hive/udfs/libudfsample.so
```

Finally, we go into the `impala-shell` interpreter where we set up some sample data, issue `CREATE FUNCTION` statements to set up the SQL function names, and call the functions in some queries:

```
[localhost:21000] > create database udf_testing;
[localhost:21000] > use udf_testing;

[localhost:21000] > create function has_vowels (string) returns boolean location
'/user/hive/udfs/libudfsample.so' symbol='HasVowels';
[localhost:21000] > select has_vowels('abc');
+------------------------+
| udfs.has_vowels('abc') |
+------------------------+
| true                   |
+------------------------+
Returned 1 row(s) in 0.13s
[localhost:21000] > select has_vowels('zxcvbnm');
+---------------------------+
| udfs.has_vowels('zxcvbnm') |
+---------------------------+
| false                      |
+---------------------------+
```

```
Returned 1 row(s) in 0.12s
[localhost:21000] > select has_vowels(null);
+-----------------------+
| udfs.has_vowels(null) |
+-----------------------+
| NULL                  |
+-----------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > select s, has_vowels(s) from t2;
+-----------+--------------------+
| s         | udfs.has_vowels(s) |
+-----------+--------------------+
| lower     | true               |
| UPPER     | true               |
| Init cap  | true               |
| CamelCase | true               |
+-----------+--------------------+
Returned 4 row(s) in 0.24s

[localhost:21000] > create function count_vowels (string) returns int location
'/user/hive/udfs/libudfsample.so' symbol='CountVowels';
[localhost:21000] > select count_vowels('cat in the hat');
+-------------------------------------+
| udfs.count_vowels('cat in the hat') |
+-------------------------------------+
| 4                                   |
+-------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select s, count_vowels(s) from t2;
+-----------+----------------------+
| s         | udfs.count_vowels(s) |
+-----------+----------------------+
| lower     | 2                    |
| UPPER     | 2                    |
| Init cap  | 3                    |
| CamelCase | 4                    |
+-----------+----------------------+
Returned 4 row(s) in 0.23s
[localhost:21000] > select count_vowels(null);
+------------------------+
| udfs.count_vowels(null) |
+------------------------+
| NULL                   |
+------------------------+
Returned 1 row(s) in 0.12s

[localhost:21000] > create function strip_vowels (string) returns string location
'/user/hive/udfs/libudfsample.so' symbol='StripVowels';
[localhost:21000] > select strip_vowels('abcdefg');
+------------------------------+
| udfs.strip_vowels('abcdefg') |
+------------------------------+
| bcdfg                        |
+------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > select strip_vowels('ABCDEFG');
+------------------------------+
| udfs.strip_vowels('abcdefg') |
+------------------------------+
| BCDFG                        |
+------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select strip_vowels(null);
+------------------------+
| udfs.strip_vowels(null) |
+------------------------+
| NULL                   |
+------------------------+
Returned 1 row(s) in 0.16s
[localhost:21000] > select s, strip_vowels(s) from t2;
+-----------+----------------------+
| s         | udfs.strip_vowels(s) |
+-----------+----------------------+
```

```
| lower     | lwr                  |
| UPPER     | PPR                  |
| Init cap  | nt cp                |
| CamelCase | CmlCs                |
+-----------+----------------------+
Returned 4 row(s) in 0.24s
```

Sample C++ UDA: SumOfSquares

This example demonstrates a user-defined aggregate function (UDA) that produces the sum of the squares of its input values.

The coding for a UDA is a little more involved than a scalar UDF, because the processing is split into several phases, each implemented by a different function. Each phase is relatively straightforward: the “update” and “merge” phases, where most of the work is done, read an input value and combine it with some accumulated intermediate value.

As in our sample UDF from the previous example, we add function signatures to a header file (in this case, `uda-sample.h`). Because this is a math-oriented UDA, we make two versions of each function, one accepting an integer value and the other accepting a floating-point value.

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val);
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val);

void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal& input, BigIntVal*
val);
void SumOfSquaresUpdate(FunctionContext* context, const DoubleVal& input, DoubleVal*
val);

void SumOfSquaresMerge(FunctionContext* context, const BigIntVal& src, BigIntVal* dst);
void SumOfSquaresMerge(FunctionContext* context, const DoubleVal& src, DoubleVal* dst);

BigIntVal SumOfSquaresFinalize(FunctionContext* context, const BigIntVal& val);
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const DoubleVal& val);
```

We add the function bodies to a C++ source file (in this case, `uda-sample.cc`):

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val) {
  val->is_null = false;
  val->val = 0;
}
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val) {
  val->is_null = false;
  val->val = 0.0;
}

void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal& input, BigIntVal*
val) {
  if (input.is_null) return;
  val->val += input.val * input.val;
}
void SumOfSquaresUpdate(FunctionContext* context, const DoubleVal& input, DoubleVal*
val) {
  if (input.is_null) return;
  val->val += input.val * input.val;
}

void SumOfSquaresMerge(FunctionContext* context, const BigIntVal& src, BigIntVal* dst)
 {
  dst->val += src.val;
}
void SumOfSquaresMerge(FunctionContext* context, const DoubleVal& src, DoubleVal* dst)
 {
  dst->val += src.val;
}

BigIntVal SumOfSquaresFinalize(FunctionContext* context, const BigIntVal& val) {
  return val;
}
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const DoubleVal& val) {
```

```
  return val;
}
```

As with the sample UDF, we build a shared library and put it into HDFS:

```
$ make
[  0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
Scanning dependencies of target udasample
[ 33%] Building CXX object CMakeFiles/udasample.dir/uda-sample.o
Linking CXX shared library udf_samples/libudasample.so
[ 33%] Built target udasample
Scanning dependencies of target uda-sample-test
[ 50%] Building CXX object CMakeFiles/uda-sample-test.dir/uda-sample-test.o
Linking CXX executable udf_samples/uda-sample-test
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
[ 83%] Built target udfsample
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudasample.so /user/hive/udfs/libudasample.so
```

To create the SQL function, we issue a `CREATE AGGREGATE FUNCTION` statement and specify the underlying C++ function names for the different phases:

```
[localhost:21000] > use udf_testing;

[localhost:21000] > create table sos (x bigint, y double);
[localhost:21000] > insert into sos values (1, 1.1), (2, 2.2), (3, 3.3), (4, 4.4);
Inserted 4 rows in 1.10s

[localhost:21000] > create aggregate function sum_of_squares(bigint) returns bigint
  > location '/user/hive/udfs/libudasample.so'
  > init_fn='SumOfSquaresInit'
  > update_fn='SumOfSquaresUpdate'
  > merge_fn='SumOfSquaresMerge'
  > finalize_fn='SumOfSquaresFinalize';

[localhost:21000] > -- Compute the same value using literals or the UDA;
[localhost:21000] > select 1*1 + 2*2 + 3*3 + 4*4;
+-------------------------------+
| 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 |
+-------------------------------+
| 30                            |
+-------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select sum_of_squares(x) from sos;
+------------------------+
| udfs.sum_of_squares(x) |
+------------------------+
| 30                     |
+------------------------+
Returned 1 row(s) in 0.35s
```

Until we create the overloaded version of the UDA, it can only handle a single data type. To allow it to handle `DOUBLE` as well as `BIGINT`, we issue another `CREATE AGGREGATE FUNCTION` statement:

```
[localhost:21000] > select sum_of_squares(y) from sos;
ERROR: AnalysisException: No matching function with signature:
udfs.sum_of_squares(DOUBLE).

[localhost:21000] > create aggregate function sum_of_squares(double) returns double
  > location '/user/hive/udfs/libudasample.so'
  > init_fn='SumOfSquaresInit'
  > update_fn='SumOfSquaresUpdate'
  > merge_fn='SumOfSquaresMerge'
  > finalize_fn='SumOfSquaresFinalize';

[localhost:21000] > -- Compute the same value using literals or the UDA;
```

```
[localhost:21000] > select 1.1*1.1 + 2.2*2.2 + 3.3*3.3 + 4.4*4.4;
+-------------------------------------------+
| 1.1 * 1.1 + 2.2 * 2.2 + 3.3 * 3.3 + 4.4 * 4.4 |
+-------------------------------------------+
| 36.3                                      |
+-------------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select sum_of_squares(y) from sos;
+------------------------+
| udfs.sum_of_squares(y) |
+------------------------+
| 36.3                   |
+------------------------+
Returned 1 row(s) in 0.35s
```

Typically, you use a UDA in queries with `GROUP BY` clauses, to produce a result set with a separate aggregate value for each combination of values from the `GROUP BY` clause. Let's change our sample table to use `0` to indicate rows containing even values, and `1` to flag rows containing odd values. Then the `GROUP BY` query can return two values, the sum of the squares for the even values, and the sum of the squares for the odd values:

```
[localhost:21000] > insert overwrite sos values (1, 1), (2, 0), (3, 1), (4, 0);
Inserted 4 rows in 1.24s

[localhost:21000] > -- Compute 1 squared + 3 squared, and 2 squared + 4 squared;
[localhost:21000] > select y, sum_of_squares(x) from sos group by y;
+---+------------------------+
| y | udfs.sum_of_squares(x) |
+---+------------------------+
| 1 | 10                     |
| 0 | 20                     |
+---+------------------------+
Returned 2 row(s) in 0.43s
```

Security Considerations for User-Defined Functions

When the Impala authorization feature is enabled:

- To call a UDF in a query, you must have the required read privilege for any databases and tables used in the query.
- The `CREATE FUNCTION` statement requires:
  - The `CREATE` privilege on the database.
  - The `ALL` privilege on two URIs where the URIs are:
    - The JAR file on the file system. For example:

```
GRANT ALL ON URI 'file:///path_to_my.jar' TO ROLE my_role;
```

    - The JAR on HDFS. For example:

```
GRANT ALL ON URI 'hdfs:///path/to/jar' TO ROLE my_role
```

See Enabling Sentry Authorization for Impala on page 87 for details about authorization in Impala.

Limitations and Restrictions for Impala UDFs

The following limitations and restrictions apply to Impala UDFs in the current release:

- Impala does not support Hive UDFs that accept or return composite or nested types, or other types not available in Impala tables.
- The Hive `current_user()` function cannot be called from a Java UDF through Impala.

- All Impala UDFs must be deterministic, that is, produce the same output each time when passed the same argument values. For example, an Impala UDF must not call functions such as `rand()` to produce different values for each invocation. It must not retrieve data from external sources, such as from disk or over the network.
- An Impala UDF must not spawn other threads or processes.
- Prior to CDH 5.7 / Impala 2.5 when the `catalogd` process is restarted, all UDFs become undefined and must be reloaded. In CDH 5.7 / Impala 2.5 and higher, this limitation only applies to older Java UDFs. Re-create those UDFs using the new `CREATE FUNCTION` syntax for Java UDFs, which excludes the function signature, to remove the limitation entirely.
- Impala currently does not support user-defined table functions (UDTFs).
- The `CHAR` and `VARCHAR` types cannot be used as input arguments or return values for UDFs.

Converting Legacy UDFs During Upgrade to CDH 5.12 or Higher

In CDH 5.7 / Impala 2.5 and higher, the CREATE FUNCTION Statement on page 228 is available for creating Java-based UDFs. UDFs created with the new syntax persist across Impala restarts, and are more compatible with Hive UDFs. Because the replication features in CDH 5.12 and higher only work with the new-style syntax, convert any older Java UDFs to use the new syntax at the same time you upgrade to CDH 5.12 or higher.

Follow these steps to convert old-style Java UDFs to the new persistent kind:

- Use `SHOW FUNCTIONS` to identify all UDFs and UDAs.
- For each function, use `SHOW CREATE FUNCTION` and save the statement in a script file.
- For Java UDFs, change the output of `SHOW CREATE FUNCTION` to use the new `CREATE FUNCTION` syntax (without argument types), which makes the UDF persistent.
- For each function, drop it and re-create it, using the new `CREATE FUNCTION` syntax for all Java UDFs.

SQL Differences Between Impala and Hive

Impala's SQL syntax follows the SQL-92 standard, and includes many industry extensions in areas such as built-in functions. See Porting SQL from Other Database Systems to Impala on page 543 for a general discussion of adapting SQL code from a variety of database systems to Impala.

Because Impala and Hive share the same metastore database and their tables are often used interchangeably, the following section covers differences between Impala and Hive in detail.

HiveQL Features not Available in Impala

The current release of Impala does not support the following SQL features that you might be familiar with from HiveQL:

- Extensibility mechanisms such as `TRANSFORM`, custom file formats, or custom SerDes.
- The `DATE` data type.
- The `BINARY` data type.
- XML functions.
- Certain aggregate functions from HiveQL: `covar_pop`, `covar_samp`, `corr`, `percentile`, `percentile_approx`, `histogram_numeric`, `collect_set`; Impala supports the set of aggregate functions listed in Impala Aggregate Functions on page 479 and analytic functions listed in Impala Analytic Functions on page 506.
- Sampling.
- Lateral views. In CDH 5.5 / Impala 2.3 and higher, Impala supports queries on complex types (`STRUCT`, `ARRAY`, or `MAP`), using join notation rather than the `EXPLODE()` keyword. See Complex Types (CDH 5.5 or higher only) on page 139 for details about Impala support for complex types.

User-defined functions (UDFs) are supported starting in Impala 1.2. See User-Defined Functions (UDFs) on page 525 for full details on Impala UDFs.

- Impala supports high-performance UDFs written in C++, as well as reusing some Java-based Hive UDFs.
- Impala supports scalar UDFs and user-defined aggregate functions (UDAFs). Impala does not currently support user-defined table generating functions (UDTFs).

- Only Impala-supported column types are supported in Java-based UDFs.
- The Hive `current_user()` function cannot be called from a Java UDF through Impala.

Impala does not currently support these HiveQL statements:

- `ANALYZE TABLE` (the Impala equivalent is `COMPUTE STATS`)
- `DESCRIBE COLUMN`
- `DESCRIBE DATABASE`
- `EXPORT TABLE`
- `IMPORT TABLE`
- `SHOW TABLE EXTENDED`
- `SHOW TBLPROPERTIES`
- `SHOW INDEXES`
- `SHOW COLUMNS`
- `INSERT OVERWRITE DIRECTORY`; use `INSERT OVERWRITE` *`table_name`* or `CREATE TABLE AS SELECT` to materialize query results into the HDFS directory associated with an Impala table.

Impala respects the `serialization.null.format` table property only for TEXT tables and ignores the property for Parquet and other formats. Hive respects the `serialization.null.format` property for Parquet and other formats and converts matching values to NULL during the scan. See Data Files for Text Tables on page 638 for using the table property in Impala.

Semantic Differences Between Impala and HiveQL Features

This section covers instances where Impala and Hive have similar functionality, sometimes including the same syntax, but there are differences in the runtime semantics of those features.

Security:

Impala utilizes the Apache Sentry authorization framework, which provides fine-grained role-based access control to protect data against unauthorized access or tampering.

The Hive component included in CDH 5.1 and higher now includes Sentry-enabled `GRANT`, `REVOKE`, and `CREATE/DROP ROLE` statements. Earlier Hive releases had a privilege system with `GRANT` and `REVOKE` statements that were primarily intended to prevent accidental deletion of data, rather than a security mechanism to protect against malicious users.

Impala can make use of privileges set up through Hive `GRANT` and `REVOKE` statements. Impala has its own `GRANT` and `REVOKE` statements in Impala 2.0 and higher. See Enabling Sentry Authorization for Impala on page 87 for the details of authorization in Impala, including how to switch from the original policy file-based privilege model to the Sentry service using privileges stored in the metastore database.

SQL statements and clauses:

The semantics of Impala SQL statements varies from HiveQL in some cases where they use similar SQL statement and clause names:

- Impala uses different syntax and names for query hints, `[SHUFFLE]` and `[NOSHUFFLE]` rather than `MapJoin` or `StreamJoin`. See Joins in Impala SELECT Statements on page 296 for the Impala details.
- Impala does not expose MapReduce specific features of `SORT BY`, `DISTRIBUTE BY`, or `CLUSTER BY`.
- Impala does not require queries to include a `FROM` clause.

Data types:

- Impala supports a limited set of implicit casts. This can help avoid undesired results from unexpected casting behavior.
  - Impala does not implicitly cast between string and numeric or Boolean types. Always use `CAST()` for these conversions.

  - Impala does perform implicit casts among the numeric types, when going from a smaller or less precise type to a larger or more precise one. For example, Impala will implicitly convert a SMALLINT to a BIGINT or FLOAT, but to convert from DOUBLE to FLOAT or INT to TINYINT requires a call to CAST() in the query.
  - Impala does perform implicit casts from string to timestamp. Impala has a restricted set of literal formats for the TIMESTAMP data type and the from_unixtime() format string; see TIMESTAMP Data Type on page 130 for details.

  See the topics under Data Types on page 101 for full details on implicit and explicit casting for each data type, and Impala Type Conversion Functions on page 423 for details about the CAST() function.

- Impala does not store or interpret timestamps using the local timezone, to avoid undesired results from unexpected time zone issues. Timestamps are stored and interpreted relative to UTC. This difference can produce different results for some calls to similarly named date/time functions between Impala and Hive. See Impala Date and Time Functions on page 424 for details about the Impala functions. See TIMESTAMP Data Type on page 130 for a discussion of how Impala handles time zones, and configuration options you can use to make Impala match the Hive behavior more closely when dealing with Parquet-encoded TIMESTAMP data or when converting between the local time zone and UTC.
- The Impala TIMESTAMP type can represent dates ranging from 1400-01-01 to 9999-12-31. This is different from the Hive date range, which is 0000-01-01 to 9999-12-31.
- Impala does not return column overflows as NULL, so that customers can distinguish between NULL data and overflow conditions similar to how they do so with traditional database systems. Impala returns the largest or smallest value in the range for the type. For example, valid values for a tinyint range from -128 to 127. In Impala, a tinyint with a value of -200 returns -128 rather than NULL. A tinyint with a value of 200 returns 127.

Miscellaneous features:

- Impala does not provide virtual columns.
- Impala does not expose locking.
- Impala does not expose some configuration properties.

Porting SQL from Other Database Systems to Impala

Although Impala uses standard SQL for queries, you might need to modify SQL source when bringing applications to Impala, due to variations in data types, built-in functions, vendor language extensions, and Hadoop-specific syntax. Even when SQL is working correctly, you might make further minor modifications for best performance.

Porting DDL and DML Statements

When adapting SQL code from a traditional database system to Impala, expect to find a number of differences in the DDL statements that you use to set up the schema. Clauses related to physical layout of files, tablespaces, and indexes have no equivalent in Impala. You might restructure your schema considerably to account for the Impala partitioning scheme and Hadoop file formats.

Expect SQL queries to have a much higher degree of compatibility. With modest rewriting to address vendor extensions and features not yet supported in Impala, you might be able to run identical or almost-identical query text on both systems.

Therefore, consider separating out the DDL into a separate Impala-specific setup script. Focus your reuse and ongoing tuning efforts on the code for SQL queries.

Porting Data Types from Other Database Systems

- Change any VARCHAR, VARCHAR2, and CHAR columns to STRING. Remove any length constraints from the column declarations; for example, change VARCHAR(32) or CHAR(1) to STRING. Impala is very flexible about the length of string values; it does not impose any length constraints or do any special processing (such as blank-padding) for STRING columns. (In Impala 2.0 and higher, there are data types VARCHAR and CHAR, with length constraints for both types and blank-padding for CHAR. However, for performance reasons, it is still preferable to use STRING columns where practical.)

- For national language character types such as NCHAR, NVARCHAR, or NCLOB, be aware that while Impala can store and query UTF-8 character data, currently some string manipulation operations only work correctly with ASCII data. See STRING Data Type on page 123 for details.
- Change any DATE, DATETIME, or TIME columns to TIMESTAMP. Remove any precision constraints. Remove any timezone clauses, and make sure your application logic or ETL process accounts for the fact that Impala expects all TIMESTAMP values to be in Coordinated Universal Time (UTC). See TIMESTAMP Data Type on page 130 for information about the TIMESTAMP data type, and Impala Date and Time Functions on page 424 for conversion functions for different date and time formats.

  You might also need to adapt date- and time-related literal values and format strings to use the supported Impala date and time formats. If you have date and time literals with different separators or different numbers of YY, MM, and so on placeholders than Impala expects, consider using calls to `regexp_replace()` to transform those values to the Impala-compatible format. See TIMESTAMP Data Type on page 130 for information about the allowed formats for date and time literals, and Impala String Functions on page 462 for string conversion functions such as `regexp_replace()`.

  Instead of SYSDATE, call the function `NOW()`.

  Instead of adding or subtracting directly from a date value to produce a value *N* days in the past or future, use an INTERVAL expression, for example `NOW() + INTERVAL 30 DAYS`.
- Although Impala supports INTERVAL expressions for datetime arithmetic, as shown in TIMESTAMP Data Type on page 130, INTERVAL is not available as a column data type in Impala. For any INTERVAL values stored in tables, convert them to numeric values that you can add or subtract using the functions in Impala Date and Time Functions on page 424. For example, if you had a table DEADLINES with an INT column TIME_PERIOD, you could construct dates N days in the future like so:

```
SELECT NOW() + INTERVAL time_period DAYS from deadlines;
```

- For YEAR columns, change to the smallest Impala integer type that has sufficient range. See Data Types on page 101 for details about ranges, casting, and so on for the various numeric data types.
- Change any DECIMAL and NUMBER types. If fixed-point precision is not required, you can use FLOAT or DOUBLE on the Impala side depending on the range of values. For applications that require precise decimal values, such as financial data, you might need to make more extensive changes to table structure and application logic, such as using separate integer columns for dollars and cents, or encoding numbers as string values and writing UDFs to manipulate them. See Data Types on page 101 for details about ranges, casting, and so on for the various numeric data types.
- FLOAT, DOUBLE, and REAL types are supported in Impala. Remove any precision and scale specifications. (In Impala, REAL is just an alias for DOUBLE; columns declared as REAL are turned into DOUBLE behind the scenes.) See Data Types on page 101 for details about ranges, casting, and so on for the various numeric data types.
- Most integer types from other systems have equivalents in Impala, perhaps under different names such as BIGINT instead of INT8. For any that are unavailable, for example MEDIUMINT, switch to the smallest Impala integer type that has sufficient range. Remove any precision specifications. See Data Types on page 101 for details about ranges, casting, and so on for the various numeric data types.
- Remove any UNSIGNED constraints. All Impala numeric types are signed. See Data Types on page 101 for details about ranges, casting, and so on for the various numeric data types.
- For any types holding bitwise values, use an integer type with enough range to hold all the relevant bits within a positive integer. See Data Types on page 101 for details about ranges, casting, and so on for the various numeric data types.

  For example, TINYINT has a maximum positive value of 127, not 256, so to manipulate 8-bit bitfields as positive numbers switch to the next largest type SMALLINT.

```
[localhost:21000] > select cast(127*2 as tinyint);
+-------------------------+
```

```
| cast(127 * 2 as tinyint) |
+--------------------------+
| -2                       |
+--------------------------+
[localhost:21000] > select cast(128 as tinyint);
+----------------------+
| cast(128 as tinyint) |
+----------------------+
| -128                 |
+----------------------+
[localhost:21000] > select cast(127*2 as smallint);
+---------------------------+
| cast(127 * 2 as smallint) |
+---------------------------+
| 254                       |
+---------------------------+
```

Impala does not support notation such as `b'0101'` for bit literals.

- For BLOB values, use STRING to represent CLOB or TEXT types (character based large objects) up to 32 KB in size. Binary large objects such as BLOB, RAW BINARY, and VARBINARY do not currently have an equivalent in Impala.
- For Boolean-like types such as BOOL, use the Impala BOOLEAN type.
- Because Impala currently does not support composite or nested types, any spatial data types in other database systems do not have direct equivalents in Impala. You could represent spatial values in string format and write UDFs to process them. See User-Defined Functions (UDFs) on page 525 for details. Where practical, separate spatial types into separate tables so that Impala can still work with the non-spatial data.
- Take out any DEFAULT clauses. Impala can use data files produced from many different sources, such as Pig, Hive, or MapReduce jobs. The fast import mechanisms of LOAD DATA and external tables mean that Impala is flexible about the format of data files, and Impala does not necessarily validate or cleanse data before querying it. When copying data through Impala INSERT statements, you can use conditional functions such as CASE or NVL to substitute some other value for NULL fields; see Impala Conditional Functions on page 457 for details.
- Take out any constraints from your CREATE TABLE and ALTER TABLE statements, for example PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, UNSIGNED, or CHECK constraints. Impala can use data files produced from many different sources, such as Pig, Hive, or MapReduce jobs. Therefore, Impala expects initial data validation to happen earlier during the ETL or ELT cycle. After data is loaded into Impala tables, you can perform queries to test for NULL values. When copying data through Impala INSERT statements, you can use conditional functions such as CASE or NVL to substitute some other value for NULL fields; see Impala Conditional Functions on page 457 for details.

  Do as much verification as practical before loading data into Impala. After data is loaded into Impala, you can do further verification using SQL queries to check if values have expected ranges, if values are NULL or not, and so on. If there is a problem with the data, you will need to re-run earlier stages of the ETL process, or do an INSERT ... SELECT statement in Impala to copy the faulty data to a new table and transform or filter out the bad values.
- Take out any CREATE INDEX, DROP INDEX, and ALTER INDEX statements, and equivalent ALTER TABLE statements. Remove any INDEX, KEY, or PRIMARY KEY clauses from CREATE TABLE and ALTER TABLE statements. Impala is optimized for bulk read operations for data warehouse-style queries, and therefore does not support indexes for its tables.
- Calls to built-in functions with out-of-range or otherwise incorrect arguments, return NULL in Impala as opposed to raising exceptions. (This rule applies even when the ABORT_ON_ERROR=true query option is in effect.) Run small-scale queries using representative data to doublecheck that calls to built-in functions are returning expected values rather than NULL. For example, unsupported CAST operations do not raise an error in Impala:

```
select cast('foo' as int);
+--------------------+
| cast('foo' as int) |
+--------------------+
| NULL               |
+--------------------+
```

- For any other type not supported in Impala, you could represent their values in string format and write UDFs to process them. See User-Defined Functions (UDFs) on page 525 for details.
- To detect the presence of unsupported or unconvertable data types in data files, do initial testing with the `ABORT_ON_ERROR=true` query option in effect. This option causes queries to fail immediately if they encounter disallowed type conversions. See ABORT_ON_ERROR Query Option on page 323 for details. For example:

```
set abort_on_error=true;
select count(*) from (select * from t1);
-- The above query will fail if the data files for T1 contain any
-- values that can't be converted to the expected Impala data types.
-- For example, if T1.C1 is defined as INT but the column contains
-- floating-point values like 1.1, the query will return an error.
```

SQL Statements to Remove or Adapt

The following SQL statements or clauses are not currently supported or supported with limitations in Impala:

- Impala supports the `DELETE` statement only for Kudu tables.

  Impala is intended for data warehouse-style operations where you do bulk moves and transforms of large quantities of data. When not using Kudu tables, instead of `DELETE`, use `INSERT OVERWRITE` to entirely replace the contents of a table or partition, or use `INSERT ... SELECT` to copy a subset of data (everything but the rows you intended to delete) from one table to another. See DML Statements on page 204 for an overview of Impala DML statements.

- Impala supports the `UPDATE` statement only for Kudu tables.

  When not using Kudu tables, instead of `UPDATE`, do all necessary transformations early in the ETL process, such as in the job that generates the original data, or when copying from one table to another to convert to a particular file format or partitioning scheme. See DML Statements on page 204 for an overview of Impala DML statements.

- Impala has no transactional statements, such as `COMMIT` or `ROLLBACK`.

  Impala effectively works like the `AUTOCOMMIT` mode in some database systems, where changes take effect as soon as they are made.

- If your database, table, column, or other names conflict with Impala reserved words, use different names or quote the names with backticks.

  See Impala Reserved Words on page 745 for the current list of Impala reserved words.

  Conversely, if you use a keyword that Impala does not recognize, it might be interpreted as a table or column alias.

  For example, in `SELECT * FROM t1 NATURAL JOIN t2`, Impala does not recognize the `NATURAL` keyword and interprets it as an alias for the table `t1`. If you experience any unexpected behavior with queries, check the list of reserved words to make sure all keywords in join and `WHERE` clauses are supported keywords in Impala.

- Impala has some restrictions on subquery support. See Subqueries in Impala SELECT Statements for the current details.

- Impala supports `UNION` and `UNION ALL` set operators, but not `INTERSECT`.

  Prefer `UNION ALL` over `UNION` when you know the data sets are disjoint or duplicate values are not a problem; `UNION ALL` is more efficient because it avoids materializing and sorting the entire result set to eliminate duplicate values.

- Impala requires query aliases for the subqueries used as inline views in the `FROM` clause.

  For example, without the alias `contents_of_t1` at the end, the following query gives a syntax error:

```
SELECT COUNT(*) FROM (SELECT * FROM t1) contents_of_t1;
```

Aliases are not required for the subqueries used in other parts of queries. For example:

```
SELECT * FROM functional.alltypes WHERE id = (SELECT MIN(id) FROM functional.alltypes);
```

- When an alias is declared for an expression in a query, that alias cannot be referenced again within the same SELECT list.

  For example, the average alias cannot be referenced twice in the SELECT list as below. You will receive an error:

```
SELECT AVG(x) AS average, average+1 FROM t1 GROUP BY x;
```

  An alias can be referenced again in the same query if not in the SELECT list. For example, the average alias can be referenced twice as shown below:

```
SELECT AVG(x) AS average FROM t1 GROUP BY x HAVING average > 3;
```

- Impala does not support NATURAL JOIN, and it does not support the USING clause in joins. See Joins in Impala SELECT Statements on page 296 for details on the syntax for Impala join clauses.
- Impala supports a limited choice of partitioning types.

  Partitions are defined based on each distinct combination of values for one or more partition key columns. Impala does not redistribute or check data to create evenly distributed partitions. You must choose partition key columns based on your knowledge of the data volume and distribution. Adapt any tables that use range, list, hash, or key partitioning to use the Impala partition syntax for CREATE TABLE and ALTER TABLE statements.

  Impala partitioning is similar to range partitioning where every range has exactly one value, or key partitioning where the hash function produces a separate bucket for every combination of key values. See Partitioning for Impala Tables on page 625 for usage details, and CREATE TABLE Statement on page 234 and ALTER TABLE Statement on page 205 for syntax.

  Note: Because the number of separate partitions is potentially higher than in other database systems, keep a close eye on the number of partitions and the volume of data in each one; scale back the number of partition key columns if you end up with too many partitions with a small volume of data in each one.

  To distribute work for a query across a cluster, you need at least one HDFS block per node. HDFS blocks are typically multiple megabytes, especially for Parquet files. Therefore, if each partition holds only a few megabytes of data, you are unlikely to see much parallelism in the query because such a small amount of data is typically processed by a single node.

- For the "top-N" queries, Impala uses the LIMIT clause rather than comparing against a pseudo column named ROWNUM or ROW_NUM.

  See LIMIT Clause on page 308 for details.

SQL Constructs to Double-check

Some SQL constructs that are supported have behavior or defaults more oriented towards convenience than optimal performance. Also, sometimes machine-generated SQL, perhaps issued through JDBC or ODBC applications, might have inefficiencies or exceed internal Impala limits. As you port SQL code, examine and possibly update the following where appropriate:

- A CREATE TABLE statement with no STORED AS clause creates data files in plain text format, which is convenient for data interchange but not a good choice for high-volume data with high-performance queries. See How Impala Works with Hadoop File Formats on page 634 for why and how to use specific file formats for compact data and high-performance queries. Especially see Using the Parquet File Format with Impala Tables on page 643, for details about the file format most heavily optimized for large-scale data warehouse queries.

- Adapting tables that were already partitioned in a different database system could produce an Impala table with a high number of partitions and not enough data in each one, leading to underutilization of Impala's parallel query features.

  See Partitioning for Impala Tables on page 625 for details about setting up partitioning and tuning the performance of queries on partitioned tables.

- The `INSERT ... VALUES` syntax is suitable for setting up toy tables with a few rows for functional testing when used with HDFS. Each such statement creates a separate tiny file in HDFS, and it is not a scalable technique for loading megabytes or gigabytes (let alone petabytes) of data.

  Consider revising your data load process to produce raw data files outside of Impala, then setting up Impala external tables or using the `LOAD DATA` statement to use those data files instantly in Impala tables, with no conversion or indexing stage. See External Tables on page 197 and LOAD DATA Statement on page 288 for details about the Impala techniques for working with data files produced outside of Impala; see Data Loading and Querying Examples on page 52 for examples of ETL workflow for Impala.

  `INSERT` works fine for Kudu tables even though not particularly fast.

- If your ETL process is not optimized for Hadoop, you might end up with highly fragmented small data files, or a single giant data file that cannot take advantage of distributed parallel queries or partitioning. In this case, use an `INSERT ... SELECT` statement to copy the data into a new table and reorganize into a more efficient layout in the same operation. See INSERT Statement on page 277 for details about the `INSERT` statement.

  You can do `INSERT ... SELECT` into a table with a more efficient file format (see How Impala Works with Hadoop File Formats on page 634) or from an unpartitioned table into a partitioned one. See Partitioning for Impala Tables on page 625.

- Complex queries may have high codegen time. As a workaround, set the query option `DISABLE_CODEGEN=true` if queries fail for this reason. See DISABLE_CODEGEN Query Option on page 327 for details.

- If practical, rewrite `UNION` queries to use the `UNION ALL` operator instead. Prefer `UNION ALL` over `UNION` when you know the data sets are disjoint or duplicate values are not a problem; `UNION ALL` is more efficient because it avoids materializing and sorting the entire result set to eliminate duplicate values.

Next Porting Steps after Verifying Syntax and Semantics

Some of the decisions you make during the porting process can have an impact on performance. After your SQL code is ported and working correctly, examine the performance-related aspects of your schema design, physical layout, and queries to make sure that the ported application is taking full advantage of Impala's parallelism, performance-related SQL features, and integration with Hadoop components. The following are a few of the areas you should examine:

- For the optimal performance, we recommend that you run `COMPUTE STATS` on all tables.
- Use the most efficient file format for your data volumes, table structure, and query characteristics.
- Partition on columns that are often used for filtering in `WHERE` clauses.
- Your ETL process should produce a relatively small number of multi-megabyte data files rather than a huge number of small files.

See Tuning Impala for Performance on page 565 for details about the performance tuning process.

Resource Management

Impala includes the features that balance and maximize resources in CDH clusters. This topic describes how you can improve efficiency of your CDH cluster using those features.

- **Static service pools**

  Use the static service pools to allocate dedicated resources for Impala to manage and prioritize workloads on clusters.

- **Admission control**

  Within the constraints of the static service pool, you can further subdivide Impala's resources using dynamic resource pools and admission control.

See Admission Control and Query Queuing on page 549 for an overview and guidelines on admission control.

See Configuring Resource Pools and Admission Control on page 554 for configuring resource pools and managing admission control.

Admission Control and Query Queuing

Admission control is an Impala feature that imposes limits on concurrent SQL queries, to avoid resource usage spikes and out-of-memory conditions on busy CDH clusters. The admission control feature lets you set an upper limit on the number of concurrent Impala queries and on the memory used by those queries. Any additional queries are queued until the earlier ones finish, rather than being cancelled or running slowly and causing contention. As other queries finish, the queued queries are allowed to proceed.

In CDH 5.7 / Impala 2.5 and higher, you can specify these limits and thresholds for each pool rather than globally. That way, you can balance the resource usage and throughput between steady well-defined workloads, rare resource-intensive queries, and ad-hoc exploratory queries.

In addition to the threshold values for currently executing queries, you can place limits on the maximum number of queries that are queued (waiting) and a limit on the amount of time they might wait before returning with an error. These queue settings let you ensure that queries do not wait indefinitely so that you can detect and correct “starvation” scenarios.

Queries, DML statements, and some DDL statements, including `CREATE TABLE AS SELECT` and `COMPUTE STATS` are affected by admission control.

On a busy CDH cluster, you might find there is an optimal number of Impala queries that run concurrently. For example, when the I/O capacity is fully utilized by I/O-intensive queries, you might not find any throughput benefit in running more concurrent queries. By allowing some queries to run at full speed while others wait, rather than having all queries contend for resources and run slowly, admission control can result in higher overall throughput.

For another example, consider a memory-bound workload such as many large joins or aggregation queries. Each such query could briefly use many gigabytes of memory to process intermediate results. Because Impala by default cancels queries that exceed the specified memory limit, running multiple large-scale queries at once might require re-running some queries that are cancelled. In this case, admission control improves the reliability and stability of the overall workload by only allowing as many concurrent queries as the overall memory of the cluster can accommodate.

Concurrent Queries and Admission Control

One way to limit resource usage through admission control is to set an upper limit on the number of concurrent queries. This is the initial technique you might use when you do not have extensive information about memory usage for your workload. The settings can be specified separately for each dynamic resource pool.

Max Running Queries

Maximum number of concurrently running queries in this pool. The default value is unlimited for CDH 5.7 or higher. (optional)

The maximum number of queries that can run concurrently in this pool. The default value is unlimited. Any queries for this pool that exceed **Max Running Queries** are added to the admission control queue until other queries finish. You can use **Max Running Queries** in the early stages of resource management, when you do not have extensive data about query memory usage, to determine if the cluster performs better overall if throttling is applied to Impala queries.

For a workload with many small queries, you typically specify a high value for this setting, or leave the default setting of "unlimited". For a workload with expensive queries, where some number of concurrent queries saturate the memory, I/O, CPU, or network capacity of the cluster, set the value low enough that the cluster resources are not overcommitted for Impala.

Once you have enabled memory-based admission control using other pool settings, you can still use **Max Running Queries** as a safeguard. If queries exceed either the total estimated memory or the maximum number of concurrent queries, they are added to the queue.

Max Queued Queries

Maximum number of queries that can be queued in this pool. The default value is 200 for CDH 5.3 or higher and 50 for previous versions of Impala. (optional)

Queue Timeout

The amount of time, in milliseconds, that a query waits in the admission control queue for this pool before being canceled. The default value is 60,000 milliseconds.

In the following cases, **Queue Timeout** is not significant, and you can specify a high value to avoid canceling queries unexpectedly:

- In a low-concurrency workload where few or no queries are queued
- In an environment without a strict SLA, where it does not matter if queries occasionally take longer than usual because they are held in admission control

You might also need to increase the value to use Impala with some business intelligence tools that have their own timeout intervals for queries.

In a high-concurrency workload, especially for queries with a tight SLA, long wait times in admission control can cause a serious problem. For example, if a query needs to run in 10 seconds, and you have tuned it so that it runs in 8 seconds, it violates its SLA if it waits in the admission control queue longer than 2 seconds. In a case like this, set a low timeout value and monitor how many queries are cancelled because of timeouts. This technique helps you to discover capacity, tuning, and scaling problems early, and helps avoid wasting resources by running expensive queries that have already missed their SLA.

If you identify some queries that can have a high timeout value, and others that benefit from a low timeout value, you can create separate pools with different values for this setting.

You can combine these settings with the memory-based approach described in Memory Limits and Admission Control on page 550. If either the maximum number of or the expected memory usage of the concurrent queries is exceeded, subsequent queries are queued until the concurrent workload falls below the threshold again.

Memory Limits and Admission Control

Each dynamic resource pool can have an upper limit on the cluster-wide memory used by queries executing in that pool.

Use the following settings to manage memory-based admission control.

Max Memory

The maximum amount of aggregate memory available across the cluster to all queries executing in this pool. This should be a portion of the aggregate configured memory for Impala daemons, which will be shown in the settings dialog next to this option for convenience. Setting this to a non-zero value enables memory based admission control.

Impala determines the expected maximum memory used by all queries in the pool and holds back any further queries that would result in **Max Memory** being exceeded.

If you specify **Max Memory**, you should specify the amount of memory to allocate to each query in this pool. You can do this in two ways:

- By setting **Maximum Query Memory Limit** and **Minimum Query Memory Limit**. This is preferred in CDH 6.1 / Impala 3.1 and greater and gives Impala flexibility to set aside more memory to queries that are expected to be memory-hungry.
- By setting **Default Query Memory Limit** to the exact amount of memory that Impala should set aside for queries in that pool.

Note that in the following cases, Impala will rely entirely on memory estimates to determine how much memory to set aside for each query. This is not recommended because it can result in queries not running or being starved for memory if the estimates are inaccurate. And it can affect other queries running on the same node.

- Max Memory, Maximum Query Memory Limit, and Minimum Query Memory Limit are not set, and the `MEM_LIMIT` query option is not set for the query.
- Default Query Memory Limit is set to 0, and the `MEM_LIMIT` query option is not set for the query.

Minimum Query Memory Limit and Maximum Query Memory Limit

These two options determine the minimum and maximum per-host memory limit that will be chosen by Impala Admission control for queries in this resource pool. If set, Impala admission control will choose a memory limit between the minimum and maximum values based on the per-host memory estimate for the query. The memory limit chosen determines the amount of memory that Impala admission control will set aside for this query on each host that the query is running on. The aggregate memory across all of the hosts that the query is running on is counted against the pool's **Max Memory**.

Minimum Query Memory Limit must be less than or equal to **Maximum Query Memory Limit** and **Max Memory**.

A user can override Impala's choice of memory limit by setting the `MEM_LIMIT` query option. If the **Clamp MEM_LIMIT Query Option** setting is set to `TRUE` and the user sets `MEM_LIMIT` to a value that is outside of the range specified by these two options, then the effective memory limit will be either the minimum or maximum, depending on whether `MEM_LIMIT` is lower than or higher the range.

For example, assume a resource pool with the following parameters set:

- Minimum Query Memory Limit = 2GB
- Maximum Query Memory Limit = 10GB

If a user tries to submit a query with the `MEM_LIMIT` query option set to 14 GB, the following would happen:

- If Clamp MEM_LIMIT Query Option = true, admission controller would override `MEM_LIMIT` with 10 GB and attempt admission using that value.
- If Clamp MEM_LIMIT Query Option = false, the admission controller will retain the `MEM_LIMIT` of 14 GB set by the user and will attempt admission using the value.

Default Query Memory Limit

The default memory limit applied to queries executing in this pool when no explicit `MEM_LIMIT` query option is set. The memory limit chosen determines the amount of memory that Impala Admission control will set aside for this query on each host that the query is running on. The aggregate memory across all of the hosts that the query is running on is counted against the pool's **Max Memory**. This option is deprecated in CDH 6.1 / Impala 3.1 and higher and is replaced by **Maximum Query Memory Limit** and **Minimum Query Memory Limit**.

Do not set this if either **Maximum Query Memory Limit** or **Minimum Query Memory Limit** is set.

Clamp MEM_LIMIT Query Option

If this field is not selected, the `MEM_LIMIT` query option will not be bounded by the **Maximum Query Memory Limit** and the **Minimum Query Memory Limit** values specified for this resource pool. By default, this field is selected in CDH 6.1 and higher. The field is disabled if both **Minimum Query Memory Limit** and **Maximum Query Memory Limit** are not set.

For example, consider the following scenario:

- The cluster is running `impalad` daemons on five hosts.
- A dynamic resource pool has **Max Memory** set to 100 GB.
- The **Maximum Query Memory Limit** for the pool is 10 GB and **Minimum Query Memory Limit** is 2 GB. Therefore, any query running in this pool could use up to 50 GB of memory (**Maximum Query Memory Limit** * number of Impala nodes).
- Impala will execute varying numbers of queries concurrently because queries may be given memory limits anywhere between 2 GB and 10 GB, depending on the estimated memory requirements. For example, Impala may execute up to 10 small queries with 2 GB memory limits or two large queries with 10 GB memory limits because that is what will fit in the 100 GB cluster-wide limit when executing on five hosts.
- The executing queries may use less memory than the per-host memory limit or the **Max Memory** cluster-wide limit if they do not need that much memory. In general this is not a problem so long as you are able to execute enough queries concurrently to meet your needs.

You can combine the memory-based settings with the upper limit on concurrent queries described in Concurrent Queries and Admission Control on page 549. If either the maximum number of or the expected memory usage of the concurrent queries is exceeded, subsequent queries are queued until the concurrent workload falls below the threshold again.

How Impala Admission Control Relates to Other Resource Management Tools

The admission control feature is similar in some ways to the Cloudera Manager static partitioning feature, as well as the YARN resource management framework. These features can be used separately or together. This section describes some similarities and differences, to help you decide which combination of resource management features to use for Impala.

Admission control is a lightweight, decentralized system that is suitable for workloads consisting primarily of Impala queries and other SQL statements. It sets “soft” limits that smooth out Impala memory usage during times of heavy load, rather than taking an all-or-nothing approach that cancels jobs that are too resource-intensive.

Because the admission control system does not interact with other Hadoop workloads such as MapReduce jobs, you might use YARN with static service pools on CDH clusters where resources are shared between Impala and other Hadoop components. This configuration is recommended when using Impala in a ***multitenant*** cluster. Devote a percentage of cluster resources to Impala, and allocate another percentage for MapReduce and other batch-style workloads. Let admission control handle the concurrency and memory usage for the Impala work within the cluster, and let YARN manage the work for other components within the cluster. In this scenario, Impala's resources are not managed by YARN.

The Impala admission control feature uses the same configuration mechanism as the YARN resource manager to map users to pools and authenticate them.

Although the Impala admission control feature uses a `fair-scheduler.xml` configuration file behind the scenes, this file does not depend on which scheduler is used for YARN. You still use this file, and Cloudera Manager can generate it for you, even when YARN is using the capacity scheduler.

How Impala Schedules and Enforces Limits on Concurrent Queries

The admission control system is decentralized, embedded in each Impala daemon and communicating through the statestore mechanism. Although the limits you set for memory usage and number of concurrent queries apply cluster-wide, each Impala daemon makes its own decisions about whether to allow each query to run immediately or to queue it for a less-busy time. These decisions are fast, meaning the admission control mechanism is low-overhead, but might be imprecise during times of heavy load across many coordinators. There could be times when the more

queries were queued (in aggregate across the cluster) than the specified limit, or when number of admitted queries exceeds the expected number. Thus, you typically err on the high side for the size of the queue, because there is not a big penalty for having a large number of queued queries; and you typically err on the low side for configuring memory resources, to leave some headroom in case more queries are admitted than expected, without running out of memory and being cancelled as a result.

To avoid a large backlog of queued requests, you can set an upper limit on the size of the queue for queries that are queued. When the number of queued queries exceeds this limit, further queries are cancelled rather than being queued. You can also configure a timeout period per pool, after which queued queries are cancelled, to avoid indefinite waits. If a cluster reaches this state where queries are cancelled due to too many concurrent requests or long waits for query execution to begin, that is a signal for an administrator to take action, either by provisioning more resources, scheduling work on the cluster to smooth out the load, or by doing Impala performance tuning to enable higher throughput.

How Admission Control works with Impala Clients (JDBC, ODBC, HiveServer2)

Most aspects of admission control work transparently with client interfaces such as JDBC and ODBC:

- If a SQL statement is put into a queue rather than running immediately, the API call blocks until the statement is dequeued and begins execution. At that point, the client program can request to fetch results, which might also block until results become available.
- If a SQL statement is cancelled because it has been queued for too long or because it exceeded the memory limit during execution, the error is returned to the client program with a descriptive error message.

In Impala 2.0 and higher, you can submit a SQL SET statement from the client application to change the REQUEST_POOL query option. This option lets you submit queries to different resource pools, as described in REQUEST_POOL Query Option on page 355.

At any time, the set of queued queries could include queries submitted through multiple different Impala daemon hosts. All the queries submitted through a particular host will be executed in order, so a CREATE TABLE followed by an INSERT on the same table would succeed. Queries submitted through different hosts are not guaranteed to be executed in the order they were received. Therefore, if you are using load-balancing or other round-robin scheduling where different statements are submitted through different hosts, set up all table structures ahead of time so that the statements controlled by the queuing system are primarily queries, where order is not significant. Or, if a sequence of statements needs to happen in strict order (such as an INSERT followed by a SELECT), submit all those statements through a single session, while connected to the same Impala daemon host.

Admission control has the following limitations or special behavior when used with JDBC or ODBC applications:

- The other resource-related query options, RESERVATION_REQUEST_TIMEOUT and V_CPU_CORES, are no longer used. Those query options only applied to using Impala with Llama, which is no longer supported.

SQL and Schema Considerations for Admission Control

When queries complete quickly and are tuned for optimal memory usage, there is less chance of performance or capacity problems during times of heavy load. Before setting up admission control, tune your Impala queries to ensure that the query plans are efficient and the memory estimates are accurate. Understanding the nature of your workload, and which queries are the most resource-intensive, helps you to plan how to divide the queries into different pools and decide what limits to define for each pool.

For large tables, especially those involved in join queries, keep their statistics up to date after loading substantial amounts of new data or adding new partitions. Use the COMPUTE STATS statement for unpartitioned tables, and COMPUTE INCREMENTAL STATS for partitioned tables.

When you use dynamic resource pools with a **Max Memory** setting enabled, you typically override the memory estimates that Impala makes based on the statistics from the COMPUTE STATS statement. You either set the MEM_LIMIT query option within a particular session to set an upper memory limit for queries within that session, or a default MEM_LIMIT setting for all queries processed by the impalad instance, or a default MEM_LIMIT setting for all queries assigned to a particular dynamic resource pool. By designating a consistent memory limit for a set of similar queries that use the same resource pool, you avoid unnecessary query queuing or out-of-memory conditions that can arise during high-concurrency workloads when memory estimates for some queries are inaccurate.

Follow other steps from Tuning Impala for Performance on page 565 to tune your queries.

Guidelines for Using Admission Control

The limits imposed by admission control are de-centrally managed “soft” limits. Each Impala coordinator node makes its own decisions about whether to allow queries to run immediately or to queue them. These decisions rely on information passed back and forth between nodes by the StateStore service. If a sudden surge in requests causes more queries than anticipated to run concurrently, then the throughput could decrease due to queries spilling to disk or contending for resources. Or queries could be cancelled if they exceed the `MEM_LIMIT` setting while running.

In `impala-shell`, you can also specify which resource pool to direct queries to by setting the `REQUEST_POOL` query option.

If you set up different resource pools for different users and groups, consider reusing any classifications you developed for use with Sentry security. See Enabling Sentry Authorization for Impala on page 87 for details.

Where practical, use Cloudera Manager to configure the admission control parameters. The Cloudera Manager GUI is much simpler than editing the configuration files directly.

To see how admission control works for particular queries, examine the profile output or the summary output for the query.

- Profile

  The information is available through the `PROFILE` statement in `impala-shell` immediately after running a query in the shell, on the **queries** page of the Impala debug web UI, or in the Impala log file (basic information at log level 1, more detailed information at log level 2).

  The profile output contains details about the admission decision, such as whether the query was queued or not and which resource pool it was assigned to. It also includes the estimated and actual memory usage for the query, so you can fine-tune the configuration for the memory limits of the resource pools.

- Summary

  Starting in CDH 6.1, the information is available in `impala-shell` when the `LIVE_PROGRESS` or `LIVE_SUMMARY` query option is set to `TRUE`.

  You can also start an `impala-shell` session with the `--live_progress` or `--live_summary` flags to monitor all queries in that `impala-shell` session.

  The summary output includes the queuing status consisting of whether the query was queued and what was the latest queuing reason.

For details about all the Fair Scheduler configuration settings, see Fair Scheduler Configuration, in particular the tags such as `<queue>` and `<aclSubmitApps>` to map users and groups to particular resource pools (queues).

Configuring Resource Pools and Admission Control

Impala includes features that balance and maximize resources in your CDH cluster. This topic describes how you can improve efficiency of your a CDH cluster using those features.

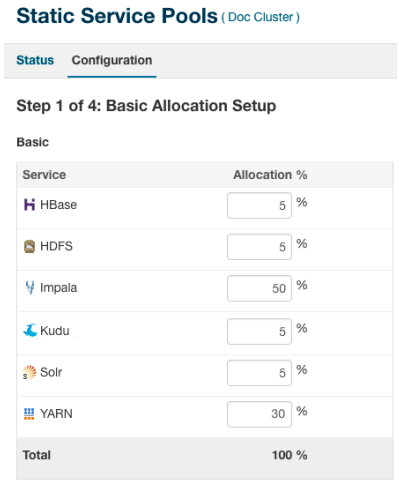
A typical deployment uses the following:

- Creating Static Service Pools
- Using Admission Control
  - Setting Per-query Memory Limits
  - Creating Dynamic Resource Pools

Creating Static Service Pools

To manage and prioritize workloads on clusters, use the static service pools to allocate dedicated resources for Impala for predictable resource availability. When static service pools are used, Cloudera Manager creates a cgroup in which Impala runs. This cgroup limits memory, CPU and Disk I/O according to the static partitioning policy.

The following is a sample configuration for Static Service Pools shown in Cloudera Manager.



- HDFS always needs to have a minimum of 5-10% of the resources.
- Generally, YARN and Impala split the rest of the resources.
  - For mostly batch workloads, you might allocate YARN 60%, Impala 30%, and HDFS 10%.
  - For mostly ad-hoc query workloads, you might allocate Impala 60%, YARN 30%, and HDFS 10%.

Using Admission Control

Within the constraints of the static service pool, using dynamic resource pools and the admission control, you can further subdivide Impala's resource usage among dynamic resource pools in multitenant use cases.

Allocating resources judiciously allows your most important queries to run faster and more reliably.

Note: Impala dynamic resource pools are different than the default YARN dynamic resource pools. You can turn on dynamic resource pools that are exclusively for use by Impala.

Enabling or Disabling Impala Admission Control in Cloudera Manager

We recommend enabling admission control on all production clusters to alleviate possible capacity issues. The capacity issues could be because of a high volume of concurrent queries, because of heavy-duty join and aggregation queries that require large amounts of memory, or because Impala is being used alongside other Hadoop data management components and the resource usage of Impala must be constrained to work well in a multitenant deployment.

Important:

In CDH 5.8 and higher, admission control and dynamic resource pools are enabled by default. However, until you configure the settings for the dynamic resource pools, the admission control feature is effectively not enabled as, with the default the settings, each dynamic pool will allow all queries of any memory requirement to execute in the pool.

1. Go to the Impala service.

2. In the **Configuration** tab, select **Category** > **Admission Control**.
3. Select or clear both the **Enable Impala Admission Control** checkbox and the **Enable Dynamic Resource Pools** checkbox.
4. Enter a **Reason for change**, and then click **Save Changes** to commit the changes.
5. Restart the Impala service.

After completing this task, for further configuration settings, customize the configuration settings for the dynamic resource pools, as described in below.

Creating an Impala Dynamic Resource Pool

There is always a resource pool designated as `root.default`. By default, all Impala queries run in this pool when the dynamic resource pool feature is enabled for Impala. You create additional pools when your workload includes identifiable groups of queries (such as from a particular application, or a particular group within your organization) that have their own requirements for concurrency, memory use, or service level agreement (SLA). Each pool has its own settings related to memory, number of queries, and timeout interval.

1. Select **Clusters** > ***Cluster name*** > **Dynamic Resource Pool Configuration**. If the cluster has an Impala service, the **Resource Pools** tab displays under the **Impala Admission Control** tab.
2. Click the **Impala Admission Control** tab.
3. Click **Create Resource Pool**.
4. Specify a name and resource limits for the pool:
   - In the **Resource Pool Name** field, type a unique name containing only alphanumeric characters.
   - Optionally, click the **Submission Access Control** tab to specify which users and groups can submit queries. By default, anyone can submit queries. To restrict this permission, select the **Allow these users and groups** option and provide a comma-delimited list of users and groups in the Users and Groups fields respectively.
5. Click **Create**.
6. Click **Refresh Dynamic Resource Pools**.

Configuration Settings for Impala Dynamic Resource Pool

Impala dynamic resource pools support the following settings.

Max Memory

Maximum amount of aggregate memory available across the cluster to all queries executing in this pool. This should be a portion of the aggregate configured memory for Impala daemons, which will be shown in the settings dialog next to this option for convenience. Setting this to a non-zero value enables memory based admission control.

Impala determines the expected maximum memory used by all queries in the pool and holds back any further queries that would result in **Max Memory** being exceeded.

If you specify **Max Memory**, you should specify the amount of memory to allocate to each query in this pool. You can do this in two ways:

- By setting **Maximum Query Memory Limit** and **Minimum Query Memory Limit**. This is preferred in CDH 6.1 and higher and gives Impala flexibility to set aside more memory to queries that are expected to be memory-hungry.
- By setting **Default Query Memory Limit** to the exact amount of memory that Impala should set aside for queries in that pool.

Note that if you do not set any of the above options, or set **Default Query Memory Limit** to 0, Impala will rely entirely on memory estimates to determine how much memory to set aside for each query. This is not recommended because it can result in queries not running or being starved for memory if the estimates are inaccurate.

For example, consider the following scenario:

- The cluster is running `impalad` daemons on five hosts.
- A dynamic resource pool has **Max Memory** set to 100 GB.

- The **Maximum Query Memory Limit** for the pool is 10 GB and **Minimum Query Memory Limit** is 2 GB. Therefore, any query running in this pool could use up to 50 GB of memory (**Maximum Query Memory Limit** * number of Impala nodes).
- Impala will execute varying numbers of queries concurrently because queries may be given memory limits anywhere between 2 GB and 10 GB, depending on the estimated memory requirements. For example, Impala may execute up to 10 small queries with 2 GB memory limits or two large queries with 10 GB memory limits because that is what will fit in the 100 GB cluster-wide limit when executing on five hosts.
- The executing queries may use less memory than the per-host memory limit or the **Max Memory** cluster-wide limit if they do not need that much memory. In general this is not a problem so long as you are able to execute enough queries concurrently to meet your needs.

Minimum Query Memory Limit and Maximum Query Memory Limit

These two options determine the minimum and maximum per-host memory limit that will be chosen by Impala Admission control for queries in this resource pool. If set, Impala admission control will choose a memory limit between the minimum and maximum value based on the per-host memory estimate for the query. The memory limit chosen determines the amount of memory that Impala admission control will set aside for this query on each host that the query is running on. The aggregate memory across all of the hosts that the query is running on is counted against the pool's **Max Memory.**

Minimum Query Memory Limit must be less than or equal to **Maximum Query Memory Limit** and **Max Memory**.

You can override Impala's choice of memory limit by setting the `MEM_LIMIT` query option. If the **Clamp MEM_LIMIT Query Option** is selected and the user sets `MEM_LIMIT` to a value that is outside of the range specified by these two options, then the effective memory limit will be either the minimum or maximum, depending on whether `MEM_LIMIT` is lower than or higher than the range.

Default Query Memory Limit

The default memory limit applied to queries executing in this pool when no explicit `MEM_LIMIT` query option is set. The memory limit chosen determines the amount of memory that Impala Admission control will set aside for this query on each host that the query is running on. The aggregate memory across all of the hosts that the query is running on is counted against the pool's **Max Memory.**

This option is deprecated from CDH 6.1 and higher and is replaced by **Maximum Query Memory Limit** and **Minimum Query Memory Limit**. Do not set this field if either **Maximum Query Memory Limit** or **Minimum Query Memory Limit** is set.

Max Running Queries

Maximum number of concurrently running queries in this pool. The default value is unlimited for CDH 5.7 or higher. (optional)

The maximum number of queries that can run concurrently in this pool. The default value is unlimited. Any queries for this pool that exceed **Max Running Queries** are added to the admission control queue until other queries finish. You can use **Max Running Queries** in the early stages of resource management, when you do not have extensive data about query memory usage, to determine if the cluster performs better overall if throttling is applied to Impala queries.

For a workload with many small queries, you typically specify a high value for this setting, or leave the default setting of "unlimited". For a workload with expensive queries, where some number of concurrent queries saturate the memory, I/O, CPU, or network capacity of the cluster, set the value low enough that the cluster resources are not overcommitted for Impala.

Once you have enabled memory-based admission control using other pool settings, you can still use **Max Running Queries** as a safeguard. If queries exceed either the total estimated memory or the maximum number of concurrent queries, they are added to the queue.

Max Queued Queries

Maximum number of queries that can be queued in this pool. The default value is 200 for CDH 5.3 or higher and 50 for previous versions of Impala. (optional)

Queue Timeout

The amount of time, in milliseconds, that a query waits in the admission control queue for this pool before being canceled. The default value is 60,000 milliseconds.

It the following cases, **Queue Timeout** is not significant, and you can specify a high value to avoid canceling queries unexpectedly:

- In a low-concurrency workload where few or no queries are queued
- In an environment without a strict SLA, where it does not matter if queries occasionally take longer than usual because they are held in admission control

You might also need to increase the value to use Impala with some business intelligence tools that have their own timeout intervals for queries.

In a high-concurrency workload, especially for queries with a tight SLA, long wait times in admission control can cause a serious problem. For example, if a query needs to run in 10 seconds, and you have tuned it so that it runs in 8 seconds, it violates its SLA if it waits in the admission control queue longer than 2 seconds. In a case like this, set a low timeout value and monitor how many queries are cancelled because of timeouts. This technique helps you to discover capacity, tuning, and scaling problems early, and helps avoid wasting resources by running expensive queries that have already missed their SLA.

If you identify some queries that can have a high timeout value, and others that benefit from a low timeout value, you can create separate pools with different values for this setting.

Clamp MEM_LIMIT Query Option

If this field is not selected, the `MEM_LIMIT` query option will not be bounded by the **Maximum Query Memory Limit** and the **Minimum Query Memory Limit** values specified for this resource pool. By default, this field is selected in CDH 6.1 and higher. The field is disabled if both **Minimum Query Memory Limit** and **Maximum Query Memory Limit** are not set.

Setting Per-query Memory Limits

Use per-query memory limits to prevent queries from consuming excessive memory resources that impact other queries. Cloudera recommends that you set the query memory limits whenever possible.

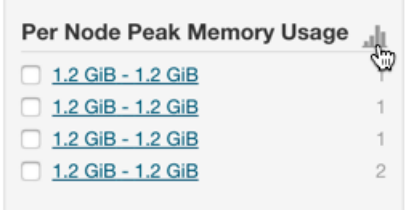
If you set the **Max Memory** for a resource pool, Impala attempts to throttle queries if there is not enough memory to run them within the specified resources.

Only use admission control with maximum memory resources if you can ensure there are query memory limits. Set the pool **Maximum Query Memory Limit** to be certain. You can override this setting with the `MEM_LIMIT` query option, if necessary.

Typically, you set query memory limits using the `set MEM_LIMIT=Xg;` query option. When you find the right value for your business case, memory-based admission control works well. The potential downside is that queries that attempt to use more memory might perform poorly or even be cancelled.

To find a reasonable Maximum Query Memory Limit:

1. Run the workload.
2. In Cloudera Manager, go to **Impala** > **Queries**.
3. Click **Select Attributes**.
4. Select **Per Node Peak Memory Usage** and click **Update**.
5. Allow the system time to gather information, then click the **Show Histogram** icon to see the results.



6. Use the histogram to find a value that accounts for most queries. Queries that require more resources than this limit should explicitly set the memory limit to ensure they can run to completion.

Configuring Admission Control in Command Line Interface

To configure admission control, use a combination of startup options for the Impala daemon and edit or create the configuration files `fair-scheduler.xml` and `llama-site.xml`.

For a straightforward configuration using a single resource pool named `default`, you can specify configuration options on the command line and skip the `fair-scheduler.xml` and `llama-site.xml` configuration files.

For an advanced configuration with multiple resource pools using different settings:

1. Set up the `fair-scheduler.xml` and `llama-site.xml` configuration files manually.
2. Provide the paths to each one using the `impalad` command-line options, `--fair_scheduler_allocation_path` and `--llama_site_path` respectively.

The Impala admission control feature only uses the Fair Scheduler configuration settings to determine how to map users and groups to different resource pools. For example, you might set up different resource pools with separate memory limits, and maximum number of concurrent and queued queries, for different categories of users within your organization. For details about all the Fair Scheduler configuration settings, see the Apache wiki.

The Impala admission control feature only uses a small subset of possible settings from the `llama-site.xml` configuration file:

```
llama.am.throttling.maximum.placed.reservations.queue_name
llama.am.throttling.maximum.queued.reservations.queue_name
impala.admission-control.pool-default-query-options.queue_name
impala.admission-control.pool-queue-timeout-ms.queue_name
```

The `impala.admission-control.pool-queue-timeout-ms` setting specifies the timeout value for this pool, in milliseconds.

The`impala.admission-control.pool-default-query-options` settings designates the default query options for all queries that run in this pool. Its argument value is a comma-delimited string of 'key=value' pairs, for example,`'key1=val1, key2=val2'`. For example, this is where you might set a default memory limit for all queries in the pool, using an argument such as `MEM_LIMIT=5G`.

The `impala.admission-control.*` configuration settings are available in CDH 5.7 / Impala 2.5 and higher.

Example Admission Control Configuration Files

For clusters not managed by Cloudera Manager, here are sample `fair-scheduler.xml` and `llama-site.xml` files that define resource pools `root.default`, `root.development`, and `root.production`. These files define resource pools for Impala admission control and are separate from the similar `fair-scheduler.xml` that defines resource pools for YARN.

fair-scheduler.xml:

Although Impala does not use the `vcores` value, you must still specify it to satisfy YARN requirements for the file contents.

Each `<aclSubmitApps>` tag (other than the one for `root`) contains a comma-separated list of users, then a space, then a comma-separated list of groups; these are the users and groups allowed to submit Impala statements to the corresponding resource pool.

If you leave the `<aclSubmitApps>` element empty for a pool, nobody can submit directly to that pool; child pools can specify their own `<aclSubmitApps>` values to authorize users and groups to submit to those pools.

```
<allocations>
    <queue name="root">
        <aclSubmitApps> </aclSubmitApps>
        <queue name="default">
            <maxResources>50000 mb, 0 vcores</maxResources>
```

```
            <aclSubmitApps>*</aclSubmitApps>
        </queue>
        <queue name="development">
            <maxResources>200000 mb, 0 vcores</maxResources>
            <aclSubmitApps>user1,user2 dev,ops,admin</aclSubmitApps>
        </queue>
        <queue name="production">
            <maxResources>1000000 mb, 0 vcores</maxResources>
            <aclSubmitApps> ops,admin</aclSubmitApps>
        </queue>
    </queue>
    <queuePlacementPolicy>
        <rule name="specified" create="false"/>
        <rule name="default" />
    </queuePlacementPolicy>
</allocations>
```

llama-site.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
    <name>llama.am.throttling.maximum.placed.reservations.root.default</name>
    <value>10</value>
  </property>
  <property>
    <name>llama.am.throttling.maximum.queued.reservations.root.default</name>
    <value>50</value>
  </property>
  <property>
    <name>impala.admission-control.pool-default-query-options.root.default</name>
    <value>mem_limit=128m,query_timeout_s=20,max_io_buffers=10</value>
  </property>
  <property>
    <name>impala.admission-control.pool-queue-timeout-ms.root.default</name>
    <value>30000</value>
  </property>
  <property>
    <name>impala.admission-control.max-query-mem-limit.root.default.regularPool</name>
    <value>1610612736</value><!--1.5GB-->
  </property>
  <property>
    <name>impala.admission-control.min-query-mem-limit.root.default.regularPool</name>
    <value>52428800</value><!--50MB-->
  </property>
  <property>

<name>impala.admission-control.clamp-mem-limit-query-option.root.default.regularPool</name>

    <value>true</value>
  </property>
```

Configuring Cluster-wide Admission Control



Important: Although the following options are still present in the Cloudera Manager interface under the **Admission Control** configuration settings dialog, avoid using them in CDH 5.7 / Impala 2.5 and higher. These settings only apply if you enable admission control but leave dynamic resource pools disabled. In CDH 5.7 / Impala 2.5 and higher, we recommend that you set up dynamic resource pools and customize the settings for each pool as described in Using Admission Control on page 555.

The following Impala configuration options let you adjust the settings of the admission control feature. When supplying the options on the `impalad` command line, prepend the option name with `--`.

`queue_wait_timeout_ms`

Purpose: Maximum amount of time (in milliseconds) that a request waits to be admitted before timing out.

Type: `int64`

Default: `60000`

`default_pool_max_requests`

Purpose: Maximum number of concurrent outstanding requests allowed to run before incoming requests are queued. Because this limit applies cluster-wide, but each Impala node makes independent decisions to run queries immediately or queue them, it is a soft limit; the overall number of concurrent queries might be slightly higher during times of heavy load. A negative value indicates no limit. Ignored if `fair_scheduler_config_path` and `llama_site_path` are set.

Type: `int64`

Default: -1, meaning unlimited (prior to CDH 5.7 / Impala 2.5 the default was 200)

`default_pool_max_queued`

Purpose: Maximum number of requests allowed to be queued before rejecting requests. Because this limit applies cluster-wide, but each Impala node makes independent decisions to run queries immediately or queue them, it is a soft limit; the overall number of queued queries might be slightly higher during times of heavy load. A negative value or 0 indicates requests are always rejected once the maximum concurrent requests are executing. Ignored if `fair_scheduler_config_path` and `llama_site_path` are set.

Type: `int64`

Default: unlimited

`default_pool_mem_limit`

Purpose: Maximum amount of memory (across the entire cluster) that all outstanding requests in this pool can use before new requests to this pool are queued. Specified in bytes, megabytes, or gigabytes by a number followed by the suffix b (optional), `m`, or `g`, either uppercase or lowercase. You can specify floating-point values for megabytes and gigabytes, to represent fractional numbers such as `1.5`. You can also specify it as a percentage of the physical memory by specifying the suffix `%`. 0 or no setting indicates no limit. Defaults to bytes if no unit is given. Because this limit applies cluster-wide, but each Impala node makes independent decisions to run queries immediately or queue them, it is a soft limit; the overall memory used by concurrent queries might be slightly higher during times of heavy load. Ignored if `fair_scheduler_config_path` and `llama_site_path` are set.



Note: Impala relies on the statistics produced by the `COMPUTE STATS` statement to estimate memory usage for each query. See COMPUTE STATS Statement on page 219 for guidelines about how and when to use this statement.

Type: string

Default: `""` (empty string, meaning unlimited)

`disable_pool_max_requests`

Purpose: Disables all per-pool limits on the maximum number of running requests.

Type: Boolean

Default: `false`

`disable_pool_mem_limits`

Purpose: Disables all per-pool mem limits.

Type: Boolean

Default: `false`

`fair_scheduler_allocation_path`

Purpose: Path to the fair scheduler allocation file (`fair-scheduler.xml`).

Type: string

Default: "" (empty string)

Usage notes: Admission control only uses a small subset of the settings that can go in this file, as described below. For details about all the Fair Scheduler configuration settings, see the Apache wiki.

`llama_site_path`

Purpose: Path to the configuration file used by admission control (`llama-site.xml`). If set, `fair_scheduler_allocation_path` must also be set.

Type: string

Default: "" (empty string)

Usage notes: Admission control only uses a few of the settings that can go in this file, as described below.

Admission Control Sample Scenario

Anne Chang is administrator for an enterprise data hub that runs a number of workloads, including Impala.

Anne has a 20-node cluster that uses Cloudera Manager static partitioning. Because of the heavy Impala workload, Anne needs to make sure Impala gets enough resources. While the best configuration values might not be known in advance, she decides to start by allocating 50% of resources to Impala. Each node has 128 GiB dedicated to each impalad. Impala has 2560 GiB in aggregate that can be shared across the resource pools she creates.

Next, Anne studies the workload in more detail. After some research, she might choose to revisit these initial values for static partitioning.

To figure out how to further allocate Impala's resources, Anne needs to consider the workloads and users, and determine their requirements. There are a few main sources of Impala queries:

- Large reporting queries executed by an external process/tool. These are critical business intelligence queries that are important for business decisions. It is important that they get the resources they need to run. There typically are not many of these queries at a given time.
- Frequent, small queries generated by a web UI. These queries scan a limited amount of data and do not require expensive joins or aggregations. These queries are important, but not as critical, perhaps the client tries resending the query or the end user refreshes the page.
- Occasionally, expert users might run ad-hoc queries. The queries can vary significantly in their resource requirements. While Anne wants a good experience for these users, it is hard to control what they do (for example, submitting inefficient or incorrect queries by mistake). Anne restricts these queries by default and tells users to reach out to her if they need more resources.

To set up admission control for this workload, Anne first runs the workloads independently, so that she can observe the workload's resource usage in Cloudera Manager. If they could not easily be run manually, but had been run in the past, Anne uses the history information from Cloudera Manager. It can be helpful to use other search criteria (for example, *user*) to isolate queries by workload. Anne uses the Cloudera Manager chart for Per-Node Peak Memory usage to identify the maximum memory requirements for the queries.

From this data, Anne observes the following about the queries in the groups above:

- Large reporting queries use up to 32 GiB per node. There are typically 1 or 2 queries running at a time. On one occasion, she observed that 3 of these queries were running concurrently. Queries can take 3 minutes to complete.
- Web UI-generated queries use between 100 MiB per node to usually less than 4 GiB per node of memory, but occasionally as much as 10 GiB per node. Queries take, on average, 5 seconds, and there can be as many as 140 incoming queries per minute.
- Anne has little data on ad hoc queries, but some are trivial (approximately 100 MiB per node), others join several tables (requiring a few GiB per node), and one user submitted a huge cross join of all tables that used all system resources (that was likely a mistake).

Based on these observations, Anne creates the admission control configuration with the following pools:

XL_Reporting

Property	Value
Max Memory	1280 GiB
Maximum Query Memory Limit	32 GiB
Minimum Query Memory Limit	32 GiB
Max Running Queries	2
Queue Timeout	5 minutes

This pool is for large reporting queries. To support running 2 queries at a time, the pool memory resources are set to 1280 GiB (aggregate cluster memory). This is for 2 queries, each with 32 GiB per node, across 20 nodes. Anne sets the pool's Maximum Query Memory Limit to 32 GiB so that no query uses more than 32 GiB on any given node. She sets Max Running Queries to 2 (though it is not necessary she do so). She increases the pool's queue timeout to 5 minutes in case a third query comes in and has to wait. She does not expect more than 3 concurrent queries, and she does not want them to wait that long anyway, so she does not increase the queue timeout. If the workload increases in the future, she might choose to adjust the configuration or buy more hardware.

HighThroughput_UI

Property	Value
Max Memory	960 GiB (inferred)
Maximum Query Memory Limit	4 GiB
Minimum Query Memory Limit	2 GiB
Max Running Queries	12
Queue Timeout	5 minutes

This pool is used for the small, high throughput queries generated by the web tool. Anne sets the Maximum Query Memory Limit to 4 GiB per node, and sets Max Running Queries to 12. This implies a maximum amount of memory per node used by the queries in this pool: 48 GiB per node (12 queries * 4 GiB per node memory limit).

Notice that Anne does not set the pool memory resources, but does set the pool's Maximum Query Memory Limit. This is intentional: admission control processes queries faster when a pool uses the Max Running Queries limit instead of the peak memory resources.

This should be enough memory for most queries, since only a few go over 4 GiB per node. For those that do require more memory, they can probably still complete with less memory (spilling if necessary). If, on occasion, a query cannot run with this much memory and it fails, Anne might reconsider this configuration later, or perhaps she does not need to worry about a few rare failures from this web UI.

With regard to throughput, since these queries take around 5 seconds and she is allowing 12 concurrent queries, the pool should be able to handle approximately 144 queries per minute, which is enough for the peak maximum expected of 140 queries per minute. In case there is a large burst of queries, Anne wants them to queue. The default maximum size of the queue is already 200, which should be more than large enough. Anne does not need to change it.

Default

Property	Value
Max Memory	320 GiB
Maximum Query Memory Limit	4 GiB
Minimum Query Memory Limit	2 GiB

Property	Value
Max Running Queries	Unlimited
Queue Timeout	60 Seconds

The default pool (which already exists) is a catch all for ad-hoc queries. Anne wants to use the remaining memory not used by the first two pools, 16 GiB per node (XL_Reporting uses 64 GiB per node, High_Throughput_UI uses 48 GiB per node). For the other pools to get the resources they expect, she must still set the Max Memory resources and the Maximum Query Memory Limit. She sets the Max Memory resources to 320 GiB (16 * 20). She sets the Maximum Query Memory Limit to 4 GiB per node for now. That is somewhat arbitrary, but satisfies some of the ad hoc queries she observed. If someone writes a bad query by mistake, she does not actually want it using all the system resources. If a user has a large query to submit, an expert user can override the Maximum Query Memory Limit (up to 16 GiB per node, since that is bound by the pool Max Memory resources). If that is still insufficient for this user's workload, the user should work with Anne to adjust the settings and perhaps create a dedicated pool for the workload.

Tuning Impala for Performance

The following sections explain the factors affecting the performance of Impala features, and procedures for tuning, monitoring, and benchmarking Impala queries and other SQL operations.

This section also describes techniques for maximizing Impala scalability. Scalability is tied to performance: it means that performance remains high as the system workload increases. For example, reducing the disk I/O performed by a query can speed up an individual query, and at the same time improve scalability by making it practical to run more queries simultaneously. Sometimes, an optimization technique improves scalability more than performance. For example, reducing memory usage for a query might not change the query performance much, but might improve scalability by allowing more Impala queries or other kinds of jobs to run at the same time without running out of memory.



Note:

Before starting any performance tuning or benchmarking, make sure your system is configured with all the recommended minimum hardware requirements from Hardware Requirements on page 24 and software settings from Post-Installation Configuration for Impala on page 36.

- Partitioning for Impala Tables on page 625. This technique physically divides the data based on the different values in frequently queried columns, allowing queries to skip reading a large percentage of the data in a table.
- Performance Considerations for Join Queries on page 568. Joins are the main class of queries that you can tune at the SQL level, as opposed to changing physical factors such as the file format or the hardware configuration. The related topics Overview of Column Statistics on page 576 and Overview of Table Statistics on page 575 are also important primarily for join performance.
- Overview of Table Statistics on page 575 and Overview of Column Statistics on page 576. Gathering table and column statistics, using the `COMPUTE STATS` statement, helps Impala automatically optimize the performance for join queries, without requiring changes to SQL query statements. (This process is greatly simplified in Impala 1.2.2 and higher, because the `COMPUTE STATS` statement gathers both kinds of statistics in one operation, and does not require any setup and configuration as was previously necessary for the `ANALYZE TABLE` statement in Hive.)
- Testing Impala Performance on page 601. Do some post-setup testing to ensure Impala is using optimal settings for performance, before conducting any benchmark tests.
- Benchmarking Impala Queries on page 588. The configuration and sample data that you use for initial experiments with Impala is often not appropriate for doing performance tests.
- Controlling Impala Resource Usage on page 588. The more memory Impala can utilize, the better query performance you can expect. In a cluster running other kinds of workloads as well, you must make tradeoffs to make sure all Hadoop components have enough memory to perform well, so you might cap the memory that Impala can use.
- Using Impala with the Amazon S3 Filesystem on page 692. Queries against data stored in the Amazon Simple Storage Service (S3) have different performance characteristics than when the data is stored in HDFS.

A good source of tips related to scalability and performance tuning is the Impala Cookbook presentation. These slides are updated periodically as new features come out and new benchmarks are performed.

Impala Performance Guidelines and Best Practices

Here are performance guidelines and best practices that you can use during planning, experimentation, and performance tuning for an Impala-enabled CDH cluster. All of this information is also available in more detail elsewhere in the Impala documentation; it is gathered together here to serve as a cookbook and emphasize which performance techniques typically provide the highest return on investment

Choose the appropriate file format for the data

Typically, for large volumes of data (multiple gigabytes per table or partition), the Parquet file format performs best because of its combination of columnar storage layout, large I/O request size, and compression and encoding. See How Impala Works with Hadoop File Formats on page 634 for comparisons of all file formats supported by Impala, and Using the Parquet File Format with Impala Tables on page 643 for details about the Parquet file format.

> **Note:** For smaller volumes of data, a few gigabytes or less for each table or partition, you might not see significant performance differences between file formats. At small data volumes, reduced I/O from an efficient compressed file format can be counterbalanced by reduced opportunity for parallel execution. When planning for a production deployment or conducting benchmarks, always use realistic data volumes to get a true picture of performance and scalability.

Avoid data ingestion processes that produce many small files

When producing data files outside of Impala, prefer either text format or Avro, where you can build up the files row by row. Once the data is in Impala, you can convert it to the more efficient Parquet format and split into multiple data files using a single `INSERT ... SELECT` statement. Or, if you have the infrastructure to produce multi-megabyte Parquet files as part of your data preparation process, do that and skip the conversion step inside Impala.

Always use `INSERT ... SELECT` to copy significant volumes of data from table to table within Impala. Avoid `INSERT ... VALUES` for any substantial volume of data or performance-critical tables, because each such statement produces a separate tiny data file. See INSERT Statement on page 277 for examples of the `INSERT ... SELECT` syntax.

For example, if you have thousands of partitions in a Parquet table, each with less than 256 MB of data, consider partitioning in a less granular way, such as by year / month rather than year / month / day. If an inefficient data ingestion process produces thousands of data files in the same table or partition, consider compacting the data by performing an `INSERT ... SELECT` to copy all the data to a different table; the data will be reorganized into a smaller number of larger files by this process.

Choose partitioning granularity based on actual data volume

Partitioning is a technique that physically divides the data based on values of one or more columns, such as by year, month, day, region, city, section of a web site, and so on. When you issue queries that request a specific value or range of values for the partition key columns, Impala can avoid reading the irrelevant data, potentially yielding a huge savings in disk I/O.

When deciding which column(s) to use for partitioning, choose the right level of granularity. For example, should you partition by year, month, and day, or only by year and month? Choose a partitioning strategy that puts at least 256 MB of data in each partition, to take advantage of HDFS bulk I/O and Impala distributed queries.

Over-partitioning can also cause query planning to take longer than necessary, as Impala prunes the unnecessary partitions. Ideally, keep the number of partitions in the table under 30 thousand.

When preparing data files to go in a partition directory, create several large files rather than many small ones. If you receive data in the form of many small files and have no control over the input format, consider using the `INSERT ... SELECT` syntax to copy data from one table or partition to another, which compacts the files into a relatively small number (based on the number of nodes in the cluster).

If you need to reduce the overall number of partitions and increase the amount of data in each partition, first look for partition key columns that are rarely referenced or are referenced in non-critical queries (not subject to an SLA). For example, your web site log data might be partitioned by year, month, day, and hour, but if most queries roll up the results by day, perhaps you only need to partition by year, month, and day.

If you need to reduce the granularity even more, consider creating “buckets”, computed values corresponding to different sets of partition key values. For example, you can use the `TRUNC( )` function with a `TIMESTAMP` column to group date and time values based on intervals such as week or quarter. See Impala Date and Time Functions on page 424 for details.

See Partitioning for Impala Tables on page 625 for full details and performance considerations for partitioning.

Use smallest appropriate integer types for partition key columns

Although it is tempting to use strings for partition key columns, since those values are turned into HDFS directory names anyway, you can minimize memory usage by using numeric values for common partition key fields such as YEAR, MONTH, and DAY. Use the smallest integer type that holds the appropriate range of values, typically TINYINT for MONTH and DAY, and SMALLINT for YEAR. Use the EXTRACT() function to pull out individual date and time fields from a TIMESTAMP value, and CAST() the return value to the appropriate integer type.

Choose an appropriate Parquet block size

By default, the Impala INSERT ... SELECT statement creates Parquet files with a 256 MB block size. (This default was changed in Impala 2.0. Formerly, the limit was 1 GB, but Impala made conservative estimates about compression, resulting in files that were smaller than 1 GB.)

Each Parquet file written by Impala is a single block, allowing the whole file to be processed as a unit by a single host. As you copy Parquet files into HDFS or between HDFS filesystems, use hdfs dfs -pb to preserve the original block size.

If there is only one or a few data block in your Parquet table, or in a partition that is the only one accessed by a query, then you might experience a slowdown for a different reason: not enough data to take advantage of Impala's parallel distributed queries. Each data block is processed by a single core on one of the DataNodes. In a 100-node cluster of 16-core machines, you could potentially process thousands of data files simultaneously. You want to find a sweet spot between "many tiny files" and "single giant file" that balances bulk I/O and parallel processing. You can set the PARQUET_FILE_SIZE query option before doing an INSERT ... SELECT statement to reduce the size of each generated Parquet file. (Specify the file size as an absolute number of bytes, or in Impala 2.0 and later, in units ending with m for megabytes or g for gigabytes.) Run benchmarks with different file sizes to find the right balance point for your particular data volume.

Gather statistics for all tables used in performance-critical or high-volume join queries

Gather the statistics with the COMPUTE STATS statement. See Performance Considerations for Join Queries on page 568 for details.

Minimize the overhead of transmitting results back to the client

Use techniques such as:

- Aggregation. If you need to know how many rows match a condition, the total values of matching values from some column, the lowest or highest matching value, and so on, call aggregate functions such as COUNT(), SUM(), and MAX() in the query rather than sending the result set to an application and doing those computations there. Remember that the size of an unaggregated result set could be huge, requiring substantial time to transmit across the network.
- Filtering. Use all applicable tests in the WHERE clause of a query to eliminate rows that are not relevant, rather than producing a big result set and filtering it using application logic.
- LIMIT clause. If you only need to see a few sample values from a result set, or the top or bottom values from a query using ORDER BY, include the LIMIT clause to reduce the size of the result set rather than asking for the full result set and then throwing most of the rows away.
- Avoid overhead from pretty-printing the result set and displaying it on the screen. When you retrieve the results through impala-shell, use impala-shell options such as -B and --output_delimiter to produce results without special formatting, and redirect output to a file rather than printing to the screen. Consider using INSERT ... SELECT to write the results directly to new files in HDFS. See impala-shell Configuration Options on page 714 for details about the impala-shell command-line options.

Verify that your queries are planned in an efficient logical manner

Examine the EXPLAIN plan for a query before actually running it. See EXPLAIN Statement on page 271 and Using the EXPLAIN Plan for Performance Tuning on page 602 for details.

Verify performance characteristics of queries

Verify that the low-level aspects of I/O, memory usage, network bandwidth, CPU utilization, and so on are within expected ranges by examining the query profile for a query after running it. See Using the Query Profile for Performance Tuning on page 604 for details.

Use appropriate operating system settings

See Optimizing Performance in CDH for recommendations about operating system settings that you can change to influence Impala performance. In particular, you might find that changing the `vm.swappiness` Linux kernel setting to a non-zero value improves overall performance.

Hotspot analysis

In the context of Impala, a hotspot is defined as “an Impala daemon that for a single query or a workload is spending a far greater amount of time processing data relative to its neighbours”.

Before discussing the options to tackle this issue some background is first required to understand how this problem can occur.

By default, the scheduling of scan based plan fragments is deterministic. This means that for multiple queries needing to read the same block of data, the same node will be picked to host the scan. The default scheduling logic does not take into account node workload from prior queries. The complexity of materializing a tuple depends on a few factors, namely: decoding and decompression. If the tuples are densely packed into data pages due to good encoding/compression ratios, there will be more work required when reconstructing the data. Each compression codec offers different performance tradeoffs and should be considered before writing the data. Due to the deterministic nature of the scheduler, single nodes can become bottlenecks for highly concurrent queries that use the same tables.

If, for example, a Parquet based dataset is tiny, e.g. a small dimension table, such that it fits into a single HDFS block (Impala by default will create 256 MB blocks when Parquet is used, each containing a single row group) then there are a number of options that can be considered to resolve the potential scheduling hotspots when querying this data:

- In CDH 5.7 and higher, the scheduler’s deterministic behaviour can be changed using the following query options: `REPLICA_PREFERENCE` and `RANDOM_REPLICA`. For a detailed description of each of these modes see IMPALA-2696.
- HDFS caching can be used to cache block replicas. This will cause the Impala scheduler to randomly pick (from CDH 5.4 and higher) a node that is hosting a cached block replica for the scan. Note, although HDFS caching has benefits, it serves only to help with the reading of raw block data and not cached tuple data, but with the right number of cached replicas (by default, HDFS only caches one replica), even load distribution can be achieved for smaller datasets.
- Do not compress the table data. The uncompressed table data spans more nodes and eliminates skew caused by compression.
- Reduce the Parquet file size via the `PARQUET_FILE_SIZE` query option when writing the table data. Using this approach the data will span more nodes. However it’s not recommended to drop the size below 32 MB.

Performance Considerations for Join Queries

Queries involving join operations often require more tuning than queries that refer to only one table. The maximum size of the result set from a join query is the product of the number of rows in all the joined tables. When joining several tables with millions or billions of rows, any missed opportunity to filter the result set, or other inefficiency in the query, could lead to an operation that does not finish in a practical time and has to be cancelled.

The simplest technique for tuning an Impala join query is to collect statistics on each table involved in the join using the `COMPUTE STATS` statement, and then let Impala automatically optimize the query based on the size of each table, number of distinct values of each column, and so on. The `COMPUTE STATS` statement and the join optimization are new features introduced in Impala 1.2.2. For accurate statistics about each table, issue the `COMPUTE STATS` statement after loading the data into that table, and again if the amount of data changes substantially due to an `INSERT`, `LOAD DATA`, adding a partition, and so on.

If statistics are not available for all the tables in the join query, or if Impala chooses a join order that is not the most efficient, you can override the automatic join order optimization by specifying the STRAIGHT_JOIN keyword immediately after the SELECT and any DISTINCT or ALL keywords. In this case, Impala uses the order the tables appear in the query to guide how the joins are processed.

When you use the STRAIGHT_JOIN technique, you must order the tables in the join query manually instead of relying on the Impala optimizer. The optimizer uses sophisticated techniques to estimate the size of the result set at each stage of the join. For manual ordering, use this heuristic approach to start with, and then experiment to fine-tune the order:

- Specify the largest table first. This table is read from disk by each Impala node and so its size is not significant in terms of memory usage during the query.
- Next, specify the smallest table. The contents of the second, third, and so on tables are all transmitted across the network. You want to minimize the size of the result set from each subsequent stage of the join query. The most likely approach involves joining a small table first, so that the result set remains small even as subsequent larger tables are processed.
- Join the next smallest table, then the next smallest, and so on.

For example, if you had tables BIG, MEDIUM, SMALL, and TINY, the logical join order to try would be BIG, TINY, SMALL, MEDIUM.

The terms “largest” and “smallest” refers to the size of the intermediate result set based on the number of rows and columns from each table that are part of the result set. For example, if you join one table sales with another table customers, a query might find results from 100 different customers who made a total of 5000 purchases. In that case, you would specify SELECT ... FROM sales JOIN customers ..., putting customers on the right side because it is smaller in the context of this query.

The Impala query planner chooses between different techniques for performing join queries, depending on the absolute and relative sizes of the tables. **Broadcast joins** are the default, where the right-hand table is considered to be smaller than the left-hand table, and its contents are sent to all the other nodes involved in the query. The alternative technique is known as a **partitioned join** (not related to a partitioned table), which is more suitable for large tables of roughly equal size. With this technique, portions of each table are sent to appropriate other nodes where those subsets of rows can be processed in parallel. The choice of broadcast or partitioned join also depends on statistics being available for all tables in the join, gathered by the COMPUTE STATS statement.

To see which join strategy is used for a particular query, issue an EXPLAIN statement for the query. If you find that a query uses a broadcast join when you know through benchmarking that a partitioned join would be more efficient, or vice versa, add a hint to the query to specify the precise join mechanism to use. See Optimizer Hints in Impala on page 387 for details.

How Joins Are Processed when Statistics Are Unavailable

If table or column statistics are not available for some tables in a join, Impala still reorders the tables using the information that is available. Tables with statistics are placed on the left side of the join order, in descending order of cost based on overall size and cardinality. Tables without statistics are treated as zero-size, that is, they are always placed on the right side of the join order.

Overriding Join Reordering with STRAIGHT_JOIN

If an Impala join query is inefficient because of outdated statistics or unexpected data distribution, you can keep Impala from reordering the joined tables by using the STRAIGHT_JOIN keyword immediately after the SELECT and any DISTINCT or ALL keywords. The STRAIGHT_JOIN keyword turns off the reordering of join clauses that Impala does internally, and produces a plan that relies on the join clauses being ordered optimally in the query text.



Note:

The STRAIGHT_JOIN hint affects the join order of table references in the query block containing the hint. It does not affect the join order of nested queries, such as views, inline views, or WHERE-clause subqueries. To use this hint for performance tuning of complex queries, apply the hint to all query blocks that need a fixed join order.

In this example, the subselect from the BIG table produces a very small result set, but the table might still be treated as if it were the biggest and placed first in the join order. Using STRAIGHT_JOIN for the last join clause prevents the final table from being reordered, keeping it as the rightmost table in the join order.

```
select straight_join x from medium join small join (select * from big where c1 < 10) as
 big
  where medium.id = small.id and small.id = big.id;

-- If the query contains [DISTINCT | ALL], the hint goes after those keywords.
select distinct straight_join x from medium join small join (select * from big where c1
 < 10) as big
  where medium.id = small.id and small.id = big.id;
```

Examples of Join Order Optimization

Here are examples showing joins between tables with 1 billion, 200 million, and 1 million rows. (In this case, the tables are unpartitioned and using Parquet format.) The smaller tables contain subsets of data from the largest one, for convenience of joining on the unique ID column. The smallest table only contains a subset of columns from the others.

```
[localhost:21000] > create table big stored as parquet as select * from raw_data;
+----------------------------+
| summary                    |
+----------------------------+
| Inserted 1000000000 row(s) |
+----------------------------+
Returned 1 row(s) in 671.56s
[localhost:21000] > desc big;
+-----------+---------+---------+
| name      | type    | comment |
+-----------+---------+---------+
| id        | int     |         |
| val       | int     |         |
| zfill     | string  |         |
| name      | string  |         |
| assertion | boolean |         |
+-----------+---------+---------+
Returned 5 row(s) in 0.01s
[localhost:21000] > create table medium stored as parquet as select * from big limit
200 * floor(1e6);
+---------------------------+
| summary                   |
+---------------------------+
| Inserted 200000000 row(s) |
+---------------------------+
Returned 1 row(s) in 138.31s
[localhost:21000] > create table small stored as parquet as select id,val,name from big
 where assertion = true limit 1 * floor(1e6);
+-------------------------+
| summary                 |
+-------------------------+
| Inserted 1000000 row(s) |
+-------------------------+
Returned 1 row(s) in 6.32s
```

For any kind of performance experimentation, use the EXPLAIN statement to see how any expensive query will be performed without actually running it, and enable verbose EXPLAIN plans containing more performance-oriented detail: The most interesting plan lines are highlighted in bold, showing that without statistics for the joined tables,

Impala cannot make a good estimate of the number of rows involved at each stage of processing, and is likely to stick with the BROADCAST join mechanism that sends a complete copy of one of the tables to each node.

```
[localhost:21000] > set explain_level=verbose;
EXPLAIN_LEVEL set to verbose
[localhost:21000] > explain select count(*) from big join medium where big.id = medium.id;
+------------------------------------------------------------+
| Explain String                                             |
+------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=2.10GB VCores=2    |
|                                                            |
| PLAN FRAGMENT 0                                            |
|   PARTITION: UNPARTITIONED                                 |
|                                                            |
|   6:AGGREGATE (merge finalize)                             |
|   |  output: SUM(COUNT(*))                                 |
|   |  cardinality: 1                                        |
|   |  per-host memory: unavailable                          |
|   |  tuple ids: 2                                          |
|   |                                                        |
|   5:EXCHANGE                                               |
|      cardinality: 1                                        |
|      per-host memory: unavailable                          |
|      tuple ids: 2                                          |
|                                                            |
| PLAN FRAGMENT 1                                            |
|   PARTITION: RANDOM                                        |
|                                                            |
|   STREAM DATA SINK                                         |
|     EXCHANGE ID: 5                                         |
|     UNPARTITIONED                                          |
|                                                            |
|   3:AGGREGATE                                              |
|   |  output: COUNT(*)                                      |
|   |  cardinality: 1                                        |
|   |  per-host memory: 10.00MB                              |
|   |  tuple ids: 2                                          |
|   |                                                        |
|   2:HASH JOIN                                              |
|   |  join op: INNER JOIN (BROADCAST)                       |
|   |  hash predicates:                                      |
|   |    big.id = medium.id                                  |
|   |  cardinality: unavailable                              |
|   |  per-host memory: 2.00GB                               |
|   |  tuple ids: 0 1                                        |
|   |                                                        |
|   |----4:EXCHANGE                                          |
|   |       cardinality: unavailable                         |
|   |       per-host memory: 0B                              |
|   |       tuple ids: 1                                     |
|   |                                                        |
|   0:SCAN HDFS                                              |
|      table=join_order.big #partitions=1/1 size=23.12GB     |
|      table stats: unavailable                              |
|      column stats: unavailable                             |
|      cardinality: unavailable                              |
|      per-host memory: 88.00MB                              |
|      tuple ids: 0                                          |
|                                                            |
| PLAN FRAGMENT 2                                            |
|   PARTITION: RANDOM                                        |
|                                                            |
|   STREAM DATA SINK                                         |
|     EXCHANGE ID: 4                                         |
|     UNPARTITIONED                                          |
|                                                            |
|   1:SCAN HDFS                                              |
|      table=join_order.medium #partitions=1/1 size=4.62GB   |
|      table stats: unavailable                              |
|      column stats: unavailable                             |
|      cardinality: unavailable                              |
|      per-host memory: 88.00MB                              |
|      tuple ids: 1                                          |
```

```
+-----------------------------------------------------------+
Returned 64 row(s) in 0.04s
```

Gathering statistics for all the tables is straightforward, one COMPUTE STATS statement per table:

```
[localhost:21000] > compute stats small;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 3 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 4.26s
[localhost:21000] > compute stats medium;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 5 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 42.11s
[localhost:21000] > compute stats big;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 5 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 165.44s
```

With statistics in place, Impala can choose a more effective join order rather than following the left-to-right sequence of tables in the query, and can choose BROADCAST or PARTITIONED join strategies based on the overall sizes and number of rows in the table:

```
[localhost:21000] > explain select count(*) from medium join big where big.id = medium.id;
Query: explain select count(*) from medium join big where big.id = medium.id
+-----------------------------------------------------------+
| Explain String                                            |
+-----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=937.23MB VCores=2 |
|                                                           |
| PLAN FRAGMENT 0                                           |
|   PARTITION: UNPARTITIONED                                |
|                                                           |
|   6:AGGREGATE (merge finalize)                            |
|   |  output: SUM(COUNT(*))                                |
|   |  cardinality: 1                                       |
|   |  per-host memory: unavailable                         |
|   |  tuple ids: 2                                         |
|   |                                                       |
|   5:EXCHANGE                                              |
|      cardinality: 1                                       |
|      per-host memory: unavailable                         |
|      tuple ids: 2                                         |
|                                                           |
| PLAN FRAGMENT 1                                           |
|   PARTITION: RANDOM                                       |
|                                                           |
|   STREAM DATA SINK                                        |
|     EXCHANGE ID: 5                                        |
|     UNPARTITIONED                                         |
|                                                           |
|   3:AGGREGATE                                             |
|   |  output: COUNT(*)                                     |
|   |  cardinality: 1                                       |
|   |  per-host memory: 10.00MB                             |
|   |  tuple ids: 2                                         |
|   |                                                       |
|   2:HASH JOIN                                             |
|   |  join op: INNER JOIN (BROADCAST)                      |
|   |  hash predicates:                                     |
|   |    big.id = medium.id                                 |
|   |  cardinality: 1443004441                              |
|   |  per-host memory: 839.23MB                            |
```

```
|    |  tuple ids: 1 0                                                |
|    |                                                                |
|    |----4:EXCHANGE                                                  |
|    |       cardinality: 200000000                                   |
|    |       per-host memory: 0B                                      |
|    |       tuple ids: 0                                             |
|    |                                                                |
|    1:SCAN HDFS                                                      |
|       table=join_order.big #partitions=1/1 size=23.12GB             |
|       table stats: 1000000000 rows total                            |
|       column stats: all                                             |
|       cardinality: 1000000000                                       |
|       per-host memory: 88.00MB                                      |
|       tuple ids: 1                                                  |
|                                                                     |
| PLAN FRAGMENT 2                                                     |
|   PARTITION: RANDOM                                                 |
|                                                                     |
|   STREAM DATA SINK                                                  |
|     EXCHANGE ID: 4                                                  |
|     UNPARTITIONED                                                   |
|                                                                     |
|   0:SCAN HDFS                                                       |
|      table=join_order.medium #partitions=1/1 size=4.62GB            |
|      table stats: 200000000 rows total                              |
|      column stats: all                                              |
|      cardinality: 200000000                                         |
|      per-host memory: 88.00MB                                       |
|      tuple ids: 0                                                   |
+---------------------------------------------------------------------+
Returned 64 row(s) in 0.04s

[localhost:21000] > explain select count(*) from small join big where big.id = small.id;
Query: explain select count(*) from small join big where big.id = small.id
+---------------------------------------------------------------------+
| Explain String                                                      |
+---------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=101.15MB VCores=2           |
|                                                                     |
| PLAN FRAGMENT 0                                                     |
|   PARTITION: UNPARTITIONED                                          |
|                                                                     |
|   6:AGGREGATE (merge finalize)                                      |
|   |  output: SUM(COUNT(*))                                          |
|   |  cardinality: 1                                                 |
|   |  per-host memory: unavailable                                   |
|   |  tuple ids: 2                                                   |
|   |                                                                 |
|   5:EXCHANGE                                                        |
|      cardinality: 1                                                 |
|      per-host memory: unavailable                                   |
|      tuple ids: 2                                                   |
|                                                                     |
| PLAN FRAGMENT 1                                                     |
|   PARTITION: RANDOM                                                 |
|                                                                     |
|   STREAM DATA SINK                                                  |
|     EXCHANGE ID: 5                                                  |
|     UNPARTITIONED                                                   |
|                                                                     |
|   3:AGGREGATE                                                       |
|   |  output: COUNT(*)                                               |
|   |  cardinality: 1                                                 |
|   |  per-host memory: 10.00MB                                       |
|   |  tuple ids: 2                                                   |
|   |                                                                 |
|   2:HASH JOIN                                                       |
|   |  join op: INNER JOIN (BROADCAST)                                |
|   |  hash predicates:                                               |
|   |    big.id = small.id                                            |
|   |  cardinality: 1000000000                                        |
|   |  per-host memory: 3.15MB                                        |
|   |  tuple ids: 1 0                                                 |
```

```
|   |                                                          |
|   |----4:EXCHANGE                                            |
|   |       cardinality: 1000000                               |
|   |       per-host memory: 0B                                |
|   |       tuple ids: 0                                       |
|   |                                                          |
|   1:SCAN HDFS                                                |
|      table=join_order.big #partitions=1/1 size=23.12GB       |
|      table stats: 1000000000 rows total                      |
|      column stats: all                                       |
|      cardinality: 1000000000                                 |
|      per-host memory: 88.00MB                                |
|      tuple ids: 1                                            |
|                                                              |
| PLAN FRAGMENT 2                                              |
|   PARTITION: RANDOM                                          |
|                                                              |
|   STREAM DATA SINK                                           |
|     EXCHANGE ID: 4                                           |
|     UNPARTITIONED                                            |
|                                                              |
|   0:SCAN HDFS                                                |
|      table=join_order.small #partitions=1/1 size=17.93MB     |
|      table stats: 1000000 rows total                         |
|      column stats: all                                       |
|      cardinality: 1000000                                    |
|      per-host memory: 32.00MB                                |
|      tuple ids: 0                                            |
+--------------------------------------------------------------+
Returned 64 row(s) in 0.03s
```

When queries like these are actually run, the execution times are relatively consistent regardless of the table order in the query text. Here are examples using both the unique `ID` column and the `VAL` column containing duplicate values:

```
[localhost:21000] > select count(*) from big join small on (big.id = small.id);
Query: select count(*) from big join small on (big.id = small.id)
+----------+
| count(*) |
+----------+
| 1000000  |
+----------+
Returned 1 row(s) in 21.68s
[localhost:21000] > select count(*) from small join big on (big.id = small.id);
Query: select count(*) from small join big on (big.id = small.id)
+----------+
| count(*) |
+----------+
| 1000000  |
+----------+
Returned 1 row(s) in 20.45s

[localhost:21000] > select count(*) from big join small on (big.val = small.val);
+------------+
| count(*)   |
+------------+
| 2000948962 |
+------------+
Returned 1 row(s) in 108.85s
[localhost:21000] > select count(*) from small join big on (big.val = small.val);
+------------+
| count(*)   |
+------------+
| 2000948962 |
+------------+
Returned 1 row(s) in 100.76s
```



Note: When examining the performance of join queries and the effectiveness of the join order optimization, make sure the query involves enough data and cluster resources to see a difference depending on the query plan. For example, a single data file of just a few megabytes will reside in a single HDFS block and be processed on a single node. Likewise, if you use a single-node or two-node cluster, there might not be much difference in efficiency for the broadcast or partitioned join strategies.

Table and Column Statistics

Impala can do better optimization for complex or multi-table queries when it has access to statistics about the volume of data and how the values are distributed. Impala uses this information to help parallelize and distribute the work for a query. For example, optimizing join queries requires a way of determining if one table is “bigger” than another, which is a function of the number of rows and the average row size for each table. The following sections describe the categories of statistics Impala can work with, and how to produce them and keep them up to date.

Overview of Table Statistics

The Impala query planner can make use of statistics about entire tables and partitions. This information includes physical characteristics such as the number of rows, number of data files, the total size of the data files, and the file format. For partitioned tables, the numbers are calculated per partition, and as totals for the whole table. This metadata is stored in the metastore database, and can be updated by either Impala or Hive. If a number is not available, the value -1 is used as a placeholder. Some numbers, such as number and total sizes of data files, are always kept up to date because they can be calculated cheaply, as part of gathering HDFS block metadata.

The following example shows table stats for an unpartitioned Parquet table. The values for the number and sizes of files are always available. Initially, the number of rows is not known, because it requires a potentially expensive scan through the entire table, and so that value is displayed as -1. The `COMPUTE STATS` statement fills in any unknown table stats values.

```
show table stats parquet_snappy;
+-------+--------+---------+--------------+------------------+---------+------------------+...
| #Rows | #Files | Size    | Bytes Cached | Cache Replication | Format  | Incremental
stats |...
+-------+--------+---------+--------------+------------------+---------+------------------+...
| -1    | 96     | 23.35GB | NOT CACHED   | NOT CACHED        | PARQUET | false
      |...
+-------+--------+---------+--------------+------------------+---------+------------------+...

compute stats parquet_snappy;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 6 column(s). |
+-----------------------------------------+


show table stats parquet_snappy;
+-----------+--------+---------+--------------+------------------+---------+------------------+...
| #Rows     | #Files | Size    | Bytes Cached | Cache Replication | Format  | Incremental
 stats |...
+-----------+--------+---------+--------------+------------------+---------+------------------+...
| 1000000000 | 96     | 23.35GB | NOT CACHED   | NOT CACHED        | PARQUET | false
           |...
+-----------+--------+---------+--------------+------------------+---------+------------------+...
```

Impala performs some optimizations using this metadata on its own, and other optimizations by using a combination of table and column statistics.

To check that table statistics are available for a table, and see the details of those statistics, use the statement `SHOW TABLE STATS` *`table_name`*. See SHOW Statement on page 363 for details.

If you use the Hive-based methods of gathering statistics, see the Hive wiki for information about the required configuration on the Hive side. Where practical, use the Impala COMPUTE STATS statement to avoid potential configuration and scalability issues with the statistics-gathering process.

If you run the Hive statement ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS, Impala can only use the resulting column statistics if the table is unpartitioned. Impala cannot use Hive-generated column statistics for a partitioned table.

Overview of Column Statistics

The Impala query planner can make use of statistics about individual columns when that metadata is available in the metastore database. This technique is most valuable for columns compared across tables in join queries, to help estimate how many rows the query will retrieve from each table. These statistics are also important for correlated subqueries using the EXISTS( ) or IN( ) operators, which are processed internally the same way as join queries.

The following example shows column stats for an unpartitioned Parquet table. The values for the maximum and average sizes of some types are always available, because those figures are constant for numeric and other fixed-size types. Initially, the number of distinct values is not known, because it requires a potentially expensive scan through the entire table, and so that value is displayed as -1. The same applies to maximum and average sizes of variable-sized types, such as STRING. The COMPUTE STATS statement fills in most unknown column stats values. (It does not record the number of NULL values, because currently Impala does not use that figure for query optimization.)

```
show column stats parquet_snappy;
+-------------+----------+------------------+--------+----------+----------+
| Column      | Type     | #Distinct Values | #Nulls | Max Size | Avg Size |
+-------------+----------+------------------+--------+----------+----------+
| id          | BIGINT   | -1               | -1     | 8        | 8        |
| val         | INT      | -1               | -1     | 4        | 4        |
| zerofill    | STRING   | -1               | -1     | -1       | -1       |
| name        | STRING   | -1               | -1     | -1       | -1       |
| assertion   | BOOLEAN  | -1               | -1     | 1        | 1        |
| location_id | SMALLINT | -1               | -1     | 2        | 2        |
+-------------+----------+------------------+--------+----------+----------+

compute stats parquet_snappy;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 6 column(s). |
+-----------------------------------------+

show column stats parquet_snappy;
+-------------+----------+------------------+--------+----------+-------------------+
| Column      | Type     | #Distinct Values | #Nulls | Max Size | Avg Size          |
+-------------+----------+------------------+--------+----------+-------------------+
| id          | BIGINT   | 183861280        | -1     | 8        | 8                 |
| val         | INT      | 139017           | -1     | 4        | 4                 |
| zerofill    | STRING   | 101761           | -1     | 6        | 6                 |
| name        | STRING   | 145636240        | -1     | 22       | 13.00020027160645 |
| assertion   | BOOLEAN  | 2                | -1     | 1        | 1                 |
| location_id | SMALLINT | 339              | -1     | 2        | 2                 |
+-------------+----------+------------------+--------+----------+-------------------+
```

Note:

For column statistics to be effective in Impala, you also need to have table statistics for the applicable tables, as described in Overview of Table Statistics on page 575. When you use the Impala COMPUTE STATS statement, both table and column statistics are automatically gathered at the same time, for all columns in the table.



Note: Prior to Impala 1.4.0, COMPUTE STATS counted the number of NULL values in each column and recorded that figure in the metastore database. Because Impala does not currently use the NULL count during query planning, Impala 1.4.0 and higher speeds up the COMPUTE STATS statement by skipping this NULL counting.

To check whether column statistics are available for a particular set of columns, use the SHOW COLUMN STATS *table_name* statement, or check the extended EXPLAIN output for a query against that table that refers to those columns. See SHOW Statement on page 363 and EXPLAIN Statement on page 271 for details.

If you run the Hive statement ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS, Impala can only use the resulting column statistics if the table is unpartitioned. Impala cannot use Hive-generated column statistics for a partitioned table.

How Table and Column Statistics Work for Partitioned Tables

When you use Impala for "big data", you are highly likely to use partitioning for your biggest tables, the ones representing data that can be logically divided based on dates, geographic regions, or similar criteria. The table and column statistics are especially useful for optimizing queries on such tables. For example, a query involving one year might involve substantially more or less data than a query involving a different year, or a range of several years. Each query might be optimized differently as a result.

The following examples show how table and column stats work with a partitioned table. The table for this example is partitioned by year, month, and day. For simplicity, the sample data consists of 5 partitions, all from the same year and month. Table stats are collected independently for each partition. (In fact, the SHOW PARTITIONS statement displays exactly the same information as SHOW TABLE STATS for a partitioned table.) Column stats apply to the entire table, not to individual partitions. Because the partition key column values are represented as HDFS directories, their characteristics are typically known in advance, even when the values for non-key columns are shown as -1.

```
show partitions year_month_day;
+-------+-------+-----+-------+--------+---------+-------------+------------------+---------+...
| year  | month | day | #Rows | #Files | Size    | Bytes Cached | Cache Replication |
Format  |...
+-------+-------+-----+-------+--------+---------+-------------+------------------+---------+...
| 2013  | 12    | 1   | -1    | 1      | 2.51MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 2   | -1    | 1      | 2.53MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 3   | -1    | 1      | 2.52MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 4   | -1    | 1      | 2.51MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 5   | -1    | 1      | 2.52MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| Total |       |     | -1    | 5      | 12.58MB | 0B           |                   |
        |...
+-------+-------+-----+-------+--------+---------+-------------+------------------+---------+...

show table stats year_month_day;
+-------+-------+-----+-------+--------+---------+-------------+------------------+---------+...
| year  | month | day | #Rows | #Files | Size    | Bytes Cached | Cache Replication |
Format  |...
+-------+-------+-----+-------+--------+---------+-------------+------------------+---------+...
| 2013  | 12    | 1   | -1    | 1      | 2.51MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 2   | -1    | 1      | 2.53MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 3   | -1    | 1      | 2.52MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 4   | -1    | 1      | 2.51MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| 2013  | 12    | 5   | -1    | 1      | 2.52MB  | NOT CACHED   | NOT CACHED        |
PARQUET |...
| Total |       |     | -1    | 5      | 12.58MB | 0B           |                   |
        |...
```

```
+-------+-------+-----+-------+--------+---------+-------------+------------------+---------+...

show column stats year_month_day;
+-----------+---------+------------------+--------+----------+----------+
| Column    | Type    | #Distinct Values | #Nulls | Max Size | Avg Size |
+-----------+---------+------------------+--------+----------+----------+
| id        | INT     | -1               | -1     | 4        | 4        |
| val       | INT     | -1               | -1     | 4        | 4        |
| zfill     | STRING  | -1               | -1     | -1       | -1       |
| name      | STRING  | -1               | -1     | -1       | -1       |
| assertion | BOOLEAN | -1               | -1     | 1        | 1        |
| year      | INT     | 1                | 0      | 4        | 4        |
| month     | INT     | 1                | 0      | 4        | 4        |
| day       | INT     | 5                | 0      | 4        | 4        |
+-----------+---------+------------------+--------+----------+----------+

compute stats year_month_day;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 5 partition(s) and 5 column(s). |
+-----------------------------------------+

show table stats year_month_day;
+-------+-------+-----+--------+--------+---------+-------------+------------------+---------+...
| year  | month | day | #Rows  | #Files | Size    | Bytes Cached | Cache Replication |
 Format  |...
+-------+-------+-----+--------+--------+---------+-------------+------------------+---------+...
| 2013  | 12    | 1   | 93606  | 1      | 2.51MB  | NOT CACHED   | NOT CACHED        |
 PARQUET |...
| 2013  | 12    | 2   | 94158  | 1      | 2.53MB  | NOT CACHED   | NOT CACHED        |
 PARQUET |...
| 2013  | 12    | 3   | 94122  | 1      | 2.52MB  | NOT CACHED   | NOT CACHED        |
 PARQUET |...
| 2013  | 12    | 4   | 93559  | 1      | 2.51MB  | NOT CACHED   | NOT CACHED        |
 PARQUET |...
| 2013  | 12    | 5   | 93845  | 1      | 2.52MB  | NOT CACHED   | NOT CACHED        |
 PARQUET |...
| Total |       |     | 469290 | 5      | 12.58MB | 0B           |                   |
        |...
+-------+-------+-----+--------+--------+---------+-------------+------------------+---------+...

show column stats year_month_day;
+-----------+---------+------------------+--------+----------+--------------------+
| Column    | Type    | #Distinct Values | #Nulls | Max Size | Avg Size           |
+-----------+---------+------------------+--------+----------+--------------------+
| id        | INT     | 511129           | -1     | 4        | 4                  |
| val       | INT     | 364853           | -1     | 4        | 4                  |
| zfill     | STRING  | 311430           | -1     | 6        | 6                  |
| name      | STRING  | 471975           | -1     | 22       | 13.00160026550293  |
| assertion | BOOLEAN | 2                | -1     | 1        | 1                  |
| year      | INT     | 1                | 0      | 4        | 4                  |
| month     | INT     | 1                | 0      | 4        | 4                  |
| day       | INT     | 5                | 0      | 4        | 4                  |
+-----------+---------+------------------+--------+----------+--------------------+
```

If you run the Hive statement `ANALYZE TABLE COMPUTE STATISTICS FOR COLUMNS`, Impala can only use the resulting column statistics if the table is unpartitioned. Impala cannot use Hive-generated column statistics for a partitioned table.

Generating Table and Column Statistics

Use the `COMPUTE STATS` family of commands to collect table and column statistics. The `COMPUTE STATS` variants offer different tradeoffs between computation cost, staleness, and maintenance workflows which are explained below.



Important:

For a particular table, use either `COMPUTE STATS` or `COMPUTE INCREMENTAL STATS`. The two kinds of stats do not interoperate with each other at the table level. Without dropping the stats, if you run `COMPUTE INCREMENTAL STATS` it will overwrite the full compute stats or if you run `COMPUTE STATS` it will drop all incremental stats for consistency.

COMPUTE STATS

The `COMPUTE STATS` command collects and sets the table-level and partition-level row counts as well as all column statistics for a given table. The collection process is CPU-intensive and can take a long time to complete for very large tables.

To speed up `COMPUTE STATS` consider the following options which can be combined.

- Limit the number of columns for which statistics are collected to increase the efficiency of COMPUTE STATS. Queries benefit from statistics for those columns involved in filters, join conditions, group by or partition by clauses. Other columns are good candidates to exclude from COMPUTE STATS. This feature is available since Impala 2.12.
- Set the MT_DOP query option to use more threads within each participating impalad to compute the statistics faster - but not more efficiently. Note that computing stats on a large table with a high MT_DOP value can negatively affect other queries running at the same time if the COMPUTE STATS claims most CPU cycles. This feature is available since Impala 2.8.
- Consider the experimental extrapolation and sampling features (see below) to further increase the efficiency of computing stats.

`COMPUTE STATS` is intended to be run periodically, e.g. weekly, or on-demand when the contents of a table have changed significantly. Due to the high resource utilization and long repsonse time of t`COMPUTE STATS`, it is most practical to run it in a scheduled maintnance window where the Impala cluster is idle enough to accommodate the expensive operation. The degree of change that qualifies as “significant” depends on the query workload, but typically, if 30% of the rows have changed then it is recommended to recompute statistics.

If you reload a complete new set of data for a table, but the number of rows and number of distinct values for each column is relatively unchanged from before, you do not need to recompute stats for the table.

Experimental: Extrapolation and Sampling

Impala 2.12 and higher includes two experimental features to alleviate common issues for computing and maintaining statistics on very large tables. The following shortcomings are improved upon:

- Newly added partitions do not have row count statistics. Table scans that only access those new partitions are treated as not having stats. Similarly, table scans that access both new and old partitions estimate the scan cardinality based on those old partitions that have stats, and the new partitions without stats are treated as having 0 rows.
- The row counts of existing partitions become stale when data is added or dropped.
- Computing stats for tables with a 100,000 or more partitions might fail or be very slow due to the high cost of updating the partition metadata in the Hive Metastore.
- With transient compute resources it is important to minimize the time from starting a new cluster to successfully running queries. Since the cluster might be relatively short-lived, users might prefer to quickly collect stats that are "good enough" as opposed to spending a lot of time and resouces on computing full-fidelity stats.

For very large tables, it is often wasteful or impractical to run a full COMPUTE STATS to address the scenarios above on a frequent basis.

The sampling feature makes COMPUTE STATS more efficient by processing a fraction of the table data, and the extrapolation feature aims to reduce the frequency at which COMPUTE STATS needs to be re-run by estimating the row count of new and modified partitions.

The sampling and extrapolation features are disabled by default. They can be enabled globally or for specific tables, as follows. Set the impalad start-up configuration "--enable_stats_extrapolation" to enable the features globally. To enable them only for a specific table, set the "impala.enable.stats.extrapolation" table property to "true" for the desired table. The table-level property overrides the global setting, so it is also possible to enable sampling and extrapolation globally, but disable it for specific tables by setting the table property to "false". Example: ALTER TABLE mytable test_table SET TBLPROPERTIES("impala.enable.stats.extrapolation"="true")



Note: Why are these features experimental? Due to their probabilistic nature it is possible that these features perform pathologically poorly on tables with extreme data/file/size distributions. Since it is not feasible for us to test all possible scenarios we only cautiously advertise these new capabilities. That said, the features have been thoroughly tested and are considered functionally stable. If you decide to give these features a try, please tell us about your experience at user@impala.apache.org! We rely on user feedback to guide future inprovements in statistics collection.

Stats Extrapolation

The main idea of stats extrapolation is to estimate the row count of new and modified partitions based on the result of the last COMPUTE STATS. Enabling stats extrapolation changes the behavior of COMPUTE STATS, as well as the cardinality estimation of table scans. COMPUTE STATS no longer computes and stores per-partition row counts, and instead, only computes a table-level row count together with the total number of file bytes in the table at that time. No partition metadata is modified. The input cardinality of a table scan is estimated by converting the data volume of relevant partitions to a row count, based on the table-level row count and file bytes statistics. It is assumed that within the same table, different sets of files with the same data volume correspond to the similar number of rows on average. With extrapolation enabled, the scan cardinality estimation ignores per-partition row counts. It only relies on the table-level statistics and the scanned data volume.

The SHOW TABLE STATS and EXPLAIN commands distinguish between row counts stored in the Hive Metastore, and the row counts extrapolated based on the above process. Consult the SHOW TABLE STATS and EXPLAIN documentation for more details.

Sampling

A TABLESAMPLE clause may be added to COMPUTE STATS to limit the percentage of data to be processed. The final statistics are obtained by extrapolating the statistics from the data sample over the entire table. The extrapolated statistics are stored in the Hive Metastore, just as if no sampling was used. The following example runs COMPUTE STATS over a 10 percent data sample: COMPUTE STATS test_table TABLESAMPLE SYSTEM(10)

We have found that a 10 percent sampling rate typically offers a good tradeoff between statistics accuracy and execution cost. A sampling rate well below 10 percent has shown poor results and is not recommended.



Important: Sampling-based techniques sacrifice result accuracy for execution efficiency, so your mileage may vary for different tables and columns depending on their data distribution. The extrapolation procedure Impala uses for estimating the number of distinct values per column is inherently non-detetministic, so your results may even vary between runs of COMPUTE STATS TABLESAMPLE, even if no data has changed.

COMPUTE INCREMENTAL STATS

In Impala 2.1.0 and higher, you can use the `COMPUTE INCREMENTAL STATS` and `DROP INCREMENTAL STATS` commands. The `INCREMENTAL` clauses work with incremental statistics, a specialized feature for partitioned tables.

When you compute incremental statistics for a partitioned table, by default Impala only processes those partitions that do not yet have incremental statistics. By processing only newly added partitions, you can keep statistics up to date without incurring the overhead of reprocessing the entire table each time.

You can also compute or drop statistics for a specified subset of partitions by including a `PARTITION` clause in the `COMPUTE INCREMENTAL STATS` or `DROP INCREMENTAL STATS` statement.



Important:

In Impala 3.0 and lower, approximately 400 bytes of metadata per column per partition are needed for caching. Tables with a big number of partitions and many columns can add up to a significant memory overhead as the metadata must be cached on the `catalogd` host and on every `impalad` host that is eligible to be a coordinator. If this metadata for all tables exceeds 2 GB, you might experience service downtime. In Impala 3.1 and higher, the issue was alleviated with an improved handling of incremental stats.

When you run `COMPUTE INCREMENTAL STATS` on a table for the first time, the statistics are computed again from scratch regardless of whether the table already has statistics. Therefore, expect a one-time resource-intensive operation for scanning the entire table when running `COMPUTE INCREMENTAL STATS` for the first time on a given table.

The metadata for incremental statistics is handled differently from the original style of statistics:

- Issuing a `COMPUTE INCREMENTAL STATS` without a partition clause causes Impala to compute incremental stats for all partitions that do not already have incremental stats. This might be the entire table when running the command for the first time, but subsequent runs should only update new partitions. You can force updating a partition that already has incremental stats by issuing a `DROP INCREMENTAL STATS` before running `COMPUTE INCREMENTAL STATS`.
- The `SHOW TABLE STATS` and `SHOW PARTITIONS` statements now include an additional column showing whether incremental statistics are available for each column. A partition could already be covered by the original type of statistics based on a prior `COMPUTE STATS` statement, as indicated by a value other than `-1` under the `#Rows` column. Impala query planning uses either kind of statistics when available.
- `COMPUTE INCREMENTAL STATS` takes more time than `COMPUTE STATS` for the same volume of data. Therefore it is most suitable for tables with large data volume where new partitions are added frequently, making it impractical to run a full `COMPUTE STATS` operation for each new partition. For unpartitioned tables, or partitioned tables that are loaded once and not updated with new partitions, use the original `COMPUTE STATS` syntax.
- `COMPUTE INCREMENTAL STATS` uses some memory in the `catalogd` process, proportional to the number of partitions and number of columns in the applicable table. The memory overhead is approximately 400 bytes for each column in each partition. This memory is reserved in the `catalogd` daemon, the `statestored` daemon, and in each instance of the `impalad` daemon.
- In cases where new files are added to an existing partition, issue a `REFRESH` statement for the table, followed by a `DROP INCREMENTAL STATS` and `COMPUTE INCREMENTAL STATS` sequence for the changed partition.
- The `DROP INCREMENTAL STATS` statement operates only on a single partition at a time. To remove statistics (whether incremental or not) from all partitions of a table, issue a `DROP STATS` statement with no `INCREMENTAL` or `PARTITION` clauses.

The following considerations apply to incremental statistics when the structure of an existing table is changed (known as ***schema evolution***):

- If you use an `ALTER TABLE` statement to drop a column, the existing statistics remain valid and `COMPUTE INCREMENTAL STATS` does not rescan any partitions.
- If you use an `ALTER TABLE` statement to add a column, Impala rescans all partitions and fills in the appropriate column-level values the next time you run `COMPUTE INCREMENTAL STATS`.
- If you use an `ALTER TABLE` statement to change the data type of a column, Impala rescans all partitions and fills in the appropriate column-level values the next time you run `COMPUTE INCREMENTAL STATS`.
- If you use an `ALTER TABLE` statement to change the file format of a table, the existing statistics remain valid and a subsequent `COMPUTE INCREMENTAL STATS` does not rescan any partitions.

See COMPUTE STATS Statement on page 219 and DROP STATS Statement on page 265 for syntax details.

Maximum Serialized Stats Size

In Impala 3.0 / CDH 5.15 and lower, when executing `COMPUTE INCREMENTAL STATS` on very large tables, use the configuration setting `--inc_stats_size_limit_bytes` to prevent Impala from running out of memory while updating table metadata. If this limit is reached, Impala will stop loading the table and return an error. The error serves as an indication that `COMPUTE INCREMENTAL STATS` should not be used on the particular table. Consider spitting the table and using regular `COMPUTE STATS`]if possible.

The `--inc_stats_size_limit_bytes` limit is set as a safety check, to prevent Impala from hitting the maximum limit for the table metadata. Note that this limit is only one part of the entire table's metadata all of which together must be below 2 GB.

The default value for `--inc_stats_size_limit_bytes` is 209715200, 200 MB.

To change the `--inc_stats_size_limit_bytes` value, restart impalad and catalogd with the new value specified in bytes, for example, 1048576000 for 1 GB. See Modifying Impala Startup Options for the steps to change the option and restart Impala daemons.

In Impala 3.1 / CDH 5.16 / CDH 6.1 and higher, Impala improved how metadata is updated when executing `COMPUTE INCREMENTAL STATS`, significantly reducing the need for `--inc_stats_size_limit_bytes`.

Detecting Missing Statistics

You can check whether a specific table has statistics using the `SHOW TABLE STATS` statement (for any table) or the `SHOW PARTITIONS` statement (for a partitioned table). Both statements display the same information. If a table or a partition does not have any statistics, the `#Rows` field contains `-1`. Once you compute statistics for the table or partition, the `#Rows` field changes to an accurate value.

The following example shows a table that initially does not have any statistics. The `SHOW TABLE STATS` statement displays different values for `#Rows` before and after the `COMPUTE STATS` operation.

```
[localhost:21000] > create table no_stats (x int);
[localhost:21000] > show table stats no_stats;
+-------+--------+------+--------------+--------+-------------------+
| #Rows | #Files | Size | Bytes Cached | Format | Incremental stats |
+-------+--------+------+--------------+--------+-------------------+
| -1    | 0      | 0B   | NOT CACHED   | TEXT   | false             |
+-------+--------+------+--------------+--------+-------------------+
[localhost:21000] > compute stats no_stats;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
+-----------------------------------------+
[localhost:21000] > show table stats no_stats;
+-------+--------+------+--------------+--------+-------------------+
| #Rows | #Files | Size | Bytes Cached | Format | Incremental stats |
+-------+--------+------+--------------+--------+-------------------+
| 0     | 0      | 0B   | NOT CACHED   | TEXT   | false             |
+-------+--------+------+--------------+--------+-------------------+
```

The following example shows a similar progression with a partitioned table. Initially, `#Rows` is `-1`. After a `COMPUTE STATS` operation, `#Rows` changes to an accurate value. Any newly added partition starts with no statistics, meaning that you must collect statistics after adding a new partition.

```
[localhost:21000] > create table no_stats_partitioned (x int) partitioned by (year
smallint);
[localhost:21000] > show table stats no_stats_partitioned;
+-------+-------+--------+------+--------------+--------+-------------------+
| year  | #Rows | #Files | Size | Bytes Cached | Format | Incremental stats |
+-------+-------+--------+------+--------------+--------+-------------------+
| Total | -1    | 0      | 0B   | 0B           |        |                   |
+-------+-------+--------+------+--------------+--------+-------------------+
[localhost:21000] > show partitions no_stats_partitioned;
+-------+-------+--------+------+--------------+--------+-------------------+
| year  | #Rows | #Files | Size | Bytes Cached | Format | Incremental stats |
+-------+-------+--------+------+--------------+--------+-------------------+
```

```
| Total | -1    | 0      | 0B   | 0B           |        |                   |
+-------+-------+--------+------+--------------+--------+-------------------+
[localhost:21000] > alter table no_stats_partitioned add partition (year=2013);
[localhost:21000] > compute stats no_stats_partitioned;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
+-----------------------------------------+
[localhost:21000] > alter table no_stats_partitioned add partition (year=2014);
[localhost:21000] > show partitions no_stats_partitioned;
+-------+-------+--------+------+--------------+--------+-------------------+
| year  | #Rows | #Files | Size | Bytes Cached | Format | Incremental stats |
+-------+-------+--------+------+--------------+--------+-------------------+
| 2013  | 0     | 0      | 0B   | NOT CACHED   | TEXT   | false             |
| 2014  | -1    | 0      | 0B   | NOT CACHED   | TEXT   | false             |
| Total | 0     | 0      | 0B   | 0B           |        |                   |
+-------+-------+--------+------+--------------+--------+-------------------+
```



Note: Because the default `COMPUTE STATS` statement creates and updates statistics for all partitions in a table, if you expect to frequently add new partitions, use the `COMPUTE INCREMENTAL STATS` syntax instead, which lets you compute stats for a single specified partition, or only for those partitions that do not already have incremental stats.

If checking each individual table is impractical, due to a large number of tables or views that hide the underlying base tables, you can also check for missing statistics for a particular query. Use the `EXPLAIN` statement to preview query efficiency before actually running the query. Use the query profile output available through the `PROFILE` command in `impala-shell` or the web UI to verify query execution and timing after running the query. Both the `EXPLAIN` plan and the `PROFILE` output display a warning if any tables or partitions involved in the query do not have statistics.

```
[localhost:21000] > create table no_stats (x int);
[localhost:21000] > explain select count(*) from no_stats;
+----------------------------------------------------------------------------------------+
| Explain String                                                                         |
+----------------------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=10.00MB VCores=1                               |
| WARNING: The following tables are missing relevant table and/or column statistics.     |
| incremental_stats.no_stats                                                             |
|                                                                                        |
| 03:AGGREGATE [FINALIZE]                                                                |
| |  output: count:merge(*)                                                              |
| |                                                                                      |
| 02:EXCHANGE [UNPARTITIONED]                                                            |
| |                                                                                      |
| 01:AGGREGATE                                                                           |
| |  output: count(*)                                                                    |
| |                                                                                      |
| 00:SCAN HDFS [incremental_stats.no_stats]                                              |
|    partitions=1/1 files=0 size=0B                                                      |
+----------------------------------------------------------------------------------------+
```

Because Impala uses the ***partition pruning*** technique when possible to only evaluate certain partitions, if you have a partitioned table with statistics for some partitions and not others, whether or not the `EXPLAIN` statement shows the warning depends on the actual partitions used by the query. For example, you might see warnings or not for different queries against the same table:

```
-- No warning because all the partitions for the year 2012 have stats.
EXPLAIN SELECT ... FROM t1 WHERE year = 2012;

-- Missing stats warning because one or more partitions in this range
-- do not have stats.
EXPLAIN SELECT ... FROM t1 WHERE year BETWEEN 2006 AND 2009;
```

To confirm if any partitions at all in the table are missing statistics, you might explain a query that scans the entire table, such as `SELECT COUNT(*) FROM` *`table_name`*.

Manually Setting Table and Column Statistics with ALTER TABLE

Setting Table Statistics

The most crucial piece of data in all the statistics is the number of rows in the table (for an unpartitioned or partitioned table) and for each partition (for a partitioned table). The COMPUTE STATS statement always gathers statistics about all columns, as well as overall table statistics. If it is not practical to do a full COMPUTE STATS or COMPUTE INCREMENTAL STATS operation after adding a partition or inserting data, or if you can see that Impala would produce a more efficient plan if the number of rows was different, you can manually set the number of rows through an ALTER TABLE statement:

```
-- Set total number of rows. Applies to both unpartitioned and partitioned tables.
alter table table_name set tblproperties('numRows'='new_value',
'STATS_GENERATED_VIA_STATS_TASK'='true');

-- Set total number of rows for a specific partition. Applies to partitioned tables
only.
-- You must specify all the partition key columns in the PARTITION clause.
alter table table_name partition (keycol1=val1,keycol2=val2...) set
tblproperties('numRows'='new_value', 'STATS_GENERATED_VIA_STATS_TASK'='true');
```

This statement avoids re-scanning any data files. (The requirement to include the STATS_GENERATED_VIA_STATS_TASK property is relatively new, as a result of the issue HIVE-8648 for the Hive metastore.)

```
create table analysis_data stored as parquet as select * from raw_data;
Inserted 1000000000 rows in 181.98s
compute stats analysis_data;
insert into analysis_data select * from smaller_table_we_forgot_before;
Inserted 1000000 rows in 15.32s
-- Now there are 1001000000 rows. We can update this single data point in the stats.
alter table analysis_data set tblproperties('numRows'='1001000000',
'STATS_GENERATED_VIA_STATS_TASK'='true');
```

For a partitioned table, update both the per-partition number of rows and the number of rows for the whole table:

```
-- If the table originally contained 1 million rows, and we add another partition with
 30 thousand rows,
-- change the numRows property for the partition and the overall table.
alter table partitioned_data partition(year=2009, month=4) set tblproperties
('numRows'='30000', 'STATS_GENERATED_VIA_STATS_TASK'='true');
alter table partitioned_data set tblproperties ('numRows'='1030000',
'STATS_GENERATED_VIA_STATS_TASK'='true');
```

In practice, the COMPUTE STATS statement, or COMPUTE INCREMENTAL STATS for a partitioned table, should be fast and convenient enough that this technique is only useful for the very largest partitioned tables. Because the column statistics might be left in a stale state, do not use this technique as a replacement for COMPUTE STATS. Only use this technique if all other means of collecting statistics are impractical, or as a low-overhead operation that you run in between periodic COMPUTE STATS or COMPUTE INCREMENTAL STATS operations.

Setting Column Statistics

In CDH 5.8 / Impala 2.6 and higher, you can also use the SET COLUMN STATS clause of ALTER TABLE to manually set or change column statistics. Only use this technique in cases where it is impractical to run COMPUTE STATS or COMPUTE INCREMENTAL STATS frequently enough to keep up with data changes for a huge table.

You specify a case-insensitive symbolic name for the kind of statistics: numDVs, numNulls, avgSize, maxSize. The key names and values are both quoted. This operation applies to an entire table, not a specific partition. For example:

```
create table t1 (x int, s string);
insert into t1 values (1, 'one'), (2, 'two'), (2, 'deux');
show column stats t1;
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
```

```
| x      | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
alter table t1 set column stats x ('numDVs'='2','numNulls'='0');
alter table t1 set column stats s ('numdvs'='3','maxsize'='4');
show column stats t1;
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| x      | INT    | 2                | 0      | 4        | 4        |
| s      | STRING | 3                | -1     | 4        | -1       |
+--------+--------+------------------+--------+----------+----------+
```

Examples of Using Table and Column Statistics with Impala

The following examples walk through a sequence of `SHOW TABLE STATS`, `SHOW COLUMN STATS`, `ALTER TABLE`, and `SELECT` and `INSERT` statements to illustrate various aspects of how Impala uses statistics to help optimize queries.

This example shows table and column statistics for the `STORE` column used in the TPC-DS benchmarks for decision support systems. It is a tiny table holding data for 12 stores. Initially, before any statistics are gathered by a `COMPUTE STATS` statement, most of the numeric fields show placeholder values of -1, indicating that the figures are unknown. The figures that are filled in are values that are easily countable or deducible at the physical level, such as the number of files, total data size of the files, and the maximum and average sizes for data types that have a constant size such as `INT`, `FLOAT`, and `TIMESTAMP`.

```
[localhost:21000] > show table stats store;
+-------+--------+--------+--------+
| #Rows | #Files | Size   | Format |
+-------+--------+--------+--------+
| -1    | 1      | 3.08KB | TEXT   |
+-------+--------+--------+--------+
Returned 1 row(s) in 0.03s
[localhost:21000] > show column stats store;
+--------------------+-----------+------------------+--------+----------+----------+
| Column             | Type      | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------------------+-----------+------------------+--------+----------+----------+
| s_store_sk         | INT       | -1               | -1     | 4        | 4        |
| s_store_id         | STRING    | -1               | -1     | -1       | -1       |
| s_rec_start_date   | TIMESTAMP | -1               | -1     | 16       | 16       |
| s_rec_end_date     | TIMESTAMP | -1               | -1     | 16       | 16       |
| s_closed_date_sk   | INT       | -1               | -1     | 4        | 4        |
| s_store_name       | STRING    | -1               | -1     | -1       | -1       |
| s_number_employees | INT       | -1               | -1     | 4        | 4        |
| s_floor_space      | INT       | -1               | -1     | 4        | 4        |
| s_hours            | STRING    | -1               | -1     | -1       | -1       |
| s_manager          | STRING    | -1               | -1     | -1       | -1       |
| s_market_id        | INT       | -1               | -1     | 4        | 4        |
| s_geography_class  | STRING    | -1               | -1     | -1       | -1       |
| s_market_desc      | STRING    | -1               | -1     | -1       | -1       |
| s_market_manager   | STRING    | -1               | -1     | -1       | -1       |
| s_division_id      | INT       | -1               | -1     | 4        | 4        |
| s_division_name    | STRING    | -1               | -1     | -1       | -1       |
| s_company_id       | INT       | -1               | -1     | 4        | 4        |
| s_company_name     | STRING    | -1               | -1     | -1       | -1       |
| s_street_number    | STRING    | -1               | -1     | -1       | -1       |
| s_street_name      | STRING    | -1               | -1     | -1       | -1       |
| s_street_type      | STRING    | -1               | -1     | -1       | -1       |
| s_suite_number     | STRING    | -1               | -1     | -1       | -1       |
| s_city             | STRING    | -1               | -1     | -1       | -1       |
| s_county           | STRING    | -1               | -1     | -1       | -1       |
| s_state            | STRING    | -1               | -1     | -1       | -1       |
| s_zip              | STRING    | -1               | -1     | -1       | -1       |
| s_country          | STRING    | -1               | -1     | -1       | -1       |
| s_gmt_offset       | FLOAT     | -1               | -1     | 4        | 4        |
| s_tax_percentage   | FLOAT     | -1               | -1     | 4        | 4        |
+--------------------+-----------+------------------+--------+----------+----------+
Returned 29 row(s) in 0.04s
```

With the Hive ANALYZE TABLE statement for column statistics, you had to specify each column for which to gather statistics. The Impala COMPUTE STATS statement automatically gathers statistics for all columns, because it reads through the entire table relatively quickly and can efficiently compute the values for all the columns. This example shows how after running the COMPUTE STATS statement, statistics are filled in for both the table and all its columns:

```
[localhost:21000] > compute stats store;
+------------------------------------------+
| summary                                  |
+------------------------------------------+
| Updated 1 partition(s) and 29 column(s). |
+------------------------------------------+
Returned 1 row(s) in 1.88s
[localhost:21000] > show table stats store;
+-------+--------+--------+--------+
| #Rows | #Files | Size   | Format |
+-------+--------+--------+--------+
| 12    | 1      | 3.08KB | TEXT   |
+-------+--------+--------+--------+
Returned 1 row(s) in 0.02s
[localhost:21000] > show column stats store;
+-------------------+-----------+------------------+--------+----------+-------------------+
| Column            | Type      | #Distinct Values | #Nulls | Max Size | Avg Size
      |
+-------------------+-----------+------------------+--------+----------+-------------------+
| s_store_sk        | INT       | 12               | -1     | 4        | 4
      |
| s_store_id        | STRING    | 6                | -1     | 16       | 16
      |
| s_rec_start_date  | TIMESTAMP | 4                | -1     | 16       | 16
      |
| s_rec_end_date    | TIMESTAMP | 3                | -1     | 16       | 16
      |
| s_closed_date_sk  | INT       | 3                | -1     | 4        | 4
      |
| s_store_name      | STRING    | 8                | -1     | 5        | 4.25
      |
| s_number_employees | INT       | 9                | -1     | 4        | 4
      |
| s_floor_space     | INT       | 10               | -1     | 4        | 4
      |
| s_hours           | STRING    | 2                | -1     | 8        |
7.083300113677979 |
| s_manager         | STRING    | 7                | -1     | 15       | 12
      |
| s_market_id       | INT       | 7                | -1     | 4        | 4
      |
| s_geography_class | STRING    | 1                | -1     | 7        | 7
      |
| s_market_desc     | STRING    | 10               | -1     | 94       | 55.5
      |
| s_market_manager  | STRING    | 7                | -1     | 16       | 14
      |
| s_division_id     | INT       | 1                | -1     | 4        | 4
      |
| s_division_name   | STRING    | 1                | -1     | 7        | 7
      |
| s_company_id      | INT       | 1                | -1     | 4        | 4
      |
| s_company_name    | STRING    | 1                | -1     | 7        | 7
      |
| s_street_number   | STRING    | 9                | -1     | 3        |
2.833300113677979 |
| s_street_name     | STRING    | 12               | -1     | 11       |
6.583300113677979 |
| s_street_type     | STRING    | 8                | -1     | 9        |
4.833300113677979 |
| s_suite_number    | STRING    | 11               | -1     | 9        | 8.25
      |
| s_city            | STRING    | 2                | -1     | 8        | 6.5
      |
| s_county          | STRING    | 1                | -1     | 17       | 17
      |
```

```
| s_state            | STRING    | 1                | -1     | 2         | 2
      |
| s_zip              | STRING    | 2                | -1     | 5         | 5
      |
| s_country          | STRING    | 1                | -1     | 13        | 13
      |
| s_gmt_offset       | FLOAT     | 1                | -1     | 4         | 4
      |
| s_tax_percentage   | FLOAT     | 5                | -1     | 4         | 4
      |
+--------------------+-----------+------------------+--------+-----------+-------------------+
Returned 29 row(s) in 0.04s
```

The following example shows how statistics are represented for a partitioned table. In this case, we have set up a table to hold the world's most trivial census data, a single STRING field, partitioned by a YEAR column. The table statistics include a separate entry for each partition, plus final totals for the numeric fields. The column statistics include some easily deducible facts for the partitioning column, such as the number of distinct values (the number of partition subdirectories).

```
localhost:21000] > describe census;
+------+----------+---------+
| name | type     | comment |
+------+----------+---------+
| name | string   |         |
| year | smallint |         |
+------+----------+---------+
Returned 2 row(s) in 0.02s
[localhost:21000] > show table stats census;
+-------+-------+--------+------+---------+
| year  | #Rows | #Files | Size | Format  |
+-------+-------+--------+------+---------+
| 2000  | -1    | 0      | 0B   | TEXT    |
| 2004  | -1    | 0      | 0B   | TEXT    |
| 2008  | -1    | 0      | 0B   | TEXT    |
| 2010  | -1    | 0      | 0B   | TEXT    |
| 2011  | 0     | 1      | 22B  | TEXT    |
| 2012  | -1    | 1      | 22B  | TEXT    |
| 2013  | -1    | 1      | 231B | PARQUET |
| Total | 0     | 3      | 275B |         |
+-------+-------+--------+------+---------+
Returned 8 row(s) in 0.02s
[localhost:21000] > show column stats census;
+--------+----------+------------------+--------+----------+----------+
| Column | Type     | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+----------+------------------+--------+----------+----------+
| name   | STRING   | -1               | -1     | -1       | -1       |
| year   | SMALLINT | 7                | -1     | 2        | 2        |
+--------+----------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.02s
```

The following example shows how the statistics are filled in by a COMPUTE STATS statement in Impala.

```
[localhost:21000] > compute stats census;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 3 partition(s) and 1 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 2.16s
[localhost:21000] > show table stats census;
+-------+-------+--------+------+---------+
| year  | #Rows | #Files | Size | Format  |
+-------+-------+--------+------+---------+
| 2000  | -1    | 0      | 0B   | TEXT    |
| 2004  | -1    | 0      | 0B   | TEXT    |
| 2008  | -1    | 0      | 0B   | TEXT    |
| 2010  | -1    | 0      | 0B   | TEXT    |
| 2011  | 4     | 1      | 22B  | TEXT    |
| 2012  | 4     | 1      | 22B  | TEXT    |
| 2013  | 1     | 1      | 231B | PARQUET |
```

```
| Total | 9     | 3      | 275B |         |
+-------+-------+--------+------+---------+
Returned 8 row(s) in 0.02s
[localhost:21000] > show column stats census;
+--------+----------+------------------+--------+----------+----------+
| Column | Type     | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+----------+------------------+--------+----------+----------+
| name   | STRING   | 4                | -1     | 5        | 4.5      |
| year   | SMALLINT | 7                | -1     | 2        | 2        |
+--------+----------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.02s
```

For examples showing how some queries work differently when statistics are available, see Examples of Join Order Optimization on page 570. You can see how Impala executes a query differently in each case by observing the `EXPLAIN` output before and after collecting statistics. Measure the before and after query times, and examine the throughput numbers in before and after `SUMMARY` or `PROFILE` output, to verify how much the improved plan speeds up performance.

Benchmarking Impala Queries

Because Impala, like other Hadoop components, is designed to handle large data volumes in a distributed environment, conduct any performance tests using realistic data and cluster configurations. Use a multi-node cluster rather than a single node; run queries against tables containing terabytes of data rather than tens of gigabytes. The parallel processing techniques used by Impala are most appropriate for workloads that are beyond the capacity of a single server.

When you run queries returning large numbers of rows, the CPU time to pretty-print the output can be substantial, giving an inaccurate measurement of the actual query time. Consider using the `-B` option on the `impala-shell` command to turn off the pretty-printing, and optionally the `-o` option to store query results in a file rather than printing to the screen. See impala-shell Configuration Options on page 714 for details.

Controlling Impala Resource Usage

Sometimes, balancing raw query performance against scalability requires limiting the amount of resources, such as memory or CPU, used by a single query or group of queries. Impala can use several mechanisms that help to smooth out the load during heavy concurrent usage, resulting in faster overall query times and sharing of resources across Impala queries, MapReduce jobs, and other kinds of workloads across a CDH cluster:

- The Impala admission control feature uses a fast, distributed mechanism to hold back queries that exceed limits on the number of concurrent queries or the amount of memory used. The queries are queued, and executed as other queries finish and resources become available. You can control the concurrency limits, and specify different limits for different groups of users to divide cluster resources according to the priorities of different classes of users. This feature is new in Impala 1.3. See Admission Control and Query Queuing on page 549 for details.
- You can restrict the amount of memory Impala reserves during query execution by specifying the `-mem_limit` option for the `impalad` daemon. See Modifying Impala Startup Options for details. This limit applies only to the memory that is directly consumed by queries; Impala reserves additional memory at startup, for example to hold cached metadata.
- For production deployment, Cloudera recommends that you implement resource isolation using mechanisms such as cgroups, which you can configure using Cloudera Manager. For details, see Static Resource Pools in the Cloudera Manager documentation.

Runtime Filtering for Impala Queries (CDH 5.7 or higher only)

Runtime filtering is a wide-ranging optimization feature available in CDH 5.7 / Impala 2.5 and higher. When only a fraction of the data in a table is needed for a query against a partitioned table or to evaluate a join condition, Impala determines the appropriate conditions while the query is running, and broadcasts that information to all the `impalad`

nodes that are reading the table so that they can avoid unnecessary I/O to read partition data, and avoid unnecessary network transmission by sending only the subset of rows that match the join keys across the network.

This feature is primarily used to optimize queries against large partitioned tables (under the name ***dynamic partition pruning***) and joins of large tables. The information in this section includes concepts, internals, and troubleshooting information for the entire runtime filtering feature. For specific tuning steps for partitioned tables, see Dynamic Partition Pruning on page 628.



Important:

When this feature made its debut in CDH 5.7, the default setting was `RUNTIME_FILTER_MODE=LOCAL`. Now the default is `RUNTIME_FILTER_MODE=GLOBAL` in CDH 5.8 / Impala 2.6 and higher, which enables more wide-ranging and ambitious query optimization without requiring you to explicitly set any query options.

Background Information for Runtime Filtering

To understand how runtime filtering works at a detailed level, you must be familiar with some terminology from the field of distributed database technology:

- What a ***plan fragment*** is. Impala decomposes each query into smaller units of work that are distributed across the cluster. Wherever possible, a data block is read, filtered, and aggregated by plan fragments executing on the same host. For some operations, such as joins and combining intermediate results into a final result set, data is transmitted across the network from one Impala daemon to another.

- What `SCAN` and `HASH JOIN` plan nodes are, and their role in computing query results:

  In the Impala query plan, a ***scan node*** performs the I/O to read from the underlying data files. Although this is an expensive operation from the traditional database perspective, Hadoop clusters and Impala are optimized to do this kind of I/O in a highly parallel fashion. The major potential cost savings come from using the columnar Parquet format (where Impala can avoid reading data for unneeded columns) and partitioned tables (where Impala can avoid reading data for unneeded partitions).

  Most Impala joins use the ***hash join*** mechanism. (It is only fairly recently that Impala started using the nested-loop join technique, for certain kinds of non-equijoin queries.) In a hash join, when evaluating join conditions from two tables, Impala constructs a hash table in memory with all the different column values from the table on one side of the join. Then, for each row from the table on the other side of the join, Impala tests whether the relevant column values are in this hash table or not.

  A ***hash join node*** constructs such an in-memory hash table, then performs the comparisons to identify which rows match the relevant join conditions and should be included in the result set (or at least sent on to the subsequent intermediate stage of query processing). Because some of the input for a hash join might be transmitted across the network from another host, it is especially important from a performance perspective to prune out ahead of time any data that is known to be irrelevant.

  The more distinct values are in the columns used as join keys, the larger the in-memory hash table and thus the more memory required to process the query.

- The difference between a ***broadcast join*** and a ***shuffle join***. (The Hadoop notion of a shuffle join is sometimes referred to in Impala as a ***partitioned join***.) In a broadcast join, the table from one side of the join (typically the smaller table) is sent in its entirety to all the hosts involved in the query. Then each host can compare its portion of the data from the other (larger) table against the full set of possible join keys. In a shuffle join, there is no obvious “smaller” table, and so the contents of both tables are divided up, and corresponding portions of the data are transmitted to each host involved in the query. See Optimizer Hints in Impala on page 387 for information about how these different kinds of joins are processed.

- The notion of the build phase and probe phase when Impala processes a join query. The ***build phase*** is where the rows containing the join key columns, typically for the smaller table, are transmitted across the network and built into an in-memory hash table data structure on one or more destination nodes. The ***probe phase*** is where data is read locally (typically from the larger table) and the join key columns are compared to the values in the in-memory

hash table. The corresponding input sources (tables, subqueries, and so on) for these phases are referred to as the ***build side*** and the ***probe side***.

- How to set Impala query options: interactively within an `impala-shell` session through the `SET` command, for a JDBC or ODBC application through the `SET` statement, or globally for all `impalad` daemons through the `default_query_options` configuration setting.

Runtime Filtering Internals

The ***filter*** that is transmitted between plan fragments is essentially a list of values for join key columns. When this list of values is transmitted in time to a scan node, Impala can filter out non-matching values immediately after reading them, rather than transmitting the raw data to another host to compare against the in-memory hash table on that host.

For HDFS-based tables, this data structure is implemented as a ***Bloom filter***, which uses a probability-based algorithm to determine all possible matching values. (The probability-based aspects means that the filter might include some non-matching values, but if so, that does not cause any inaccuracy in the final results.)

Another kind of filter is the “min-max” filter. It currently only applies to Kudu tables. The filter is a data structure representing a minimum and maximum value. These filters are passed to Kudu to reduce the number of rows returned to Impala when scanning the probe side of the join.

There are different kinds of filters to match the different kinds of joins (partitioned and broadcast). A broadcast filter reflects the complete list of relevant values and can be immediately evaluated by a scan node. A partitioned filter reflects only the values processed by one host in the cluster; all the partitioned filters must be combined into one (by the coordinator node) before the scan nodes can use the results to accurately filter the data as it is read from storage.

Broadcast filters are also classified as local or global. With a local broadcast filter, the information in the filter is used by a subsequent query fragment that is running on the same host that produced the filter. A non-local broadcast filter must be transmitted across the network to a query fragment that is running on a different host. Impala designates 3 hosts to each produce non-local broadcast filters, to guard against the possibility of a single slow host taking too long. Depending on the setting of the `RUNTIME_FILTER_MODE` query option (`LOCAL` or `GLOBAL`), Impala either uses a conservative optimization strategy where filters are only consumed on the same host that produced them, or a more aggressive strategy where filters are eligible to be transmitted across the network.



Note: In CDH 5.8 / Impala 2.6 and higher, the default for runtime filtering is the `GLOBAL` setting.

File Format Considerations for Runtime Filtering

Parquet tables get the most benefit from the runtime filtering optimizations. Runtime filtering can speed up join queries against partitioned or unpartitioned Parquet tables, and single-table queries against partitioned Parquet tables. See Using the Parquet File Format with Impala Tables on page 643 for information about using Parquet tables with Impala.

For other file formats (text, Avro, RCFile, and SequenceFile), runtime filtering speeds up queries against partitioned tables only. Because partitioned tables can use a mixture of formats, Impala produces the filters in all cases, even if they are not ultimately used to optimize the query.

Wait Intervals for Runtime Filters

Because it takes time to produce runtime filters, especially for partitioned filters that must be combined by the coordinator node, there is a time interval above which it is more efficient for the scan nodes to go ahead and construct their intermediate result sets, even if that intermediate data is larger than optimal. If it only takes a few seconds to produce the filters, it is worth the extra time if pruning the unnecessary data can save minutes in the overall query time. You can specify the maximum wait time in milliseconds using the `RUNTIME_FILTER_WAIT_TIME_MS` query option.

By default, each scan node waits for up to 1 second (1000 milliseconds) for filters to arrive. If all filters have not arrived within the specified interval, the scan node proceeds, using whatever filters did arrive to help avoid reading unnecessary

data. If a filter arrives after the scan node begins reading data, the scan node applies that filter to the data that is read after the filter arrives, but not to the data that was already read.

If the cluster is relatively busy and your workload contains many resource-intensive or long-running queries, consider increasing the wait time so that complicated queries do not miss opportunities for optimization. If the cluster is lightly loaded and your workload contains many small queries taking only a few seconds, consider decreasing the wait time to avoid the 1 second delay for each query.

Query Options for Runtime Filtering

See the following sections for information about the query options that control runtime filtering:

- The first query option adjusts the "sensitivity" of this feature. By default, it is set to the highest level (GLOBAL). (This default applies to CDH 5.8 / Impala 2.6 and higher. In previous releases, the default was LOCAL.)
  - RUNTIME_FILTER_MODE Query Option (CDH 5.7 or higher only) on page 358
- The other query options are tuning knobs that you typically only adjust after doing performance testing, and that you might want to change only for the duration of a single expensive query:
  - MAX_NUM_RUNTIME_FILTERS Query Option (CDH 5.7 or higher only) on page 339
  - DISABLE_ROW_RUNTIME_FILTERING Query Option (CDH 5.7 or higher only) on page 328
  - RUNTIME_FILTER_MAX_SIZE Query Option (CDH 5.8 or higher only) on page 357
  - RUNTIME_FILTER_MIN_SIZE Query Option (CDH 5.8 or higher only) on page 357
  - RUNTIME_BLOOM_FILTER_SIZE Query Option (CDH 5.7 or higher only) on page 356; in CDH 5.8 / Impala 2.6 and higher, this setting acts as a fallback when statistics are not available, rather than as a directive.

Runtime Filtering and Query Plans

In the same way the query plan displayed by the EXPLAIN statement includes information about predicates used by each plan fragment, it also includes annotations showing whether a plan fragment produces or consumes a runtime filter. A plan fragment that produces a filter includes an annotation such as `runtime filters: filter_id <- table.column`, while a plan fragment that consumes a filter includes an annotation such as `runtime filters: filter_id -> table.column`. Setting the query option EXPLAIN_LEVEL=2 adds additional annotations showing the type of the filter, either `filter_id[bloom]` (for HDFS-based tables) or `filter_id[min_max]` (for Kudu tables).

The following example shows a query that uses a single runtime filter, labeled RF000, to prune the partitions based on evaluating the result set of a subquery at runtime:

```
CREATE TABLE yy (s STRING) PARTITIONED BY (year INT);
INSERT INTO yy PARTITION (year) VALUES ('1999', 1999), ('2000', 2000),
  ('2001', 2001), ('2010', 2010), ('2018', 2018);
COMPUTE STATS yy;

CREATE TABLE yy2 (s STRING, year INT);
INSERT INTO yy2 VALUES ('1999', 1999), ('2000', 2000), ('2001', 2001);
COMPUTE STATS yy2;

-- The following query reads an unknown number of partitions, whose key values
-- are only known at run time. The runtime filters line shows the
-- information used in query fragment 02 to decide which partitions to skip.

EXPLAIN SELECT s FROM yy WHERE year IN (SELECT year FROM yy2);
+-----------------------------------------------------------------------------+
| PLAN-ROOT SINK                                                              |
| |                                                                           |
| 04:EXCHANGE [UNPARTITIONED]                                                 |
| |                                                                           |
| 02:HASH JOIN [LEFT SEMI JOIN, BROADCAST]                                    |
| |  hash predicates: year = year                                             |
```

```
| |  runtime filters: RF000 <- year                                      |
| |                                                                      |
| |--03:EXCHANGE [BROADCAST]                                             |
| |  |                                                                   |
| |  01:SCAN HDFS [default.yy2]                                          |
| |     partitions=1/1 files=1 size=620B                                 |
| |                                                                      |
| 00:SCAN HDFS [default.yy]                                              |
|    partitions=5/5 files=5 size=1.71KB                                |
|    runtime filters: RF000 -> year                                      |
+------------------------------------------------------------------------+

SELECT s FROM yy WHERE year IN (SELECT year FROM yy2); -- Returns 3 rows from yy
PROFILE;
```

The query profile (displayed by the `PROFILE` command in `impala-shell`) contains both the `EXPLAIN` plan and more detailed information about the internal workings of the query. The profile output includes the `Filter routing table` section with information about each filter based on its ID.

Examples of Queries that Benefit from Runtime Filtering

In this example, Impala would normally do extra work to interpret the columns `C1`, `C2`, `C3`, and `ID` for each row in `HUGE_T1`, before checking the `ID` value against the in-memory hash table constructed from all the `TINY_T2.ID` values. By producing a filter containing all the `TINY_T2.ID` values even before the query starts scanning the `HUGE_T1` table, Impala can skip the unnecessary work to parse the column info as soon as it determines that an `ID` value does not match any of the values from the other table.

The example shows `COMPUTE STATS` statements for both the tables (even though that is a one-time operation after loading data into those tables) because Impala relies on up-to-date statistics to determine which one has more distinct `ID` values than the other. That information lets Impala make effective decisions about which table to use to construct the in-memory hash table, and which table to read from disk and compare against the entries in the hash table.

```
COMPUTE STATS huge_t1;
COMPUTE STATS tiny_t2;
SELECT c1, c2, c3 FROM huge_t1 JOIN tiny_t2 WHERE huge_t1.id = tiny_t2.id;
```

In this example, `T1` is a table partitioned by year. The subquery on `T2` produces multiple values, and transmits those values as a filter to the plan fragments that are reading from `T1`. Any non-matching partitions in `T1` are skipped.

```
select c1 from t1 where year in (select distinct year from t2);
```

Now the `WHERE` clause contains an additional test that does not apply to the partition key column. A filter on a column that is not a partition key is called a per-row filter. Because per-row filters only apply for Parquet, `T1` must be a Parquet table.

The subqueries result in two filters being transmitted to the scan nodes that read from `T1`. The filter on `YEAR` helps the query eliminate entire partitions based on non-matching years. The filter on `C2` lets Impala discard rows with non-matching `C2` values immediately after reading them. Without runtime filtering, Impala would have to keep the non-matching values in memory, assemble `C1`, `C2`, and `C3` into rows in the intermediate result set, and transmit all the intermediate rows back to the coordinator node, where they would be eliminated only at the very end of the query.

```
select c1, c2, c3 from t1
  where year in (select distinct year from t2)
    and c2 in (select other_column from t3);
```

This example involves a broadcast join. The fact that the ON clause would return a small number of matching rows (because there are not very many rows in TINY_T2) means that the corresponding filter is very selective. Therefore, runtime filtering will probably be effective in optimizing this query.

```
select c1 from huge_t1 join [broadcast] tiny_t2
  on huge_t1.id = tiny_t2.id
  where huge_t1.year in (select distinct year from tiny_t2)
    and c2 in (select other_column from t3);
```

This example involves a shuffle or partitioned join. Assume that most rows in HUGE_T1 have a corresponding row in HUGE_T2. The fact that the ON clause could return a large number of matching rows means that the corresponding filter would not be very selective. Therefore, runtime filtering might be less effective in optimizing this query.

```
select c1 from huge_t1 join [shuffle] huge_t2
  on huge_t1.id = huge_t2.id
  where huge_t1.year in (select distinct year from huge_t2)
    and c2 in (select other_column from t3);
```

Tuning and Troubleshooting Queries that Use Runtime Filtering

These tuning and troubleshooting procedures apply to queries that are resource-intensive enough, long-running enough, and frequent enough that you can devote special attention to optimizing them individually.

Use the EXPLAIN statement and examine the `runtime filters:` lines to determine whether runtime filters are being applied to the WHERE predicates and join clauses that you expect. For example, runtime filtering does not apply to queries that use the nested loop join mechanism due to non-equijoin operators.

Make sure statistics are up-to-date for all tables involved in the queries. Use the COMPUTE STATS statement after loading data into non-partitioned tables, and COMPUTE INCREMENTAL STATS after adding new partitions to partitioned tables.

If join queries involving large tables use unique columns as the join keys, for example joining a primary key column with a foreign key column, the overhead of producing and transmitting the filter might outweigh the performance benefit because not much data could be pruned during the early stages of the query. For such queries, consider setting the query option RUNTIME_FILTER_MODE=OFF.

Limitations and Restrictions for Runtime Filtering

The runtime filtering feature is most effective for the Parquet file formats. For other file formats, filtering only applies for partitioned tables. See File Format Considerations for Runtime Filtering on page 590. For the ways in which runtime filtering works for Kudu tables, see Impala Query Performance for Kudu Tables on page 683.

When the spill-to-disk mechanism is activated on a particular host during a query, that host does not produce any filters while processing that query. This limitation does not affect the correctness of results; it only reduces the amount of optimization that can be applied to the query.

Using HDFS Caching with Impala (CDH 5.3 or higher only)

HDFS caching provides performance and scalability benefits in production environments where Impala queries and other Hadoop jobs operate on quantities of data much larger than the physical RAM on the DataNodes, making it impractical to rely on the Linux OS cache, which only keeps the most recently used data in memory. Data read from the HDFS cache avoids the overhead of checksumming and memory-to-memory copying involved when using data from the Linux OS cache.



Note:

On a small or lightly loaded cluster, HDFS caching might not produce any speedup. It might even lead to slower queries, if I/O read operations that were performed in parallel across the entire cluster are replaced by in-memory operations operating on a smaller number of hosts. The hosts where the HDFS blocks are cached can become bottlenecks because they experience high CPU load while processing the cached data blocks, while other hosts remain idle. Therefore, always compare performance with and without this feature enabled, using a realistic workload.

In CDH 5.4 / Impala 2.2 and higher, you can spread the CPU load more evenly by specifying the `WITH REPLICATION` clause of the `CREATE TABLE` and `ALTER TABLE` statements. This clause lets you control the replication factor for HDFS caching for a specific table or partition. By default, each cached block is only present on a single host, which can lead to CPU contention if the same host processes each cached block. Increasing the replication factor lets Impala choose different hosts to process different cached blocks, to better distribute the CPU load. Always use a `WITH REPLICATION` setting of at least 3, and adjust upward if necessary to match the replication factor for the underlying HDFS data files.

In CDH 5.7 / Impala 2.5 and higher, Impala automatically randomizes which host processes a cached HDFS block, to avoid CPU hotspots. For tables where HDFS caching is not applied, Impala designates which host to process a data block using an algorithm that estimates the load on each host. If CPU hotspots still arise during queries, you can enable additional randomization for the scheduling algorithm for non-HDFS cached data by setting the `SCHEDULE_RANDOM_REPLICA` query option.

For background information about how to set up and manage HDFS caching for a CDH cluster, see the documentation for HDFS caching.

Overview of HDFS Caching for Impala

With CDH 5.1 / Impala 1.4 and higher, Impala can use the HDFS caching feature to make more effective use of RAM, so that repeated queries can take advantage of data “pinned” in memory regardless of how much data is processed overall. The HDFS caching feature lets you designate a subset of frequently accessed data to be pinned permanently in memory, remaining in the cache across multiple queries and never being evicted. This technique is suitable for tables or partitions that are frequently accessed and are small enough to fit entirely within the HDFS memory cache. For example, you might designate several dimension tables to be pinned in the cache, to speed up many different join queries that reference them. Or in a partitioned table, you might pin a partition holding data from the most recent time period because that data will be queried intensively; then when the next set of data arrives, you could unpin the previous partition and pin the partition holding the new data.

Because this Impala performance feature relies on HDFS infrastructure, it only applies to Impala tables that use HDFS data files. HDFS caching for Impala does not apply to HBase tables, S3 tables, Kudu tables, or Isilon tables.

Setting Up HDFS Caching for Impala

To use HDFS caching with Impala, first set up that feature for your CDH cluster:

- Decide how much memory to devote to the HDFS cache on each host. Remember that the total memory available for cached data is the sum of the cache sizes on all the hosts. By default, any data block is only cached on one host, although you can cache a block across multiple hosts by increasing the replication factor.
- Issue `hdfs cacheadmin` commands to set up one or more cache pools, owned by the same user as the `impalad` daemon (typically `impala`). For example:

```
hdfs cacheadmin -addPool four_gig_pool -owner impala -limit 4000000000
```

For details about the `hdfs cacheadmin` command, see the documentation for HDFS caching.

Once HDFS caching is enabled and one or more pools are available, see Enabling HDFS Caching for Impala Tables and Partitions on page 595 for how to choose which Impala data to load into the HDFS cache. On the Impala side, you specify

the cache pool name defined by the `hdfs cacheadmin` command in the Impala DDL statements that enable HDFS caching for a table or partition, such as `CREATE TABLE ... CACHED IN pool` or `ALTER TABLE ... SET CACHED IN pool`.

Enabling HDFS Caching for Impala Tables and Partitions

Begin by choosing which tables or partitions to cache. For example, these might be lookup tables that are accessed by many different join queries, or partitions corresponding to the most recent time period that are analyzed by different reports or ad hoc queries.

In your SQL statements, you specify logical divisions such as tables and partitions to be cached. Impala translates these requests into HDFS-level directives that apply to particular directories and files. For example, given a partitioned table CENSUS with a partition key column YEAR, you could choose to cache all or part of the data as follows:

In CDH 5.4 / Impala 2.2 and higher, the optional `WITH REPLICATION` clause for `CREATE TABLE` and `ALTER TABLE` lets you specify a ***replication factor***, the number of hosts on which to cache the same data blocks. When Impala processes a cached data block, where the cache replication factor is greater than 1, Impala randomly selects a host that has a cached copy of that data block. This optimization avoids excessive CPU usage on a single host when the same cached data block is processed multiple times. Cloudera recommends specifying a value greater than or equal to the HDFS block replication factor.

```
-- Cache the entire table (all partitions).
alter table census set cached in 'pool_name';

-- Remove the entire table from the cache.
alter table census set uncached;

-- Cache a portion of the table (a single partition).
-- If the table is partitioned by multiple columns (such as year, month, day),
-- the ALTER TABLE command must specify values for all those columns.
alter table census partition (year=1960) set cached in 'pool_name';

-- Cache the data from one partition on up to 4 hosts, to minimize CPU load on any
-- single host when the same data block is processed multiple times.
alter table census partition (year=1970)
  set cached in 'pool_name' with replication = 4;

-- At each stage, check the volume of cached data.
-- For large tables or partitions, the background loading might take some time,
-- so you might have to wait and reissue the statement until all the data
-- has finished being loaded into the cache.
show table stats census;
+-------+-------+--------+------+--------------+--------+
| year  | #Rows | #Files | Size | Bytes Cached | Format |
+-------+-------+--------+------+--------------+--------+
| 1900  | -1    | 1      | 11B  | NOT CACHED   | TEXT   |
| 1940  | -1    | 1      | 11B  | NOT CACHED   | TEXT   |
| 1960  | -1    | 1      | 11B  | 11B          | TEXT   |
| 1970  | -1    | 1      | 11B  | NOT CACHED   | TEXT   |
| Total | -1    | 4      | 44B  | 11B          |        |
+-------+-------+--------+------+--------------+--------+
```

CREATE TABLE considerations:

The HDFS caching feature affects the Impala `CREATE TABLE` statement as follows:

- You can put a `CACHED IN 'pool_name'` clause and optionally a `WITH REPLICATION = number_of_hosts` clause at the end of a `CREATE TABLE` statement to automatically cache the entire contents of the table, including any partitions added later. The *pool_name* is a pool that you previously set up with the `hdfs cacheadmin` command.
- Once a table is designated for HDFS caching through the `CREATE TABLE` statement, if new partitions are added later through `ALTER TABLE ... ADD PARTITION` statements, the data in those new partitions is automatically cached in the same pool.
- If you want to perform repetitive queries on a subset of data from a large table, and it is not practical to designate the entire table or specific partitions for HDFS caching, you can create a new cached table with just a subset of

the data by using CREATE TABLE ... CACHED IN '*pool_name*' AS SELECT ... WHERE .... When you are finished with generating reports from this subset of data, drop the table and both the data files and the data cached in RAM are automatically deleted.

See CREATE TABLE Statement on page 234 for the full syntax.

Other memory considerations:

Certain DDL operations, such as ALTER TABLE ... SET LOCATION, are blocked while the underlying HDFS directories contain cached files. You must uncache the files first, before changing the location, dropping the table, and so on.

When data is requested to be pinned in memory, that process happens in the background without blocking access to the data while the caching is in progress. Loading the data from disk could take some time. Impala reads each HDFS data block from memory if it has been pinned already, or from disk if it has not been pinned yet.

The amount of data that you can pin on each node through the HDFS caching mechanism is subject to a quota that is enforced by the underlying HDFS service. Before requesting to pin an Impala table or partition in memory, check that its size does not exceed this quota.



Note: Because the HDFS cache consists of combined memory from all the DataNodes in the cluster, cached tables or partitions can be bigger than the amount of HDFS cache memory on any single host.

Loading and Removing Data with HDFS Caching Enabled

When HDFS caching is enabled, extra processing happens in the background when you add or remove data through statements such as INSERT and DROP TABLE.

Inserting or loading data:

- When Impala performs an INSERT or LOAD DATA statement for a table or partition that is cached, the new data files are automatically cached and Impala recognizes that fact automatically.
- If you perform an INSERT or LOAD DATA through Hive, as always, Impala only recognizes the new data files after a REFRESH *table_name* statement in Impala.
- If the cache pool is entirely full, or becomes full before all the requested data can be cached, the Impala DDL statement returns an error. This is to avoid situations where only some of the requested data could be cached.
- When HDFS caching is enabled for a table or partition, new data files are cached automatically when they are added to the appropriate directory in HDFS, without the need for a REFRESH statement in Impala. Impala automatically performs a REFRESH once the new data is loaded into the HDFS cache.

Dropping tables, partitions, or cache pools:

The HDFS caching feature interacts with the Impala DROP TABLE and ALTER TABLE ... DROP PARTITION statements as follows:

- When you issue a DROP TABLE for a table that is entirely cached, or has some partitions cached, the DROP TABLE succeeds and all the cache directives Impala submitted for that table are removed from the HDFS cache system.
- The same applies to ALTER TABLE ... DROP PARTITION. The operation succeeds and any cache directives are removed.
- As always, the underlying data files are removed if the dropped table is an internal table, or the dropped partition is in its default location underneath an internal table. The data files are left alone if the dropped table is an external table, or if the dropped partition is in a non-default location.
- If you designated the data files as cached through the hdfs cacheadmin command, and the data files are left behind as described in the previous item, the data files remain cached. Impala only removes the cache directives submitted by Impala through the CREATE TABLE or ALTER TABLE statements. It is OK to have multiple redundant cache directives pertaining to the same files; the directives all have unique IDs and owners so that the system can tell them apart.
- If you drop an HDFS cache pool through the hdfs cacheadmin command, all the Impala data files are preserved, just no longer cached. After a subsequent REFRESH, SHOW TABLE STATS reports 0 bytes cached for each associated Impala table or partition.

Relocating a table or partition:

The HDFS caching feature interacts with the Impala `ALTER TABLE ... SET LOCATION` statement as follows:

- If you have designated a table or partition as cached through the `CREATE TABLE` or `ALTER TABLE` statements, subsequent attempts to relocate the table or partition through an `ALTER TABLE ... SET LOCATION` statement will fail. You must issue an `ALTER TABLE ... SET UNCACHED` statement for the table or partition first. Otherwise, Impala would lose track of some cached data files and have no way to uncache them later.

Administration for HDFS Caching with Impala

Here are the guidelines and steps to check or change the status of HDFS caching for Impala data:

hdfs cacheadmin command:

- If you drop a cache pool with the `hdfs cacheadmin` command, Impala queries against the associated data files will still work, by falling back to reading the files from disk. After performing a `REFRESH` on the table, Impala reports the number of bytes cached as 0 for all associated tables and partitions.
- You might use `hdfs cacheadmin` to get a list of existing cache pools, or detailed information about the pools, as follows:

```
hdfs cacheadmin -listDirectives         # Basic info
Found 122 entries
  ID POOL       REPL EXPIRY  PATH
 123 testPool      1 never   /user/hive/warehouse/tpcds.store_sales
 124 testPool      1 never   /user/hive/warehouse/tpcds.store_sales/ss_date=1998-01-15
 125 testPool      1 never   /user/hive/warehouse/tpcds.store_sales/ss_date=1998-02-01
...

hdfs cacheadmin -listDirectives -stats  # More details
Found 122 entries
  ID POOL       REPL EXPIRY  PATH                                                BYTES_NEEDED  BYTES_CACHED  FILES_NEEDED
  FILES_CACHED
 123 testPool      1 never   /user/hive/warehouse/tpcds.store_sales                         0             0             0
             0
 124 testPool      1 never   /user/hive/warehouse/tpcds.store_sales/ss_date=1998-01-15  143169        143169             1
             1
 125 testPool      1 never   /user/hive/warehouse/tpcds.store_sales/ss_date=1998-02-01  112447        112447             1
             1
...
```

Impala SHOW statement:

- For each table or partition, the `SHOW TABLE STATS` or `SHOW PARTITIONS` statement displays the number of bytes currently cached by the HDFS caching feature. If there are no cache directives in place for that table or partition, the result set displays `NOT CACHED`. A value of 0, or a smaller number than the overall size of the table or partition, indicates that the cache request has been submitted but the data has not been entirely loaded into memory yet. See SHOW Statement on page 363 for details.

Cloudera Manager:

- You can enable or disable HDFS caching through Cloudera Manager, using the configuration setting **Maximum Memory Used for Caching** for the HDFS service. This control sets the HDFS configuration parameter `dfs_datanode_max_locked_memory`, which specifies the upper limit of HDFS cache size on each node.
- All the other manipulation of the HDFS caching settings, such as what files are cached, is done through the command line, either Impala DDL statements or the Linux `hdfs cacheadmin` command.

Impala memory limits:

The Impala HDFS caching feature interacts with the Impala memory limits as follows:

- The maximum size of each HDFS cache pool is specified externally to Impala, through the `hdfs cacheadmin` command.
- All the memory used for HDFS caching is separate from the `impalad` daemon address space and does not count towards the limits of the `--mem_limit` startup option, `MEM_LIMIT` query option, or further limits imposed through YARN resource management or the Linux `cgroups` mechanism.
- Because accessing HDFS cached data avoids a memory-to-memory copy operation, queries involving cached data require less memory on the Impala side than the equivalent queries on uncached data. In addition to any performance benefits in a single-user environment, the reduced memory helps to improve scalability under high-concurrency workloads.

Performance Considerations for HDFS Caching with Impala

In Impala 1.4.0 and higher, Impala supports efficient reads from data that is pinned in memory through HDFS caching. Impala takes advantage of the HDFS API and reads the data from memory rather than from disk whether the data files are pinned using Impala DDL statements, or using the command-line mechanism where you specify HDFS paths.

When you examine the output of the `impala-shell` `SUMMARY` command, or look in the metrics report for the `impalad` daemon, you see how many bytes are read from the HDFS cache. For example, this excerpt from a query profile illustrates that all the data read during a particular phase of the query came from the HDFS cache, because the `BytesRead` and `BytesReadDataNodeCache` values are identical.

```
HDFS_SCAN_NODE (id=0):(Total: 11s114ms, non-child: 11s114ms, % non-child: 100.00%)
        - AverageHdfsReadThreadConcurrency: 0.00
        - AverageScannerThreadConcurrency: 32.75
        - BytesRead: 10.47 GB (11240756479)
        - BytesReadDataNodeCache: 10.47 GB (11240756479)
        - BytesReadLocal: 10.47 GB (11240756479)
        - BytesReadShortCircuit: 10.47 GB (11240756479)
        - DecompressionTime: 27s572ms
```

For queries involving smaller amounts of data, or in single-user workloads, you might not notice a significant difference in query response time with or without HDFS caching. Even with HDFS caching turned off, the data for the query might still be in the Linux OS buffer cache. The benefits become clearer as data volume increases, and especially as the system processes more concurrent queries. HDFS caching improves the scalability of the overall system. That is, it prevents query performance from declining when the workload outstrips the capacity of the Linux OS cache.

Due to a limitation of HDFS, zero-copy reads are not supported with encryption. Cloudera recommends not using HDFS caching for Impala data files in encryption zones. The queries fall back to the normal read path during query execution, which might cause some performance overhead.

SELECT considerations:

The Impala HDFS caching feature interacts with the `SELECT` statement and query performance as follows:

- Impala automatically reads from memory any data that has been designated as cached and actually loaded into the HDFS cache. (It could take some time after the initial request to fully populate the cache for a table with large size or many partitions.) The speedup comes from two aspects: reading from RAM instead of disk, and accessing the data straight from the cache area instead of copying from one RAM area to another. This second aspect yields further performance improvement over the standard OS caching mechanism, which still results in memory-to-memory copying of cached data.
- For small amounts of data, the query speedup might not be noticeable in terms of wall clock time. The performance might be roughly the same with HDFS caching turned on or off, due to recently used data being held in the Linux OS cache. The difference is more pronounced with:
  - Data volumes (for all queries running concurrently) that exceed the size of the Linux OS cache.
  - A busy cluster running many concurrent queries, where the reduction in memory-to-memory copying and overall memory usage during queries results in greater scalability and throughput.
  - Thus, to really exercise and benchmark this feature in a development environment, you might need to simulate realistic workloads and concurrent queries that match your production environment.
  - One way to simulate a heavy workload on a lightly loaded system is to flush the OS buffer cache (on each DataNode) between iterations of queries against the same tables or partitions:

```
sync
echo 1 > /proc/sys/vm/drop_caches
```

- Impala queries take advantage of HDFS cached data regardless of whether the cache directive was issued by Impala or externally through the `hdfs cacheadmin` command, for example for an external table where the cached data files might be accessed by several different Hadoop components.
- If your query returns a large result set, the time reported for the query could be dominated by the time needed to print the results on the screen. To measure the time for the underlying query processing, query the `COUNT()` of the big result set, which does all the same processing but only prints a single line to the screen.

Detecting and Correcting HDFS Block Skew Conditions

For best performance of Impala parallel queries, the work is divided equally across hosts in the cluster, and all hosts take approximately equal time to finish their work. If one host takes substantially longer than others, the extra time needed for the slow host can become the dominant factor in query performance. Therefore, one of the first steps in performance tuning for Impala is to detect and correct such conditions.

The main cause of uneven performance that you can correct within Impala is ***skew*** in the number of HDFS data blocks processed by each host, where some hosts process substantially more data blocks than others. This condition can occur because of uneven distribution of the data values themselves, for example causing certain data files or partitions to be large while others are very small. (Although it is possible to have unevenly distributed data without any problems with the distribution of HDFS blocks.) Block skew could also be due to the underlying block allocation policies within HDFS, the replication factor of the data files, and the way that Impala chooses the host to process each data block.

The most convenient way to detect block skew, or slow-host issues in general, is to examine the "executive summary" information from the query profile after running a query:

- In `impala-shell`, issue the `SUMMARY` command immediately after the query is complete, to see just the summary information. If you detect issues involving skew, you might switch to issuing the `PROFILE` command, which displays the summary information followed by a detailed performance analysis.
- In the Cloudera Manager interface or the Impala debug web UI, click on the **Profile** link associated with the query after it is complete. The executive summary information is displayed early in the profile output.

For each phase of the query, you see an **Avg Time** and a **Max Time** value, along with **#Hosts** indicating how many hosts are involved in that query phase. For all the phases with **#Hosts** greater than one, look for cases where the maximum time is substantially greater than the average time. Focus on the phases that took the longest, for example, those taking multiple seconds rather than milliseconds or microseconds.

If you detect that some hosts take longer than others, first rule out non-Impala causes. One reason that some hosts could be slower than others is if those hosts have less capacity than the others, or if they are substantially busier due to unevenly distributed non-Impala workloads:

- For clusters running Impala, keep the relative capacities of all hosts roughly equal. Any cost savings from including some underpowered hosts in the cluster will likely be outweighed by poor or uneven performance, and the time spent diagnosing performance issues.
- If non-Impala workloads cause slowdowns on some hosts but not others, use the appropriate load-balancing techniques for the non-Impala components to smooth out the load across the cluster.

If the hosts on your cluster are evenly powered and evenly loaded, examine the detailed profile output to determine which host is taking longer than others for the query phase in question. Examine how many bytes are processed during that phase on that host, how much memory is used, and how many bytes are transmitted across the network.

The most common symptom is a higher number of bytes read on one host than others, due to one host being requested to process a higher number of HDFS data blocks. This condition is more likely to occur when the number of blocks accessed by the query is relatively small. For example, if you have a 10-node cluster and the query processes 10 HDFS blocks, each node might not process exactly one block. If one node sits idle while another node processes two blocks, the query could take twice as long as if the data was perfectly distributed.

Possible solutions in this case include:

- If the query is artificially small, perhaps for benchmarking purposes, scale it up to process a larger data set. For example, if some nodes read 10 HDFS data blocks while others read 11, the overall effect of the uneven distribution is much lower than when some nodes did twice as much work as others. As a guideline, aim for a "sweet spot" where each node reads 2 GB or more from HDFS per query. Queries that process lower volumes than that could experience inconsistent performance that smooths out as queries become more data-intensive.
- If the query processes only a few large blocks, so that many nodes sit idle and cannot help to parallelize the query, consider reducing the overall block size. For example, you might adjust the `PARQUET_FILE_SIZE` query option

before copying or converting data into a Parquet table. Or you might adjust the granularity of data files produced earlier in the ETL pipeline by non-Impala components. In Impala 2.0 and later, the default Parquet block size is 256 MB, reduced from 1 GB, to improve parallelism for common cluster sizes and data volumes.

- Reduce the amount of compression applied to the data. For text data files, the highest degree of compression (gzip) produces unsplittable files that are more difficult for Impala to process in parallel, and require extra memory during processing to hold the compressed and uncompressed data simultaneously. For binary formats such as Parquet and Avro, compression can result in fewer data blocks overall, but remember that when queries process relatively few blocks, there is less opportunity for parallel execution and many nodes in the cluster might sit idle. Note that when Impala writes Parquet data with the query option `COMPRESSION_CODEC=NONE` enabled, the data is still typically compact due to the encoding schemes used by Parquet, independent of the final compression step.

Data Cache for Remote Reads

When Impala compute nodes and its storage are not co-located, the network bandwidth requirement goes up as the network traffic includes the data fetch as well as the shuffling exchange traffic of intermediate results.



Note: This is an experimental feature in Impala 3.3 / CDH 6.3 and is not generally supported.

To mitigate the pressure on the network, you can enable the compute nodes to cache the working set read from remote filesystems, such as, remote HDFS data node, S3, ABFS, ADLS.

To enable remote data cache:

1. In Cloudera Manager, navigate to **Clusters** > **Impala Service** > **.**
2. In the **Configuration** tab, select **Impala Daemon** in Scope, and select **Advanced** in Category.
3. Enter the following in the **Impala Daemon Command Line Argument Advanced Configuration Snippet (Safety Valve)** field, set the `--data_cache` Impala Daemon start-up flag as below:

```
--data_cache=dir1,dir2,dir3,...:quota
```

   The flag is set to a list of directories, separated by `,`, followed by a `:`, and a capacity *`quota`* per directory.

   The directories must be specified in absolute path.

4. Click **Save Changes** and restart the Impala service.

If set to an empty string, data caching is disabled.

Cached data is stored in the specified directories.

The specified directories must exist in the local filesystem of each Impala Daemon, or Impala will fail to start.

In addition, the filesystem which the directory resides in must support hole punching.

The cache can consume up to the `quota` bytes for each of the directories specified.

The default setting for `--data_cache` is an empty string.

For example, with the following setting, the data cache may use up to 2 TB, with 1 TB max in `/data/0` and `/data/1` respectively.

```
--data_cache=/data/0,/data/1:1TB
```

Testing Impala Performance

Test to ensure that Impala is configured for optimal performance. If you have installed Impala without Cloudera Manager, complete the processes described in this topic to help ensure a proper configuration. Even if you installed Impala with Cloudera Manager, which automatically applies appropriate configurations, these procedures can be used to verify that Impala is set up correctly.

Checking Impala Configuration Values

You can inspect Impala configuration values by connecting to your Impala server using a browser.

To check Impala configuration values:

1. Use a browser to connect to one of the hosts running `impalad` in your environment. Connect using an address of the form `http://hostname:port/varz`.

   Note: In the preceding example, replace `hostname` and `port` with the name and port of your Impala server. The default port is 25000.

2. Review the configured values.

   For example, to check that your system is configured to use block locality tracking information, you would check that the value for `dfs.datanode.hdfs-blocks-metadata.enabled` is `true`.

To check data locality:

1. Execute a query on a dataset that is available across multiple nodes. For example, for a table named `MyTable` that has a reasonable chance of being spread across multiple DataNodes:

```
[impalad-host:21000] > SELECT COUNT (*) FROM MyTable
```

2. After the query completes, review the contents of the Impala logs. You should find a recent message similar to the following:

```
Total remote scan volume = 0
```

The presence of remote scans may indicate `impalad` is not running on the correct nodes. This can be because some DataNodes do not have `impalad` running or it can be because the `impalad` instance that is starting the query is unable to contact one or more of the `impalad` instances.

To understand the causes of this issue:

1. Connect to the debugging web server. By default, this server runs on port 25000. This page lists all `impalad` instances running in your cluster. If there are fewer instances than you expect, this often indicates some DataNodes are not running `impalad`. Ensure `impalad` is started on all DataNodes.
2. If you are using multi-homed hosts, ensure that the Impala daemon's hostname resolves to the interface on which `impalad` is running. The hostname Impala is using is displayed when `impalad` starts. To explicitly set the hostname, use the `--hostname` flag.
3. Check that `statestored` is running as expected. Review the contents of the state store log to ensure all instances of `impalad` are listed as having connected to the state store.

Reviewing Impala Logs

You can review the contents of the Impala logs for signs that short-circuit reads or block location tracking are not functioning. Before checking logs, execute a simple query against a small HDFS dataset. Completing a query task generates log messages using current settings. Information on executing queries can be found in Using the Impala

Shell (impala-shell Command) on page 713. Information on logging can be found in Using Impala Logging on page 709. Log messages and their interpretations are as follows:

Log Message	Interpretation
`Unknown disk id. This will negatively affect performance. Check your hdfs settings to enable block location metadata`	Tracking block locality is not enabled.
`Unable to load native-hadoop library for your platform... using builtin-java classes where applicable`	Native checksumming is not enabled.

Understanding Impala Query Performance - EXPLAIN Plans and Query Profiles

To understand the high-level performance considerations for Impala queries, read the output of the `EXPLAIN` statement for the query. You can get the `EXPLAIN` plan without actually running the query itself.

For an overview of the physical performance characteristics for a query, issue the `SUMMARY` statement in `impala-shell` immediately after executing a query. This condensed information shows which phases of execution took the most time, and how the estimates for memory usage and number of rows at each phase compare to the actual values.

To understand the detailed performance characteristics for a query, issue the `PROFILE` statement in `impala-shell` immediately after executing a query. This low-level information includes physical details about memory, CPU, I/O, and network usage, and thus is only available after the query is actually run.

Also, see Performance Considerations for the Impala-HBase Integration on page 685 and Understanding and Tuning Impala Query Performance for S3 Data on page 697 for examples of interpreting `EXPLAIN` plans for queries against HBase tables and data stored in the Amazon Simple Storage System (S3).

Using the EXPLAIN Plan for Performance Tuning

The `EXPLAIN` statement gives you an outline of the logical steps that a query will perform, such as how the work will be distributed among the nodes and how intermediate results will be combined to produce the final result set. You can see these details before actually running the query. You can use this information to check that the query will not operate in some very unexpected or inefficient way.

```
[impalad-host:21000] > explain select count(*) from customer_address;
+------------------------------------------------------------+
| Explain String                                             |
+------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=42.00MB VCores=1 |
|                                                            |
| 03:AGGREGATE [MERGE FINALIZE]                              |
| |  output: sum(count(*))                                   |
| |                                                          |
| 02:EXCHANGE [PARTITION=UNPARTITIONED]                      |
| |                                                          |
| 01:AGGREGATE                                               |
| |  output: count(*)                                        |
| |                                                          |
| 00:SCAN HDFS [default.customer_address]                    |
|    partitions=1/1 size=5.25MB                              |
+------------------------------------------------------------+
```

Read the `EXPLAIN` plan from bottom to top:

- The last part of the plan shows the low-level details such as the expected amount of data that will be read, where you can judge the effectiveness of your partitioning strategy and estimate how long it will take to scan a table based on total data size and the size of the cluster.
- As you work your way up, next you see the operations that will be parallelized and performed on each Impala node.
- At the higher levels, you see how data flows when intermediate result sets are combined and transmitted from one node to another.

- See EXPLAIN_LEVEL Query Option on page 331 for details about the EXPLAIN_LEVEL query option, which lets you customize how much detail to show in the EXPLAIN plan depending on whether you are doing high-level or low-level tuning, dealing with logical or physical aspects of the query.

The EXPLAIN plan is also printed at the beginning of the query profile report described in Using the Query Profile for Performance Tuning on page 604, for convenience in examining both the logical and physical aspects of the query side-by-side.

The amount of detail displayed in the EXPLAIN output is controlled by the EXPLAIN_LEVEL query option. You typically increase this setting from `standard` to `extended` (or from `1` to `2`) when doublechecking the presence of table and column statistics during performance tuning, or when estimating query resource usage in conjunction with the resource management features in CDH 5.

Using the SUMMARY Report for Performance Tuning

The SUMMARY command within the `impala-shell` interpreter gives you an easy-to-digest overview of the timings for the different phases of execution for a query. Like the EXPLAIN plan, it is easy to see potential performance bottlenecks. Like the PROFILE output, it is available after the query is run and so displays actual timing numbers.

The SUMMARY report is also printed at the beginning of the query profile report described in Using the Query Profile for Performance Tuning on page 604, for convenience in examining high-level and low-level aspects of the query side-by-side.

For example, here is a query involving an aggregate function, on a single-node VM. The different stages of the query and their timings are shown (rolled up for all nodes), along with estimated and actual values used in planning the query. In this case, the AVG( ) function is computed for a subset of data on each node (stage 01) and then the aggregated results from all nodes are combined at the end (stage 03). You can see which stages took the most time, and whether any estimates were substantially different than the actual data distribution. (When examining the time values, be sure to consider the suffixes such as `us` for microseconds and `ms` for milliseconds, rather than just looking for the largest numbers.)

```
[localhost:21000] > select avg(ss_sales_price) from store_sales where ss_coupon_amt =
0;
+----------------------+
| avg(ss_sales_price) |
+----------------------+
| 37.80770926328327   |
+----------------------+
[localhost:21000] > summary;
+-------------+-------+---------+---------+------+-----------+---------+--------------+----------------+
| Operator     | #Hosts | Avg Time | Max Time | #Rows | Est. #Rows | Peak Mem | Est.
Peak Mem | Detail          |
+-------------+-------+---------+---------+------+-----------+---------+--------------+----------------+
| 03:AGGREGATE | 1      | 1.03ms   | 1.03ms   | 1     | 1          | 48.00 KB | -1 B
        | MERGE FINALIZE  |
| 02:EXCHANGE  | 1      | 0ns      | 0ns      | 1     | 1          | 0 B      | -1 B
        | UNPARTITIONED   |
| 01:AGGREGATE | 1      | 30.79ms  | 30.79ms  | 1     | 1          | 80.00 KB | 10.00
MB      |                 |
| 00:SCAN HDFS | 1      | 5.45s    | 5.45s    | 2.21M | -1         | 64.05 MB | 432.00
 MB     | tpc.store_sales |
+-------------+-------+---------+---------+------+-----------+---------+--------------+----------------+
```

Notice how the longest initial phase of the query is measured in seconds (s), while later phases working on smaller intermediate results are measured in milliseconds (ms) or even nanoseconds (ns).

Here is an example from a more complicated query, as it would appear in the PROFILE output:

```
Operator              #Hosts   Avg Time   Max Time    #Rows  Est. #Rows  Peak Mem  Est.
 Peak Mem  Detail
------------------------------------------------------------------------------------------------------------------
09:MERGING-EXCHANGE        1   79.738us   79.738us        5           5         0
   -1.00 B  UNPARTITIONED
05:TOP-N                   3   84.693us   88.810us        5           5  12.00 KB
  120.00 B
```

```
04:AGGREGATE               3    5.263ms    6.432ms        5           5  44.00 KB
  10.00 MB  MERGE FINALIZE
08:AGGREGATE               3   16.659ms   27.444ms   52.52K     600.12K   3.20 MB
  15.11 MB  MERGE
07:EXCHANGE                3    2.644ms      5.1ms   52.52K     600.12K         0
         0  HASH(o_orderpriority)
03:AGGREGATE               3  342.913ms  966.291ms   52.52K     600.12K  10.80 MB
  15.11 MB
02:HASH JOIN               3    2s165ms    2s171ms  144.87K     600.12K  13.63 MB
 941.01 KB  INNER JOIN, BROADCAST
|--06:EXCHANGE             3    8.296ms    8.692ms   57.22K      15.00K         0
         0  BROADCAST
|  01:SCAN HDFS            2    1s412ms    1s978ms   57.22K      15.00K  24.21 MB
 176.00 MB  tpch.orders o
00:SCAN HDFS               3    8s032ms    8s558ms    3.79M     600.12K  32.29 MB
 264.00 MB  tpch.lineitem l
```

Using the Query Profile for Performance Tuning

The `PROFILE` command, available in the `impala-shell` interpreter, produces a detailed low-level report showing how the most recent query was executed. Unlike the `EXPLAIN` plan described in Using the EXPLAIN Plan for Performance Tuning on page 602, this information is only available after the query has finished. It shows physical details such as the number of bytes read, maximum memory usage, and so on for each node. You can use this information to determine if the query is I/O-bound or CPU-bound, whether some network condition is imposing a bottleneck, whether a slowdown is affecting some nodes but not others, and to check that recommended configuration settings such as short-circuit local reads are in effect.

By default, time values in the profile output reflect the wall-clock time taken by an operation. For values denoting system time or user time, the measurement unit is reflected in the metric name, such as `ScannerThreadsSysTime` or `ScannerThreadsUserTime`. For example, a multi-threaded I/O operation might show a small figure for wall-clock time, while the corresponding system time is larger, representing the sum of the CPU time taken by each thread. Or a wall-clock time figure might be larger because it counts time spent waiting, while the corresponding system and user time figures only measure the time while the operation is actively using CPU cycles.

The `EXPLAIN` plan is also printed at the beginning of the query profile report, for convenience in examining both the logical and physical aspects of the query side-by-side. The `EXPLAIN_LEVEL` query option also controls the verbosity of the `EXPLAIN` output printed by the `PROFILE` command.

In CDH 6.2 / Impala 3.2, a new `Per Node Profiles` section was added to the profile output. The new section includes the following metrics that can be controlled by the `RESOURCE_TRACE_RATIO` query option.

- `CpuIoWaitPercentage`
- `CpuSysPercentage`
- `CpuUserPercentage`
- `HostDiskReadThroughput`: All data read by the host as part of the execution of this query (spilling), by the HDFS data node, and by other processes running on the same system.
- `HostDiskWriteThroughput`: All data written by the host as part of the execution of this query (spilling), by the HDFS data node, and by other processes running on the same system.
- `HostNetworkRx`: All data received by the host as part of the execution of this query, other queries, and other processes running on the same system.
- `HostNetworkTx`: All data transmitted by the host as part of the execution of this query, other queries, and other processes running on the same system.

See Query Details for the steps to download the query profile in the text format in Cloudera Manager.

Scalability Considerations for Impala

This section explains how the size of your cluster and the volume of data influences SQL performance and schema design for Impala tables. Typically, adding more cluster capacity reduces problems due to memory limits or disk throughput. On the other hand, larger clusters are more likely to have other kinds of scalability issues, such as a single slow node that causes performance problems for queries.

A good source of tips related to scalability and performance tuning is the Impala Cookbook presentation. These slides are updated periodically as new features come out and new benchmarks are performed.

Impact of Many Tables or Partitions on Impala Catalog Performance and Memory Usage

Because Hadoop I/O is optimized for reading and writing large files, Impala is optimized for tables containing relatively few, large data files. Schemas containing thousands of tables, or tables containing thousands of partitions, can encounter performance issues during startup or during DDL operations such as `ALTER TABLE` statements.



Important:

Because of a change in the default heap size for the `catalogd` daemon in CDH 5.7 / Impala 2.5 and higher, the following procedure to increase the `catalogd` memory limit might be required following an upgrade to CDH 5.7 / Impala 2.5 even if not needed previously.

For schemas with large numbers of tables, partitions, and data files, the `catalogd` daemon might encounter an out-of-memory error. To prevent the error, increase the memory limit for the `catalogd` daemon:

1. Check current memory usage for the `catalogd` daemon by running the following commands on the host where that daemon runs on your cluster:

```
jcmd catalogd_pid VM.flags
jmap -heap catalogd_pid
```

2. Decide on a large enough value for the `catalogd` heap.
   - On systems managed by Cloudera Manager, include this value in the configuration field **Java Heap Size of Catalog Server in Bytes** (Cloudera Manager 5.7 and higher), or **Impala Catalog Server Environment Advanced Configuration Snippet (Safety Valve)** (prior to Cloudera Manager 5.7). Then restart the Impala service.
   - On systems not managed by Cloudera Manager, put the `JAVA_TOOL_OPTIONS` environment variable setting into the startup script for the `catalogd` daemon, then restart the `catalogd` daemon.

     For example, the following environment variable setting specifies the maximum heap size of 8 GB.

```
JAVA_TOOL_OPTIONS="-Xmx8g"
```

3. Use the same `jcmd` and `jmap` commands as earlier to verify that the new settings are in effect.

Scalability Consideration for Large Clusters

When processing queries, Impala daemons (`Impalads`) frequently exchange large volumes of data with other Impala daemons in the cluster, for example during a partitioned hash join. This communication between the Impala daemons

happens through remote procedure calls (RPCs). In CDH 5.14 / CDH 6.0 and lower, intercommunication was done exclusively using the Apache Thrift library. With Thrift RPC, the number of network connections per host sharply increases as the number of nodes goes up:

number of connections per host = (number of nodes) x (average number of query fragments per host)

For example, in a 100-node cluster with 32 concurrent queries, each of which had 50 fragments, there would be 100 x 32 x 50 = 160,000 connections and threads per host. Across the entire cluster, there would be 16 million connections. As more nodes are added to a CDH cluster due to increase in data and workloads, the excessive number of RPC threads and network connections can lead to instability and poor performance in Impala in CDH 5.14 / CDH 6.0 and lower.

In CDH 5.15.0 / CDH 6.1 and higher, Impala can use an alternate RPC option via KRPC that provides improvements in throughput and reliability while reducing the resource usage significantly. Using KRPC, Impala can reliably handle concurrent complex queries on large data sets. See this blog for detail information on KRPC for communications among Impala daemons.

If you are experiencing the scalability issue with connections but unable to upgrade to CDH 5.15 / CDH 6.1, these are some of the alternate options you can consider to alleviate the issues:

- Use dedicated coordinators.
- Isolate some workloads.
- Increase the status reporting interval by setting the start up flag `--status_report_interval` to a larger value. By default, it's set to 5 seconds.
- Use Workload Experience Manager analysis to identify any other improvements.

Scalability Considerations for the Impala Statestore

Before CDH 5.5 / Impala 2.3, the statestore sent only one kind of message to its subscribers. This message contained all updates for any topics that a subscriber had subscribed to. It also served to let subscribers know that the statestore had not failed, and conversely the statestore used the success of sending a heartbeat to a subscriber to decide whether or not the subscriber had failed.

Combining topic updates and failure detection in a single message led to bottlenecks in clusters with large numbers of tables, partitions, and HDFS data blocks. When the statestore was overloaded with metadata updates to transmit, heartbeat messages were sent less frequently, sometimes causing subscribers to time out their connection with the statestore. Increasing the subscriber timeout and decreasing the frequency of statestore heartbeats worked around the problem, but reduced responsiveness when the statestore failed or restarted.

As of CDH 5.5 / Impala 2.3, the statestore now sends topic updates and heartbeats in separate messages. This allows the statestore to send and receive a steady stream of lightweight heartbeats, and removes the requirement to send topic updates according to a fixed schedule, reducing statestore network overhead.

The statestore now has the following relevant configuration flags for the `statestored` daemon:

`-statestore_num_update_threads`

The number of threads inside the statestore dedicated to sending topic updates. You should not typically need to change this value.

Default: 10

`-statestore_update_frequency_ms`

The frequency, in milliseconds, with which the statestore tries to send topic updates to each subscriber. This is a best-effort value; if the statestore is unable to meet this frequency, it sends topic updates as fast as it can. You should not typically need to change this value.

Default: 2000

`-statestore_num_heartbeat_threads`

The number of threads inside the statestore dedicated to sending heartbeats. You should not typically need to change this value.

Default: 10

-statestore_heartbeat_frequency_ms

The frequency, in milliseconds, with which the statestore tries to send heartbeats to each subscriber. This value should be good for large catalogs and clusters up to approximately 150 nodes. Beyond that, you might need to increase this value to make the interval longer between heartbeat messages.

Default: 1000 (one heartbeat message every second)

As of CDH 5.5 / Impala 2.3 , not all of these flags are present in the Cloudera Manager user interface. Some must be set using the **Advanced Configuration Snippet** fields for the statestore component.

If it takes a very long time for a cluster to start up, and `impala-shell` consistently displays `This Impala daemon is not ready to accept user requests`, the statestore might be taking too long to send the entire catalog topic to the cluster. In this case, consider adding `--load_catalog_in_background=false` to your catalog service configuration. This setting stops the statestore from loading the entire catalog into memory at cluster startup. Instead, metadata for each table is loaded when the table is accessed for the first time.

Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher)

The buffer pool feature, available in CDH 5.13 and higher, changes the way Impala allocates memory during a query. Most of the memory needed is reserved at the beginning of the query, avoiding cases where a query might run for a long time before failing with an out-of-memory error. The actual memory estimates and memory buffers are typically smaller than before, so that more queries can run concurrently or process larger volumes of data than previously.

The buffer pool feature includes some query options that you can fine-tune: BUFFER_POOL_LIMIT Query Option on page 324, DEFAULT_SPILLABLE_BUFFER_SIZE Query Option on page 327, MAX_ROW_SIZE Query Option on page 340, and MIN_SPILLABLE_BUFFER_SIZE Query Option on page 345.

Most of the effects of the buffer pool are transparent to you as an Impala user. Memory use during spilling is now steadier and more predictable, instead of increasing rapidly as more data is spilled to disk. The main change from a user perspective is the need to increase the `MAX_ROW_SIZE` query option setting when querying tables with columns containing long strings, many columns, or other combinations of factors that produce very large rows. If Impala encounters rows that are too large to process with the default query option settings, the query fails with an error message suggesting to increase the `MAX_ROW_SIZE` setting.

SQL Operations that Spill to Disk

Certain memory-intensive operations write temporary data to disk (known as ***spilling*** to disk) when Impala is close to exceeding its memory limit on a particular host.



Note:

In CDH 5.13 and higher, also see Effect of Buffer Pool on Memory Usage (CDH 5.13 and higher) on page 607 for changes to Impala memory allocation that might change the details of which queries spill to disk, and how much memory and disk space is involved in the spilling operation.

The result is a query that completes successfully, rather than failing with an out-of-memory error. The tradeoff is decreased performance due to the extra disk I/O to write the temporary data and read it back in. The slowdown could be potentially be significant. Thus, while this feature improves reliability, you should optimize your queries, system parameters, and hardware configuration to make this spilling a rare occurrence.

What kinds of queries might spill to disk:

Several SQL clauses and constructs require memory allocations that could activat the spilling mechanism:

- when a query uses a `GROUP BY` clause for columns with millions or billions of distinct values, Impala keeps a similar number of temporary results in memory, to accumulate the aggregate results for each value in the group.

- When large tables are joined together, Impala keeps the values of the join columns from one table in memory, to compare them to incoming values from the other table.
- When a large result set is sorted by the ORDER BY clause, each node sorts its portion of the result set in memory.
- The DISTINCT and UNION operators build in-memory data structures to represent all values found so far, to eliminate duplicates as the query progresses.

When the spill-to-disk feature is activated for a join node within a query, Impala does not produce any runtime filters for that join operation on that host. Other join nodes within the query are not affected.

In CDH 5.13 / Impala 2.10 and higher, the way SQL operators such as GROUP BY, DISTINCT, and joins, transition between using additional memory or activating the spill-to-disk feature is changed. The memory required to spill to disk is reserved up front, and you can examine it in the EXPLAIN plan when the EXPLAIN_LEVEL query option is set to 2 or higher.

The infrastructure of the spilling feature affects the way the affected SQL operators, such as GROUP BY, DISTINCT, and joins, use memory. On each host that participates in the query, each such operator in a query requires memory to store rows of data and other data structures. Impala reserves a certain amount of memory up front for each operator that supports spill-to-disk that is sufficient to execute the operator. If an operator accumulates more data than can fit in the reserved memory, it can either reserve more memory to continue processing data in memory or start spilling data to temporary scratch files on disk. Thus, operators with spill-to-disk support can adapt to different memory constraints by using however much memory is available to speed up execution, yet tolerate low memory conditions by spilling data to disk.

The amount data depends on the portion of the data being handled by that host, and thus the operator may end up consuming different amounts of memory on different hosts.

How Impala handles scratch disk space for spilling:

By default, intermediate files used during large sort, join, aggregation, or analytic function operations are stored in the directory /tmp/impala-scratch . These files are removed when the operation finishes. (Multiple concurrent queries can perform operations that use the “spill to disk” technique, without any name conflicts for these temporary files.) You can specify a different location by starting the impalad daemon with the --scratch_dirs="*path_to_directory*" configuration option or the equivalent configuration option in the Cloudera Manager user interface. You can specify a single directory, or a comma-separated list of directories. The scratch directories must be on the local filesystem, not in HDFS. You might specify different directory paths for different hosts, depending on the capacity and speed of the available storage devices. In CDH 5.5 / Impala 2.3 or higher, Impala successfully starts (with a warning written to the log) if it cannot create or read and write files in one of the scratch directories. If there is less than 1 GB free on the filesystem where that directory resides, Impala still runs, but writes a warning message to its log. If Impala encounters an error reading or writing files in a scratch directory during a query, Impala logs the error and the query fails.

Memory usage for SQL operators:

In CDH 5.13 / Impala 2.10 and higher, the way SQL operators such as GROUP BY, DISTINCT, and joins, transition between using additional memory or activating the spill-to-disk feature is changed. The memory required to spill to disk is reserved up front, and you can examine it in the EXPLAIN plan when the EXPLAIN_LEVEL query option is set to 2 or higher.

The infrastructure of the spilling feature affects the way the affected SQL operators, such as GROUP BY, DISTINCT, and joins, use memory. On each host that participates in the query, each such operator in a query requires memory to store rows of data and other data structures. Impala reserves a certain amount of memory up front for each operator that supports spill-to-disk that is sufficient to execute the operator. If an operator accumulates more data than can fit in the reserved memory, it can either reserve more memory to continue processing data in memory or start spilling data to temporary scratch files on disk. Thus, operators with spill-to-disk support can adapt to different memory constraints by using however much memory is available to speed up execution, yet tolerate low memory conditions by spilling data to disk.

The amount of data depends on the portion of the data being handled by that host, and thus the operator may end up consuming different amounts of memory on different hosts.

Added in: This feature was added to the ORDER BY clause in CDH 5.1 / Impala 1.4. This feature was extended to cover join queries, aggregation functions, and analytic functions in CDH 5.2 / Impala 2.0. The size of the memory work area required by each operator that spills was reduced from 512 megabytes to 256 megabytes in CDH 5.4 / Impala 2.2. The spilling mechanism was reworked to take advantage of the Impala buffer pool feature and be more predictable and stable in CDH 5.13 / Impala 2.10.

Avoiding queries that spill to disk:

Because the extra I/O can impose significant performance overhead on these types of queries, try to avoid this situation by using the following steps:

1. Detect how often queries spill to disk, and how much temporary data is written. Refer to the following sources:
   - The output of the PROFILE command in the impala-shell interpreter. This data shows the memory usage for each host and in total across the cluster. The WriteIoBytes counter reports how much data was written to disk for each operator during the query. (In CDH 5.12 / Impala 2.9, the counter was named ScratchBytesWritten; in CDH 5.10 / Impala 2.8 and earlier, it was named BytesWritten.)
   - The **Impala Queries** dialog in Cloudera Manager. You can see the peak memory usage for a query, combined across all nodes in the cluster.
   - The **Queries** tab in the Impala debug web user interface. Select the query to examine and click the corresponding **Profile** link. This data breaks down the memory usage for a single host within the cluster, the host whose web interface you are connected to.
2. Use one or more techniques to reduce the possibility of the queries spilling to disk:
   - Increase the Impala memory limit if practical, for example, if you can increase the available memory by more than the amount of temporary data written to disk on a particular node. Remember that in Impala 2.0 and later, you can issue SET MEM_LIMIT as a SQL statement, which lets you fine-tune the memory usage for queries from JDBC and ODBC applications.
   - Increase the number of nodes in the cluster, to increase the aggregate memory available to Impala and reduce the amount of memory required on each node.
   - Add more memory to the hosts running Impala daemons.
   - On a cluster with resources shared between Impala and other Hadoop components, use resource management features to allocate more memory for Impala. See Resource Management on page 549 for details.
   - If the memory pressure is due to running many concurrent queries rather than a few memory-intensive ones, consider using the Impala admission control feature to lower the limit on the number of concurrent queries. By spacing out the most resource-intensive queries, you can avoid spikes in memory usage and improve overall response times. See Admission Control and Query Queuing on page 549 for details.
   - Tune the queries with the highest memory requirements, using one or more of the following techniques:
     - Run the COMPUTE STATS statement for all tables involved in large-scale joins and aggregation queries.
     - Minimize your use of STRING columns in join columns. Prefer numeric values instead.
     - Examine the EXPLAIN plan to understand the execution strategy being used for the most resource-intensive queries. See Using the EXPLAIN Plan for Performance Tuning on page 602 for details.
     - If Impala still chooses a suboptimal execution strategy even with statistics available, or if it is impractical to keep the statistics up to date for huge or rapidly changing tables, add hints to the most resource-intensive queries to select the right execution strategy. See Optimizer Hints in Impala on page 387 for details.
   - If your queries experience substantial performance overhead due to spilling, enable the DISABLE_UNSAFE_SPILLS query option. This option prevents queries whose memory usage is likely to be exorbitant from spilling to disk. See DISABLE_UNSAFE_SPILLS Query Option (CDH 5.2 or higher only) on page 329 for details. As you tune problematic queries using the preceding steps, fewer and fewer will be cancelled by this option setting.

Testing performance implications of spilling to disk:

To artificially provoke spilling, to test this feature and understand the performance implications, use a test environment with a memory limit of at least 2 GB. Issue the SET command with no arguments to check the current setting for the

MEM_LIMIT query option. Set the query option DISABLE_UNSAFE_SPILLS=true. This option limits the spill-to-disk feature to prevent runaway disk usage from queries that are known in advance to be suboptimal. Within impala-shell, run a query that you expect to be memory-intensive, based on the criteria explained earlier. A self-join of a large table is a good candidate:

```
select count(*) from big_table a join big_table b using (column_with_many_values);
```

Issue the PROFILE command to get a detailed breakdown of the memory usage on each node during the query.

Set the MEM_LIMIT query option to a value that is smaller than the peak memory usage reported in the profile output. Do not specify a memory limit lower than reported in the profile output. Now try the memory-intensive query again.

Check if the query fails with a message like the following:

```
WARNINGS: Spilling has been disabled for plans that do not have stats and are not hinted
to prevent potentially bad plans from using too many cluster resources. Compute stats
on
these tables, hint the plan or disable this behavior via query options to enable spilling.
```

If so, the query could have consumed substantial temporary disk space, slowing down so much that it would not complete in any reasonable time. Rather than rely on the spill-to-disk feature in this case, issue the COMPUTE STATS statement for the table or tables in your sample query. Then run the query again, check the peak memory usage again in the PROFILE output, and adjust the memory limit again if necessary to be lower than the peak memory usage.

At this point, you have a query that is memory-intensive, but Impala can optimize it efficiently so that the memory usage is not exorbitant. You have set an artificial constraint through the MEM_LIMIT option so that the query would normally fail with an out-of-memory error. But the automatic spill-to-disk feature means that the query should actually succeed, at the expense of some extra disk I/O to read and write temporary work data.

Try the query again, and confirm that it succeeds. Examine the PROFILE output again. This time, look for lines of this form:

```
- SpilledPartitions: N
```

If you see any such lines with *N* greater than 0, that indicates the query would have failed in Impala releases prior to 2.0, but now it succeeded because of the spill-to-disk feature. Examine the total time taken by the AGGREGATION_NODE or other query fragments containing non-zero SpilledPartitions values. Compare the times to similar fragments that did not spill, for example in the PROFILE output when the same query is run with a higher memory limit. This gives you an idea of the performance penalty of the spill operation for a particular query with a particular memory limit. If you make the memory limit just a little lower than the peak memory usage, the query only needs to write a small amount of temporary data to disk. The lower you set the memory limit, the more temporary data is written and the slower the query becomes.

Now repeat this procedure for actual queries used in your environment. Use the DISABLE_UNSAFE_SPILLS setting to identify cases where queries used more memory than necessary due to lack of statistics on the relevant tables and columns, and issue COMPUTE STATS where necessary.

When to use DISABLE_UNSAFE_SPILLS:

You might wonder, why not leave DISABLE_UNSAFE_SPILLS turned on all the time. Whether and how frequently to use this option depends on your system environment and workload.

DISABLE_UNSAFE_SPILLS is suitable for an environment with ad hoc queries whose performance characteristics and memory usage are not known in advance. It prevents “worst-case scenario” queries that use large amounts of memory unnecessarily. Thus, you might turn this option on within a session while developing new SQL code, even though it is turned off for existing applications.

Organizations where table and column statistics are generally up-to-date might leave this option turned on all the time, again to avoid worst-case scenarios for untested queries or if a problem in the ETL pipeline results in a table with no statistics. Turning on DISABLE_UNSAFE_SPILLS lets you “fail fast” in this case and immediately gather statistics or tune the problematic queries.

Some organizations might leave this option turned off. For example, you might have tables large enough that the `COMPUTE STATS` takes substantial time to run, making it impractical to re-run after loading new data. If you have examined the `EXPLAIN` plans of your queries and know that they are operating efficiently, you might leave `DISABLE_UNSAFE_SPILLS` turned off. In that case, you know that any queries that spill will not go overboard with their memory consumption.

Limits on Query Size and Complexity

There are hardcoded limits on the maximum size and complexity of queries. Currently, the maximum number of expressions in a query is 2000. You might exceed the limits with large or deeply nested queries produced by business intelligence tools or other query generators.

If you have the ability to customize such queries or the query generation logic that produces them, replace sequences of repetitive expressions with single operators such as `IN` or `BETWEEN` that can represent multiple values or ranges. For example, instead of a large number of `OR` clauses:

```
WHERE val = 1 OR val = 2 OR val = 6 OR val = 100 ...
```

use a single `IN` clause:

```
WHERE val IN (1,2,6,100,...)
```

Scalability Considerations for Impala I/O

Impala parallelizes its I/O operations aggressively, therefore the more disks you can attach to each host, the better. Impala retrieves data from disk so quickly using bulk read operations on large blocks, that most queries are CPU-bound rather than I/O-bound.

Because the kind of sequential scanning typically done by Impala queries does not benefit much from the random-access capabilities of SSDs, spinning disks typically provide the most cost-effective kind of storage for Impala data, with little or no performance penalty as compared to SSDs.

Resource management features such as YARN, Llama, and admission control typically constrain the amount of memory, CPU, or overall number of queries in a high-concurrency environment. Currently, there is no throttling mechanism for Impala I/O.

Scalability Considerations for Table Layout

Due to the overhead of retrieving and updating table metadata in the metastore database, try to limit the number of columns in a table to a maximum of approximately 2000. Although Impala can handle wider tables than this, the metastore overhead can become significant, leading to query performance that is slower than expected based on the actual data volume.

To minimize overhead related to the metastore database and Impala query planning, try to limit the number of partitions for any partitioned table to a few tens of thousands.

If the volume of data within a table makes it impractical to run exploratory queries, consider using the `TABLESAMPLE` clause to limit query processing to only a percentage of data within the table. This technique reduces the overhead for query startup, I/O to read the data, and the amount of network, CPU, and memory needed to process intermediate results during the query. See TABLESAMPLE Clause on page 315 for details.

Kerberos-Related Network Overhead for Large Clusters

When Impala starts up, or after each `kinit` refresh, Impala sends a number of simultaneous requests to the KDC. For a cluster with 100 hosts, the KDC might be able to process all the requests within roughly 5 seconds. For a cluster with

1000 hosts, the time to process the requests would be roughly 500 seconds. Impala also makes a number of DNS requests at the same time as these Kerberos-related requests.

While these authentication requests are being processed, any submitted Impala queries will fail. During this period, the KDC and DNS may be slow to respond to requests from components other than Impala, so other secure services might be affected temporarily.

In CDH 5.15 / Impala 2.12 or earlier, to reduce the frequency of the `kinit` renewal that initiates a new set of authentication requests, increase the `kerberos_reinit_interval` configuration setting for the `impalad` daemons. Currently, the default is 60 minutes. Consider using a higher value such as 360 (6 hours).

The `kerberos_reinit_interval` configuration setting is removed in CDH 6.0 / Impala 3.0, and the above step is no longer needed.

Avoiding CPU Hotspots for HDFS Cached Data

You can use the HDFS caching feature, described in Using HDFS Caching with Impala (CDH 5.3 or higher only) on page 593, with Impala to reduce I/O and memory-to-memory copying for frequently accessed tables or partitions.

In the early days of this feature, you might have found that enabling HDFS caching resulted in little or no performance improvement, because it could result in “hotspots”: instead of the I/O to read the table data being parallelized across the cluster, the I/O was reduced but the CPU load to process the data blocks might be concentrated on a single host.

To avoid hotspots, include the `WITH REPLICATION` clause with the `CREATE TABLE` or `ALTER TABLE` statements for tables that use HDFS caching. This clause allows more than one host to cache the relevant data blocks, so the CPU load can be shared, reducing the load on any one host. See CREATE TABLE Statement on page 234 and ALTER TABLE Statement on page 205 for details.

Hotspots with high CPU load for HDFS cached data could still arise in some cases, due to the way that Impala schedules the work of processing data blocks on different hosts. In CDH 5.7 / Impala 2.5 and higher, scheduling improvements mean that the work for HDFS cached data is divided better among all the hosts that have cached replicas for a particular data block. When more than one host has a cached replica for a data block, Impala assigns the work of processing that block to whichever host has done the least work (in terms of number of bytes read) for the current query. If hotspots persist even with this load-based scheduling algorithm, you can enable the query option `SCHEDULE_RANDOM_REPLICA=TRUE` to further distribute the CPU load. This setting causes Impala to randomly pick a host to process a cached data block if the scheduling algorithm encounters a tie when deciding which host has done the least work.

Scalability Considerations for File Handle Caching

One scalability aspect that affects heavily loaded clusters is the load on the metadata layer from looking up the details as each file is opened. On HDFS, that can lead to increased load on the NameNode, and on S3, this can lead to an excessive number of S3 metadata requests. For example, a query that does a full table scan on a partitioned table may need to read thousands of partitions, each partition containing multiple data files. Accessing each column of a Parquet file also involves a separate “open” call, further increasing the load on the NameNode. High NameNode overhead can add startup time (that is, increase latency) to Impala queries, and reduce overall throughput for non-Impala workloads that also require accessing HDFS files.

You can reduce the number of calls made to your file system's metadata layer by enabling the file handle caching feature. Data files that are accessed by different queries, or even multiple times within the same query, can be accessed without a new “open” call and without fetching the file details multiple times.

Impala supports file handle caching for the following file systems:

- HDFS in CDH 5.13 / Impala 2.10 and higher

  In Impala 3.2 and higher, file handle caching also applies to remote HDFS file handles. This is controlled by the `cache_remote_file_handles` flag for an `impalad`. It is recommended that you use the default value of `true` as this caching prevents your NameNode from overloading when your cluster has many remote HDFS reads.

- S3 in CDH 6.3 / Impala 3.3 and higher

  The `cache_s3_file_handles` `impalad` flag controls the S3 file handle caching. The feature is enabled by default with the flag set to `true`.

The feature is enabled by default with 20,000 file handles to be cached. To change the value, set the configuration option `max_cached_file_handles` to a non-zero value for each `impalad` daemon. From the initial default value of 20000, adjust upward if NameNode request load is still significant, or downward if it is more important to reduce the extra memory usage on each host. Each cache entry consumes 6 KB, meaning that caching 20,000 file handles requires up to 120 MB on each Impala executor. The exact memory usage varies depending on how many file handles have actually been cached; memory is freed as file handles are evicted from the cache.

If a manual operation moves a file to the trashcan while the file handle is cached, Impala still accesses the contents of that file. This is a change from prior behavior. Previously, accessing a file that was in the trashcan would cause an error. This behavior only applies to non-Impala methods of removing files, not the Impala mechanisms such as `TRUNCATE TABLE` or `DROP TABLE`.

If files are removed, replaced, or appended by operations outside of Impala, the way to bring the file information up to date is to run the `REFRESH` statement on the table.

File handle cache entries are evicted as the cache fills up, or based on a timeout period when they have not been accessed for some time.

To evaluate the effectiveness of file handle caching for a particular workload, issue the `PROFILE` statement in `impala-shell` or examine query profiles in the Impala Web UI. Look for the ratio of `CachedFileHandlesHitCount` (ideally, should be high) to `CachedFileHandlesMissCount` (ideally, should be low). Before starting any evaluation, run several representative queries to “warm up” the cache because the first time each data file is accessed is always recorded as a cache miss.

To see metrics about file handle caching for each `impalad` instance, examine the following fields on the **/metrics** page in the Impala Web UI:

- **impala-server.io.mgr.cached-file-handles-miss-count**
- **impala-server.io.mgr.num-cached-file-handles**

Scaling Limits and Guidelines

This topic lists the *scalability* limitation in Impala. For a given functional feature, it is recommended that you respect these limitations to achieve optimal scalability and performance. For example, while you might be able to create a table with 2000 columns, you will experience performance problems while querying the table. This topic does not cover functional limitations in Impala.

Unless noted otherwise, the limits were tested and certified.

The limits noted as "*generally safe*" are not certified, but recommended as generally safe. A safe range is not a hard limit as unforeseen errors or troubles in your particular environment can affect the range.

Deployment Limits

- Number of Impalad Executors
  - 80 nodes in CDH 5.14 and lower
  - 150 nodes in CDH 5.15 and higher
- Number of Impalad Coordinators: 1 coordinator for at most every 50 executors

  See Dedicated Coordinators for details.
- The number of Impala clusters per deployment
  - 1 Impala cluster in Impala 3.1 and lower
  - Multiple clusters in Impala 3.2 and higher is *generally safe*.

Data Storage Limits

There are no hard limits for the following, but you will experience gradual performance degradation as you increase these numbers.

- Number of databases
- Number of tables - total, per database
- Number of partitions - total, per table
- Number of files - total, per table, per table per partition
- Number of views - total, per database
- Number of user-defined functions - total, per database
- Parquet
  - Number of columns per row group
  - Number of row groups per block
  - Number of HDFS blocks per file

Schema Design Limits

- Number of columns
  - 300 for Kudu tables

    See Kudu Usage Limitations for more information.
  - 1000 for other types of tables

Security Limits

- Number of roles: 10,000 for Sentry

Query Limits - Compile Time

- Maximum number of columns in a query, included in a `SELECT` list, `INSERT`, and in an expression: no limit
- Number of tables referenced: no limit
- Number of plan nodes: no limit
- Number of plan fragments: no limit
- Depth of expression tree: 1000 hard limit
- Width of expression tree: 10,000 hard limit

Query Limits - Runtime Time

- Codegen
  - Very deeply nested expressions within queries can exceed internal Impala limits, leading to excessive memory usage. Setting the query option `disable_codegen=true` may reduce the impact, at a cost of longer query runtime.

How to Configure Impala with Dedicated Coordinators

Each host that runs the Impala Daemon acts as both a coordinator and as an executor, by default, managing metadata caching, query compilation, and query execution. In this configuration, Impala clients can connect to any Impala daemon and send query requests.

During highly concurrent workloads for large-scale queries, the dual roles can cause scalability issues because:

- The extra work required for a host to act as the coordinator could interfere with its capacity to perform other work for the later phases of the query. For example, coordinators can experience significant network and CPU

overhead with queries containing a large number of query fragments. Each coordinator caches metadata for all table partitions and data files, which requires coordinators to be configured with a large JVM heap. Executor-only Impala daemons should be configured with the default JVM heaps, which leaves more memory available to process joins, aggregations, and other operations performed by query executors.

- Having a large number of hosts act as coordinators can cause unnecessary network overhead, or even timeout errors, as each of those hosts communicates with the Statestored daemon for metadata updates.
- The "soft limits" imposed by the admission control feature are more likely to be exceeded when there are a large number of heavily loaded hosts acting as coordinators. Check IMPALA-3649 and IMPALA-6437 to see the status of the enhancements to mitigate this issue.

The following factors can further exacerbate the above issues:

- High number of concurrent query fragments due to query concurrency and/or query complexity
- Large metadata topic size related to the number of partitions/files/blocks
- High number of coordinator nodes
- High number of coordinators used in the same resource pool

If such scalability bottlenecks occur, in CDH 5.12 / Impala 2.9 and higher, you can assign one dedicated role to each Impala daemon host, either as a coordinator or as an executor, to address the issues.

- All explicit or load-balanced client connections must go to the coordinator hosts. These hosts perform the network communication to keep metadata up-to-date and route query results to the appropriate clients. The dedicated coordinator hosts do not participate in I/O-intensive operations such as scans, and CPU-intensive operations such as aggregations.
- The executor hosts perform the intensive I/O, CPU, and memory operations that make up the bulk of the work for each query. The executors do communicate with the Statestored daemon for membership status, but the dedicated executor hosts do not process the final result sets for queries.

Using dedicated coordinators offers the following benefits:

- Reduces memory usage by limiting the number of Impala nodes that need to cache metadata.
- Provides better concurrency by avoiding coordinator bottleneck.
- Eliminates query over-admission.
- Reduces resource, especially network, utilization on the Statestored daemon by limiting metadata broadcast to a subset of nodes.
- Improves reliability and performance for highly concurrent workloads by reducing workload stress on coordinators. Dedicated coordinators require 50% or fewer connections and threads.
- Reduces the number of explicit metadata refreshes required.
- Improves diagnosability if a bottleneck or other performance issue arises on a specific host, you can narrow down the cause more easily because each host is dedicated to specific operations within the overall Impala workload.

In this configuration with dedicated coordinators / executors, you cannot connect to the dedicated executor hosts through clients such as impala-shell or business intelligence tools as only the coordinator nodes support client connections.

Determining the Optimal Number of Dedicated Coordinators

You should have the smallest number of coordinators that will still satisfy your workload requirements in a cluster. A rough estimation is 1 coordinator for every 50 executors.

To maintain a healthy state and optimal performance, it is recommended that you keep the peak utilization of all resources used by Impala, including CPU, the number of threads, the number of connections, and RPCs, under 80%.

Consider the following factors to determine the right number of coordinators in your cluster:

- What is the number of concurrent queries?
- What percentage of the workload is DDL?
- What is the average query resource usage at the various stages (merge, runtime filter, result set size, etc.)?
- How many Impala Daemons (impalad) is in the cluster?
- Is there a high availability requirement?
- Compute/storage capacity reduction factor

Start with the below set of steps to determine the initial number of coordinators:

1. If your cluster has less than 10 nodes, we recommend that you configure one dedicated coordinator. Deploy the dedicated coordinator on a DataNode to avoid losing storage capacity. In most of the cases, one dedicated coordinator is enough to support all workloads on a cluster.
2. Add more coordinators if the dedicated coordinator CPU or network peak utilization is 80% or higher. You might need 1 coordinator for every 50 executors.
3. If the Impala service is shared by multiple workgroups with a dynamic resource pool assigned, use one coordinator per pool to avoid admission control over admission.
4. If high availability is required, double the number of coordinators. One set as an active set and the other as a backup set.

Advanced Tuning

Use the following guidelines to further tune the throughput and stability.

1. The concurrency of DML statements does not typically depend on the number of coordinators or size of the cluster. Queries that return large result sets (10,000+ rows) consume more CPU and memory resources on the coordinator. Add one or two coordinators if the workload has many such queries.
2. DDL queries, excluding `COMPUTE STATS` and `CREATE TABLE AS SELECT`, are executed only on coordinators. If your workload contains many DDL queries running concurrently, you could add one coordinator.
3. The CPU contention on coordinators can slow down query executions when concurrency is high, especially for very short queries (<10s). Add more coordinators to avoid CPU contention.
4. On a large cluster with 50+ nodes, the number of network connections from a coordinator to executors can grow quickly as query complexity increases. The growth is much greater on coordinators than executors. Add a few more coordinators if workloads are complex, i.e. (an average number of fragments * number of Impalad) > 500, but with the low memory/CPU usage to share the load. Watch IMPALA-4603 and IMPALA-7213 to track the progress on fixing this issue.
5. When using multiple coordinators for DML statements, divide queries to different groups (number of groups = number of coordinators). Configure a separate dynamic resource pool for each group and direct each group of query requests to a specific coordinator. This is to avoid query over admission.
6. The front-end connection requirement is not a factor in determining the number of dedicated coordinators. Consider setting up a connection pool at the client side instead of adding coordinators. For a short-term solution, you could increase the value of `fe_service_threads` on coordinators to allow more client connections.
7. In general, you should have a very small number of coordinators so storage capacity reduction is not a concern. On a very small cluster (less than 10 nodes), deploy a dedicated coordinator on a DataNode to avoid storage capacity reduction.

Estimating Coordinator Resource Usage

Resource	Safe range	Notes / CM tsquery to monitor
Memory	(Max JVM heap setting +	*Memory usage*:

	query concurrency * query mem_limit) <= 80% of Impala process memory allocation	SELECT mem_rss WHERE entityName = "Coordinator Instance ID" AND category = ROLE *JVM heap usage (metadata cache)*: SELECT impala_jvm_heap_current_usage_bytes WHERE entityName = "Coordinator Instance ID" AND category = ROLE (only in release 5.15 and above)
TCP Connection	Incoming + outgoing < 16K	*Incoming connection usage*: SELECT thrift_server_backend_connections_in_use WHERE entityName = "Coordinator Instance ID" AND category = ROLE *Outgoing connection usage*: SELECT backends_client_cache_clients_in_use WHERE entityName = "Coordinator Instance ID" AND category = ROLE
Threads	< 32K	SELECT thread_manager_running_threads WHERE entityName = "Coordinator Instance ID" AND category = ROLE
CPU	Concurrency = non-DDL query concurrency <= number of virtual cores allocated to Impala per node	CPU usage estimation should be based on how many cores are allocated to Impala per node, not a sum of all cores of the cluster. It is recommended that concurrency should not be more than the number of virtual cores allocated to Impala per node. *Query concurrency:* SELECT total_impala_num_queries_registered_across_impalads WHERE entityName = "IMPALA-1" AND category = SERVICE

If usage of any of the above resources exceeds the safe range, add one more coordinator.

Monitoring Coordinator Resource Usage

Using Cloudera Manager, monitor the coordinator resource usage to understand your workload and adjust the number of coordinators according to the guidelines above. The available options are:

- Impala **Queries** tab: Monitor such attributes as *DDL queries* and *Rows produced*. See Monitoring Impala Queries for detail information.
- Custom charts: Monitor activities, such as query complexity which is an average fragment count per query (total fragments / total queries).
- **tsquery**: Build the custom charts to monitor and estimate the amount of resource the coordinator needs. See tsquery Language for more information.

The following are sample queries for common resource usage monitoring. Replace `entityName` values with your coordinator instance id.

Per coordinator tsquery

Resource Usage	Tsquery
Memory usage	SELECT impala_memory_total_used, mem_tracker_process_limit WHERE entityName = "Coordinator Instance ID" AND category = ROLE
JVM heap usage (metadata cache)	SELECT impala_jvm_heap_current_usage_bytes WHERE entityName = "Coordinator Instance ID" AND category = ROLE (only in release 5.15 and above)
CPU usage	SELECT cpu_user_rate / getHostFact(numCores, 1) * 100, cpu_system_rate / getHostFact(numCores, 1) * 100 WHERE entityName="Coordinator Instance ID"
Network usage (host level)	SELECT total_bytes_receive_rate_across_network_interfaces, total_bytes_transmit_rate_across_network_interfaces WHERE entityName="Coordinator Instance ID"
Incoming connection usage	SELECT thrift_server_backend_connections_in_use WHERE entityName = "Coordinator Instance ID" AND category = ROLE
Outgoing connection usage	SELECT backends_client_cache_clients_in_use WHERE entityName = "Coordinator Instance ID" AND category = ROLE
Thread usage	SELECT thread_manager_running_threads WHERE entityName = "Coordinator Instance ID" AND category = ROLE

Cluster wide tsquery

Resource usage	Tsquery
Front-end connection usage	SELECT total_thrift_server_beeswax_frontend_connections_in_use_across_impalads, total_thrift_server_hiveserver2_frontend_connections_in_use_across_impalads
Query concurrency	SELECT total_impala_num_queries_registered_across_impalads WHERE entityName = "IMPALA-1" AND category = SERVICE

Deploying Dedicated Coordinators and Executors in Cloudera Manager

This section describes the process to configure a dedicated coordinator and a dedicated executor roles for Impala.

- **Dedicated coordinator:**
  - Should be on an edge node with no other services running on it.
  - Does not need large local disks but still needs some that can be used for Spilling.
  - Require at least the same or even larger memory allocation than executors.
- **(Dedicated)Executors:**
  - Should be colocated with DataNodes.
  - The number of hosts with dedicated executors typically increases as the cluster grows larger and handles more table partitions, data files, and concurrent queries.

To configuring dedicated coordinators/executors:

1. Navigate to **Clusters** > **Impala** > **Configuration** > **Role Groups**.
2. Click **Create** to create two role groups with the following values.
    a. Group for Coordinators
        a. **Group Name**: Coordinators
        b. **Role Type**: Impala Daemon
        c. **Copy from**:
            - Select **Impala Daemon Default Group** if you want the existing configuration gets carried over to the Coordinators.
            - Select **None** if you want to start with a blank configuration.
    b. Group for Executors
        a. **Group Name**: Executors
        b. **Role Type**: Impala Daemon
        c. **Copy from**:
            - Select **Impala Daemon Default Group** if you want the existing configuration gets carried over to the Executors.
            - Select **None** if you want to start with a blank configuration.
3. In the Role Groups page, click **Impala Daemon Default Group**.
    a. Select the set of nodes intended to be coordinators.
        a. Click **Action for Selected** and select **Move To Different Role Group…**.
        b. Select the **Coordinators**.
    b. Select the set of nodes intended to be Executors.
        a. Click **Action for Selected** and select **Move To Different Role Group…**.
        b. Select **Executors**.
4. Click **Configuration**. In the search field, type **Impala Daemon Specialization**.
5. Click **Edit Individual Values**.
6. For **Coordinators** role group, select **COORDINATOR_ONLY**.
7. For **Executors** role group, select **EXECUTOR_ONLY**.
8. Click **Save Changes** and then restart the Impala service.

Deploying Dedicated Coordinators and Executors from Command Line

To configuring dedicated coordinators/executors, you specify one of the following startup flags for the `impalad` daemon on each host:

- `--is_executor=false` for each host that does not act as an executor for Impala queries. These hosts act exclusively as query coordinators. This setting typically applies to a relatively small number of hosts, because the most common topology is to have nearly all DataNodes doing work for query execution.
- `--is_coordinator=false` for each host that does not act as a coordinator for Impala queries. These hosts act exclusively as executors. The number of hosts with this setting typically increases as the cluster grows larger and handles more table partitions, data files, and concurrent queries. As the overhead for query coordination increases, it becomes more important to centralize that work on dedicated hosts.

Metadata Management

This topic describes various knobs you can use to control how Impala manages its metadata in order to improve performance and scalability.

On-demand Metadata

In previous versions of Impala, every coordinator kept a replica of all the cache in `catalogd`, consuming large memory on each coordinator with no option to evict. Metadata always propagated through the `statestored` and suffers from head-of-line blocking, for example, one user loading a big table blocking another user loading a small table.

With this new feature, the coordinators pull metadata as needed from `catalogd` and cache it locally. The cached metadata gets evicted automatically under memory pressure.

The granularity of on-demand metadata fetches is now at the partition level between the coordinator and `catalogd`. Common use cases like add/drop partitions do not trigger unnecessary serialization/deserialization of large metadata.

This feature is disabled by default.

The feature can be used in either of the following modes.

Metadata on-demand mode

In this mode, all coordinators use the metadata on-demand.

Set the following on `catalogd`:

```
--catalog_topic_mode=minimal
```

Set the following on all `impalad` coordinators:

```
--use_local_catalog=true
```

Mixed mode

In this mode, only some coordinators are enabled to use the metadata on-demand.

We recommend that you use the mixed mode only for testing local catalog's impact on heap usage.

Set the following on `catalogd`:

```
--catalog_topic_mode=mixed
```

Set the following on `impalad` coordinators with metdadata on-demand:

```
--use_local_catalog=true
```

Limitation:

Global `INVALIDATES` are not supported when this feature is enabled. If your workload requires global `INVALIDATES`, do not use this feature.

Automatic Invalidation of Metadata Cache

To keep the size of metadata bounded, `catalogd` periodically scans all the tables and invalidates those not recently used. There are two types of configurations in `catalogd`.

Time-based cache invalidation

`Catalogd` invalidates tables that are not recently used in the specified time period (in seconds).

The `--invalidate_tables_timeout_s` flag needs to be applied to both `impalad` and `catalogd`.

Memory-based cache invalidation

When the memory pressure reaches 60% of JVM heap size after a Java garbage collection in `catalogd`, Impala invalidates 10% of the least recently used tables.

The `--invalidate_tables_on_memory_pressure` flag needs to be applied to both `impalad` and `catalogd`.

Automatic invalidation of metadata provides more stability with lower chances of running out of memory, but the feature could potentially cause performance issues and may require tuning.

Automatic Invalidation/Refresh of Metadata

When tools such as Hive and Spark are used to process the raw data ingested into Hive tables, new HMS metadata (database, tables, partitions) and filesystem metadata (new files in existing partitions/tables) is generated. In previous versions of Impala, in order to pick up this new information, Impala users needed to manually issue an `INVALIDATE` or `REFRESH` commands.

When automatic invalidate/refresh of metadata is enabled, `catalogd` polls Hive Metastore (HMS) notification events at a configurable interval and processes the following changes:



Note: This is a preview feature in CDH 6.3 / Impala 3.3 and not generally available.

- Invalidates the tables when it receives the `ALTER TABLE` event.
- Refreshes the partition when it receives the `ALTER`, `ADD`, or `DROP` partitions.
- Adds the tables or databases when it receives the `CREATE TABLE` or `CREATE DATABASE` events.
- Removes the tables from `catalogd` when it receives the `DROP TABLE` or `DROP DATABASE` events.
- Refreshes the table and partitions when it receives the `INSERT` events.

  If the table is not loaded at the time of processing the `INSERT` event, the event processor does not need to refresh the table and skips it.

This feature is controlled by the `--hms_event_polling_interval_s` flag. Start the `catalogd` with the `--hms_event_polling_interval_s` flag set to a positive integer to enable the feature and set the polling frequency in seconds. We recommend the value to be less than 5 seconds.

The following use cases are not supported:

- When you bypass HMS and add or remove data into table by adding files directly on the filesystem, HMS does not generate the `INSERT` event, and the event processor will not invalidate the corresponding table or refresh the corresponding partition.

  It is recommended that you use the `LOAD DATA` command to do the data load in such cases, so that event processor can act on the events generated by the `LOAD` command.

- The Spark API that saves data to a specified location does not generate events in HMS, thus is not supported. For example:

```
Seq((1, 2)).toDF("i",
"j").write.save("/user/hive/warehouse/spark_etl.db/customers/date=01012019")
```

This feature is turned off by default with the `--hms_event_polling_interval_s` flag set to `0`.

Configure HMS for Event Based Automatic Metadata Sync

As the first step to use the HMS event based metadata sync, enable and configure HMS notifications in Cloudera Manager.

1. Navigate to **Clusters** > **Hive** > **Configuration** > **Filters** > **SCOPE** > **Hive Metastore Server**.
2. Select **Enable Stored Notifications in Database**.
3. In **Hive Metastore Server Advanced Configuration Snippet (Safety Valve) for hive-site.xml**, click **+** to expand and enter the following:
   - **Name:** hive.metastore.notifications.add.thrift.objects
   - **Value:** true

- **Name:** hive.metastore.alter.notifications.basic
- **Value:** false

4. Click **Save Changes**.
5. If you want `INSERT` events are generated when the Spark (and other non-hive) applications insert data into **existing** tables and partitions:
   a. In **Hive Client Advanced Configuration Snippet (Safety Valve) for hive-site.xml**, click **+** to expand and enter the following:
      - **Name:** hive.metastore.dml.events
      - **Value:** true
   b. In **Hive Service Advanced Configuration Snippet (Safety Valve) for hive-site.xml**, click **+** to expand and enter the following:
      - **Name:** hive.metastore.dml.events
      - **Value:** true
   c. Click **Save Changes**.
6. Restart stale services.

Configure Catalog Server for Event-Based Automatic Metadata Sync

1. Navigate to **Clusters** > **Impala** > **Configuration** > **Filters** > **SCOPE** > **Impala Catalog Server**.
2. In **Catalog Server Advanced Configuration Snippet (Safety Valve)** , click **+** to expand and enter the following:
   - **Name:** `--hms_event_polling_interval_s`
   - **Value:** 4
3. Click **Save Changes**.
4. Restart the Catalog server when appropriate.

Disable Event Based Automatic Metadata Sync

When the `--hms_event_polling_interval_s` flag is set to a non-zero value for your `catalogd`, the event-based automatic invalidation is enabled for all databases and tables. If you wish to have the fine-grained control on which tables or databases need to be synced using events, you can use the `impala.disableHmsSync` property to disable the event processing at the table or database level.

When you add the `DBPROPERTIES` or `TBLPROPERTIES` with the `impala.disableHmsSync` key, the HMS event based sync is turned on or off. The value of the `impala.disableHmsSync` property determines if the event processing needs to be disabled for a particular table or database.

- If `'impala.disableHmsSync'='true'`, the events for that table or database are ignored and not synced with HMS.
- If `'impala.disableHmsSync'='false'` or if `impala.disableHmsSync` is not set, the automatic sync with HMS is enabled if the `--hms_event_polling_interval_s` global flag is set to non-zero.
- To disable the event based HMS sync for a new database, set the `impala.disableHmsSync` database properties in Hive as currently, Impala does not support setting database properties:

```
CREATE DATABASE <name> WITH DBPROPERTIES ('impala.disableHmsSync'='true');
```

- To enable or disable the event based HMS sync for a table:

```
CREATE TABLE <name> ... TBLPROPERTIES ('impala.disableHmsSync'='true' | 'false');
```

- To change the event based HMS sync at the table level:

```
ALTER TABLE <name> SET TBLPROPERTIES ('impala.disableHmsSync'='true' | 'false');
```

When both table and database level properties are set, the table level property takes precedence. If the table level property is not set, then the database level property is used to evaluate if the event needs to be processed or not.

If the property is changed from `true` (meaning events are skipped) to `false` (meaning events are not skipped), you need to issue a manual `INVALIDATE METADATA` command to reset event processor because it doesn't know how many events have been skipped in the past and cannot know if the object in the event is the latest. In such a case, the status of the event processor changes to `NEEDS_INVALIDATE`.

Metrics for Event Based Automatic Metadata Sync

You can use the web UI of the `catalogd` to check the state of the automatic invalidate event processor.

By default, the debug web UI of `catalogd` is at `http://impala-server-hostname:25020` (non-secure cluster) or `https://impala-server-hostname:25020` (secure cluster).

Under the web UI, there are two pages that presents the metrics for HMS event processor that is responsible for the event based automatic metadata sync.

- **/metrics#events**
- **/events**

  This provides a detailed view of the metrics of the event processor, including min, max, mean, median, of the durations and rate metrics for all the counters listed on the **/metrics#events** page.

/metrics#events Page

The **/metrics#events** page provides the following metrics about the HMS event processor.

Name	Description
events-processor. avg-events-fetch-duration	Average duration to fetch a batch of events and process it.
events-processor. avg-events-process-duration	Average time taken to process a batch of events received from the Metastore.
events-processor. events-received	Total number of the Metastore events received.
events-processor. events-received-15min-rate	Exponentially weighted moving average (EWMA) of number of events received in last 15 min. This rate of events can be used to determine if there are spikes in event processor activity during certain hours of the day.
events-processor. events-received-1min-rate	Exponentially weighted moving average (EWMA) of number of events received in last 1 min. This rate of events can be used to determine if there are spikes in event processor activity during certain hours of the day.
events-processor. events-received-5min-rate	Exponentially weighted moving average (EWMA) of number of events received in last 5 min. This rate of events can be used to determine if there are spikes in event processor activity during certain hours of the day.
events-processor. events-skipped	Total number of the Metastore events skipped. Events can be skipped based on certain flags are table and database level. You can use this metric to make decisions, such as: • If most of the events are being skipped, see if you might just turn off the event processing.

Name	Description
	• If most of the events are not skipped, see if you need to add flags on certain databases.
events-processor.status	Metastore event processor status to see if there are events being received or not. Possible states are: • `PAUSED` The event processor is paused because catalog is being reset concurrently. • `ACTIVE` The event processor is scheduled at a given frequency. • `ERROR` • The event processor is in error state and event processing has stopped. • `NEEDS_INVALIDATE` The event processor could not resolve certain events and needs a manual `INVALIDATE` command to reset the state. • `STOPPED` The event processing has been shutdown. No events will be processed. • `DISABLED` The event processor is not configured to run.

Partitioning for Impala Tables

By default, all the data files for a table are located in a single directory. Partitioning is a technique for physically dividing the data during loading, based on values from one or more columns, to speed up queries that test those columns. For example, with a `school_records` table partitioned on a `year` column, there is a separate data directory for each different year value, and all the data for that year is stored in a data file in that directory. A query that includes a `WHERE` condition such as `YEAR=1966`, `YEAR IN (1989,1999)`, or `YEAR BETWEEN 1984 AND 1989` can examine only the data files from the appropriate directory or directories, greatly reducing the amount of data to read and test.

See Attaching an External Partitioned Table to an HDFS Directory Structure on page 54 for an example that illustrates the syntax for creating partitioned tables, the underlying directory structure in HDFS, and how to attach a partitioned Impala external table to data files stored elsewhere in HDFS.

Parquet is a popular format for partitioned Impala tables because it is well suited to handle huge data volumes. See Query Performance for Impala Parquet Tables on page 645 for performance considerations for partitioned Parquet tables.

See NULL on page 170 for details about how `NULL` values are represented in partitioned tables.

See Using Impala with the Amazon S3 Filesystem on page 692 for details about setting up tables where some or all partitions reside on the Amazon Simple Storage Service (S3).

When to Use Partitioned Tables

Partitioning is typically appropriate for:

- Tables that are very large, where reading the entire data set takes an impractical amount of time.
- Tables that are always or almost always queried with conditions on the partitioning columns. In our example of a table partitioned by year, `SELECT COUNT(*) FROM school_records WHERE year = 1985` is efficient, only examining a small fraction of the data; but `SELECT COUNT(*) FROM school_records` has to process a separate data file for each year, resulting in more overall work than in an unpartitioned table. You would probably not partition this way if you frequently queried the table based on last name, student ID, and so on without testing the year.
- Columns that have reasonable cardinality (number of different values). If a column only has a small number of values, for example `Male` or `Female`, you do not gain much efficiency by eliminating only about 50% of the data to read for each query. If a column has only a few rows matching each value, the number of directories to process can become a limiting factor, and the data file in each directory could be too small to take advantage of the Hadoop mechanism for transmitting data in multi-megabyte blocks. For example, you might partition census data by year, store sales data by year and month, and web traffic data by year, month, and day. (Some users with high volumes of incoming data might even partition down to the individual hour and minute.)
- Data that already passes through an extract, transform, and load (ETL) pipeline. The values of the partitioning columns are stripped from the original data files and represented by directory names, so loading data into a partitioned table involves some sort of transformation or preprocessing.

SQL Statements for Partitioned Tables

In terms of Impala SQL syntax, partitioning affects these statements:

- `CREATE TABLE`: you specify a `PARTITIONED BY` clause when creating the table to identify names and data types of the partitioning columns. These columns are not included in the main list of columns for the table.
- In CDH 5.7 / Impala 2.5 and higher, you can also use the `PARTITIONED BY` clause in a `CREATE TABLE AS SELECT` statement. This syntax lets you use a single statement to create a partitioned table, copy data into it, and create new partitions based on the values in the inserted data.

- ALTER TABLE: you can add or drop partitions, to work with different portions of a huge data set. You can designate the HDFS directory that holds the data files for a specific partition. With data partitioned by date values, you might "age out" data that is no longer relevant.

  Note: If you are creating a partition for the first time and specifying its location, for maximum efficiency, use a single ALTER TABLE statement including both the ADD PARTITION and LOCATION clauses, rather than separate statements with ADD PARTITION and SET LOCATION clauses.

- INSERT: When you insert data into a partitioned table, you identify the partitioning columns. One or more values from each inserted row are not stored in data files, but instead determine the directory where that row value is stored. You can also specify which partition to load a set of data into, with INSERT OVERWRITE statements; you can replace the contents of a specific partition but you cannot append data to a specific partition.

  By default, if an INSERT statement creates any new subdirectories underneath a partitioned table, those subdirectories are assigned default HDFS permissions for the impala user. To make each subdirectory have the same permissions as its parent directory in HDFS, specify the --insert_inherit_permissions startup option for the impalad daemon.

- Although the syntax of the SELECT statement is the same whether or not the table is partitioned, the way queries interact with partitioned tables can have a dramatic impact on performance and scalability. The mechanism that lets queries skip certain partitions during a query is known as partition pruning; see Partition Pruning for Queries on page 627 for details.
- In Impala 1.4 and later, there is a SHOW PARTITIONS statement that displays information about each partition in a table. See SHOW Statement on page 363 for details.

Static and Dynamic Partitioning Clauses

Specifying all the partition columns in a SQL statement is called ***static partitioning***, because the statement affects a single predictable partition. For example, you use static partitioning with an ALTER TABLE statement that affects only one partition, or with an INSERT statement that inserts all values into the same partition:

```
insert into t1 partition(x=10, y='a') select c1 from some_other_table;
```

When you specify some partition key columns in an INSERT statement, but leave out the values, Impala determines which partition to insert. This technique is called ***dynamic partitioning***:

```
insert into t1 partition(x, y='b') select c1, c2 from some_other_table;
-- Create new partition if necessary based on variable year, month, and day; insert a
single value.
insert into weather partition (year, month, day) select 'cloudy',2014,4,21;
-- Create new partition if necessary for specified year and month but variable day;
insert a single value.
insert into weather partition (year=2014, month=04, day) select 'sunny',22;
```

The more key columns you specify in the PARTITION clause, the fewer columns you need in the SELECT list. The trailing columns in the SELECT list are substituted in order for the partition key columns with no specified value.

Refreshing a Single Partition

The REFRESH statement is typically used with partitioned tables when new data files are loaded into a partition by some non-Impala mechanism, such as a Hive or Spark job. The REFRESH statement makes Impala aware of the new data files so that they can be used in Impala queries. Because partitioned tables typically contain a high volume of data, the REFRESH operation for a full partitioned table can take significant time.

In CDH 5.9 / Impala 2.7 and higher, you can include a PARTITION (*partition_spec*) clause in the REFRESH statement so that only a single partition is refreshed. For example, REFRESH big_table PARTITION (year=2017,

month=9, day=30). The partition spec must include all the partition key columns. See REFRESH Statement on page 291 for more details and examples of REFRESH syntax and usage.

Permissions for Partition Subdirectories

By default, if an INSERT statement creates any new subdirectories underneath a partitioned table, those subdirectories are assigned default HDFS permissions for the impala user. To make each subdirectory have the same permissions as its parent directory in HDFS, specify the --insert_inherit_permissions startup option for the impalad daemon.

Partition Pruning for Queries

Partition pruning refers to the mechanism where a query can skip reading the data files corresponding to one or more partitions. If you can arrange for queries to prune large numbers of unnecessary partitions from the query execution plan, the queries use fewer resources and are thus proportionally faster and more scalable.

For example, if a table is partitioned by columns YEAR, MONTH, and DAY, then WHERE clauses such as WHERE year = 2013, WHERE year < 2010, or WHERE year BETWEEN 1995 AND 1998 allow Impala to skip the data files in all partitions outside the specified range. Likewise, WHERE year = 2013 AND month BETWEEN 1 AND 3 could prune even more partitions, reading the data files for only a portion of one year.

Checking if Partition Pruning Happens for a Query

To check the effectiveness of partition pruning for a query, check the EXPLAIN output for the query before running it. For example, this example shows a table with 3 partitions, where the query only reads 1 of them. The notation #partitions=1/3 in the EXPLAIN plan confirms that Impala can do the appropriate partition pruning.

```
[localhost:21000] > insert into census partition (year=2010) values ('Smith'),('Jones');
[localhost:21000] > insert into census partition (year=2011) values
('Smith'),('Jones'),('Doe');
[localhost:21000] > insert into census partition (year=2012) values ('Smith'),('Doe');
[localhost:21000] > select name from census where year=2010;
+-------+
| name  |
+-------+
| Smith |
| Jones |
+-------+
[localhost:21000] > explain select name from census where year=2010;
+----------------------------------------------------------------------+
| Explain String                                                       |
+----------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                      |
|   PARTITION: UNPARTITIONED                                           |
|                                                                      |
|   1:EXCHANGE                                                         |
|                                                                      |
| PLAN FRAGMENT 1                                                      |
|   PARTITION: RANDOM                                                  |
|                                                                      |
|   STREAM DATA SINK                                                   |
|     EXCHANGE ID: 1                                                   |
|     UNPARTITIONED                                                    |
|                                                                      |
|   0:SCAN HDFS                                                        |
|      table=predicate_propagation.census #partitions=1/3 size=12B     |
+----------------------------------------------------------------------+
```

For a report of the volume of data that was actually read and processed at each stage of the query, check the output of the SUMMARY command immediately after running the query. For a more detailed analysis, look at the output of the PROFILE command; it includes this same summary report near the start of the profile output.

What SQL Constructs Work with Partition Pruning

Impala can even do partition pruning in cases where the partition key column is not directly compared to a constant, by applying the transitive property to other parts of the `WHERE` clause. This technique is known as predicate propagation, and is available in Impala 1.2.2 and later. In this example, the census table includes another column indicating when the data was collected, which happens in 10-year intervals. Even though the query does not compare the partition key column (`YEAR`) to a constant value, Impala can deduce that only the partition `YEAR=2010` is required, and again only reads 1 out of 3 partitions.

```
[localhost:21000] > drop table census;
[localhost:21000] > create table census (name string, census_year int) partitioned by
(year int);
[localhost:21000] > insert into census partition (year=2010) values
('Smith',2010),('Jones',2010);
[localhost:21000] > insert into census partition (year=2011) values
('Smith',2020),('Jones',2020),('Doe',2020);
[localhost:21000] > insert into census partition (year=2012) values
('Smith',2020),('Doe',2020);
[localhost:21000] > select name from census where year = census_year and census_year=2010;
+-------+
| name  |
+-------+
| Smith |
| Jones |
+-------+
[localhost:21000] > explain select name from census where year = census_year and
census_year=2010;
+----------------------------------------------------------------------+
| Explain String                                                       |
+----------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                      |
|   PARTITION: UNPARTITIONED                                           |
|                                                                      |
|   1:EXCHANGE                                                         |
|                                                                      |
| PLAN FRAGMENT 1                                                      |
|   PARTITION: RANDOM                                                  |
|                                                                      |
|   STREAM DATA SINK                                                   |
|     EXCHANGE ID: 1                                                   |
|     UNPARTITIONED                                                    |
|                                                                      |
|   0:SCAN HDFS                                                        |
|      table=predicate_propagation.census #partitions=1/3 size=22B     |
|      predicates: census_year = 2010, year = census_year              |
+----------------------------------------------------------------------+
```

If a view applies to a partitioned table, any partition pruning considers the clauses on both the original query and any additional `WHERE` predicates in the query that refers to the view. Prior to Impala 1.4, only the `WHERE` clauses on the original query from the `CREATE VIEW` statement were used for partition pruning.

In queries involving both analytic functions and partitioned tables, partition pruning only occurs for columns named in the `PARTITION BY` clause of the analytic function call. For example, if an analytic function query has a clause such as `WHERE year=2016`, the way to make the query prune all other `YEAR` partitions is to include `PARTITION BY year` in the analytic function call; for example, `OVER (PARTITION BY year,`*`other_columns other_analytic_clauses`*`)`.

Dynamic Partition Pruning

The original mechanism uses to prune partitions is ***static partition pruning***, in which the conditions in the `WHERE` clause are analyzed to determine in advance which partitions can be safely skipped. In Impala 2.5 / CDH 5.7 and higher, Impala can perform ***dynamic partition pruning***, where information about the partitions is collected during the query, and Impala prunes unnecessary partitions in ways that were impractical to predict in advance.

For example, if partition key columns are compared to literal values in a WHERE clause, Impala can perform static partition pruning during the planning phase to only read the relevant partitions:

```
-- The query only needs to read 3 partitions whose key values are known ahead of time.
-- That's static partition pruning.
SELECT COUNT(*) FROM sales_table WHERE year IN (2005, 2010, 2015);
```

Dynamic partition pruning involves using information only available at run time, such as the result of a subquery. The following example shows a simple dynamic partition pruning.

```
CREATE TABLE yy (s STRING) PARTITIONED BY (year INT);
INSERT INTO yy PARTITION (year) VALUES ('1999', 1999), ('2000', 2000),
  ('2001', 2001), ('2010', 2010), ('2018', 2018);
COMPUTE STATS yy;

CREATE TABLE yy2 (s STRING, year INT);
INSERT INTO yy2 VALUES ('1999', 1999), ('2000', 2000), ('2001', 2001);
COMPUTE STATS yy2;

-- The following query reads an unknown number of partitions, whose key values
-- are only known at run time. The runtime filters line shows the
-- information used in query fragment 02 to decide which partitions to skip.

EXPLAIN SELECT s FROM yy WHERE year IN (SELECT year FROM yy2);
+-----------------------------------------------------------------------------+
| PLAN-ROOT SINK                                                              |
| |                                                                           |
| 04:EXCHANGE [UNPARTITIONED]                                                 |
| |                                                                           |
| 02:HASH JOIN [LEFT SEMI JOIN, BROADCAST]                                    |
| |  hash predicates: year = year                                             |
| |  runtime filters: RF000 <- year                                   |       |
| |                                                                           |
| |--03:EXCHANGE [BROADCAST]                                                  |
| |  |                                                                        |
| |  01:SCAN HDFS [default.yy2]                                               |
| |     partitions=1/1 files=1 size=620B                                      |
| |                                                                           |
| 00:SCAN HDFS [default.yy]                                                   |
|    partitions=5/5 files=5 size=1.71KB                             |       |
|    runtime filters: RF000 -> year                                           |
+-----------------------------------------------------------------------------+

SELECT s FROM yy WHERE year IN (SELECT year FROM yy2); -- Returns 3 rows from yy
PROFILE;
```

In the above example, Impala evaluates the subquery, sends the subquery results to all Impala nodes participating in the query, and then each impalad daemon uses the dynamic partition pruning optimization to read only the partitions with the relevant key values.

The output query plan from the EXPLAIN statement shows that runtime filters are enabled. The plan also shows that it expects to read all 5 partitions of the yy table, indicating that static partition pruning will not happen.

The Filter summary in the PROFILE output shows that the scan node filtered out based on a runtime filter of dynamic partition pruning.

```
Filter 0 (1.00 MB):
 - Files processed: 3
 - Files rejected: 1 (1)
 - Files total: 3 (3)
```

Dynamic partition pruning is especially effective for queries involving joins of several large partitioned tables. Evaluating the ON clauses of the join predicates might normally require reading data from all partitions of certain tables. If the WHERE clauses of the query refer to the partition key columns, Impala can now often skip reading many of the partitions while evaluating the ON clauses. The dynamic partition pruning optimization reduces the amount of I/O and the amount of intermediate data stored and transmitted across the network during the query.

When the spill-to-disk feature is activated for a join node within a query, Impala does not produce any runtime filters for that join operation on that host. Other join nodes within the query are not affected.

Dynamic partition pruning is part of the runtime filtering feature, which applies to other kinds of queries in addition to queries against partitioned tables. See Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588 for full details about this feature.

Partition Key Columns

The columns you choose as the partition keys should be ones that are frequently used to filter query results in important, large-scale queries. Popular examples are some combination of year, month, and day when the data has associated time values, and geographic region when the data is associated with some place.

- For time-based data, split out the separate parts into their own columns, because Impala cannot partition based on a `TIMESTAMP` column.
- The data type of the partition columns does not have a significant effect on the storage required, because the values from those columns are not stored in the data files, rather they are represented as strings inside HDFS directory names.
- In CDH 5.7 / Impala 2.5 and higher, you can enable the `OPTIMIZE_PARTITION_KEY_SCANS` query option to speed up queries that only refer to partition key columns, such as `SELECT MAX(year)`. This setting is not enabled by default because the query behavior is slightly different if the table contains partition directories without actual data inside. See OPTIMIZE_PARTITION_KEY_SCANS Query Option (CDH 5.7 or higher only) on page 348 for details.
- Partitioned tables can contain complex type columns. All the partition key columns must be scalar types.
- Remember that when Impala queries data stored in HDFS, it is most efficient to use multi-megabyte files to take advantage of the HDFS block size. For Parquet tables, the block size (and ideal size of the data files) is 256 MB in Impala 2.0 and later. Therefore, avoid specifying too many partition key columns, which could result in individual partitions containing only small amounts of data. For example, if you receive 1 GB of data per day, you might partition by year, month, and day; while if you receive 5 GB of data per minute, you might partition by year, month, day, hour, and minute. If you have data with a geographic component, you might partition based on postal code if you have many megabytes of data for each postal code, but if not, you might partition by some larger region such as city, state, or country. state

If you frequently run aggregate functions such as `MIN()`, `MAX()`, and `COUNT(DISTINCT)` on partition key columns, consider enabling the `OPTIMIZE_PARTITION_KEY_SCANS` query option, which optimizes such queries. This feature is available in CDH 5.7 / Impala 2.5 and higher. See OPTIMIZE_PARTITION_KEY_SCANS Query Option (CDH 5.7 or higher only) on page 348 for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Setting Different File Formats for Partitions

Partitioned tables have the flexibility to use different file formats for different partitions. (For background information about the different file formats Impala supports, see How Impala Works with Hadoop File Formats on page 634.) For example, if you originally received data in text format, then received new data in RCFile format, and eventually began receiving data in Parquet format, all that data could reside in the same table for queries. You just need to ensure that the table is structured so that the data files that use different file formats reside in separate partitions.

For example, here is how you might switch from text to Parquet data as you receive data for different years:

```
[localhost:21000] > create table census (name string) partitioned by (year smallint);
[localhost:21000] > alter table census add partition (year=2012); -- Text format;

[localhost:21000] > alter table census add partition (year=2013); -- Text format switches
 to Parquet before data loaded;
[localhost:21000] > alter table census partition (year=2013) set fileformat parquet;
```

```
[localhost:21000] > insert into census partition (year=2012) values
('Smith'),('Jones'),('Lee'),('Singh');
[localhost:21000] > insert into census partition (year=2013) values
('Flores'),('Bogomolov'),('Cooper'),('Appiah');
```

At this point, the HDFS directory for `year=2012` contains a text-format data file, while the HDFS directory for `year=2013` contains a Parquet data file. As always, when loading non-trivial data, you would use `INSERT ... SELECT` or `LOAD DATA` to import data in large batches, rather than `INSERT ... VALUES` which produces small files that are inefficient for real-world queries.

For other file types that Impala cannot create natively, you can switch into Hive and issue the `ALTER TABLE ... SET FILEFORMAT` statements and `INSERT` or `LOAD DATA` statements there. After switching back to Impala, issue a `REFRESH` *`table_name`* statement so that Impala recognizes any partitions or new data added through Hive.

Managing Partitions

You can add, drop, set the expected file format, or set the HDFS location of the data files for individual partitions within an Impala table. See ALTER TABLE Statement on page 205 for syntax details, and Setting Different File Formats for Partitions on page 630 for tips on managing tables containing partitions with different file formats.



Note: If you are creating a partition for the first time and specifying its location, for maximum efficiency, use a single `ALTER TABLE` statement including both the `ADD PARTITION` and `LOCATION` clauses, rather than separate statements with `ADD PARTITION` and `SET LOCATION` clauses.

What happens to the data files when a partition is dropped depends on whether the partitioned table is designated as internal or external. For an internal (managed) table, the data files are deleted. For example, if data in the partitioned table is a copy of raw data files stored elsewhere, you might save disk space by dropping older partitions that are no longer required for reporting, knowing that the original data is still available if needed later. For an external table, the data files are left alone. For example, dropping a partition without deleting the associated files lets Impala consider a smaller set of partitions, improving query efficiency and reducing overhead for DDL operations on the table; if the data is needed again later, you can add the partition again. See Overview of Impala Tables on page 196 for details and examples.

Using Partitioning with Kudu Tables

Kudu tables use a more fine-grained partitioning scheme than tables containing HDFS data files. You specify a `PARTITION BY` clause with the `CREATE TABLE` statement to identify how to divide the values from the partition key columns.

See Partitioning for Kudu Tables on page 675 for details and examples of the partitioning techniques for Kudu tables.

Keeping Statistics Up to Date for Partitioned Tables

Because the `COMPUTE STATS` statement can be resource-intensive to run on a partitioned table as new partitions are added, Impala includes a variation of this statement that allows computing statistics on a per-partition basis such that stats can be incrementally updated when new partitions are added.

Important:

For a particular table, use either `COMPUTE STATS` or `COMPUTE INCREMENTAL STATS`. The two kinds of stats do not interoperate with each other at the table level. Without dropping the stats, if you run `COMPUTE INCREMENTAL STATS` it will overwrite the full compute stats or if you run `COMPUTE STATS` it will drop all incremental stats for consistency.

When you run `COMPUTE INCREMENTAL STATS` on a table for the first time, the statistics are computed again from scratch regardless of whether the table already has statistics. Therefore, expect a one-time resource-intensive operation for scanning the entire table when running `COMPUTE INCREMENTAL STATS` for the first time on a given table.

In Impala 3.0 and lower, approximately 400 bytes of metadata per column per partition are needed for caching. Tables with a big number of partitions and many columns can add up to a significant memory overhead as the metadata must be cached on the `catalogd` host and on every `impalad` host that is eligible to be a coordinator. If this metadata for all tables exceeds 2 GB, you might experience service downtime. In Impala 3.1 and higher, the issue was alleviated with an improved handling of incremental stats.

The `COMPUTE INCREMENTAL STATS` variation computes statistics only for partitions that were added or changed since the last `COMPUTE INCREMENTAL STATS` statement, rather than the entire table. It is typically used for tables where a full `COMPUTE STATS` operation takes too long to be practical each time a partition is added or dropped. See Generating Table and Column Statistics on page 578 for full usage details.

```
-- Initially the table has no incremental stats, as indicated
-- 'false' under Incremental stats.
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | -1    | 1      | 223.74KB | NOT CACHED   | PARQUET | false
| Children    | -1    | 1      | 230.05KB | NOT CACHED   | PARQUET | false
| Electronics | -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home        | -1    | 1      | 232.56KB | NOT CACHED   | PARQUET | false
| Jewelry     | -1    | 1      | 223.72KB | NOT CACHED   | PARQUET | false
| Men         | -1    | 1      | 231.25KB | NOT CACHED   | PARQUET | false
| Music       | -1    | 1      | 237.90KB | NOT CACHED   | PARQUET | false
| Shoes       | -1    | 1      | 234.90KB | NOT CACHED   | PARQUET | false
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | -1    | 1      | 226.27KB | NOT CACHED   | PARQUET | false
| Total       | -1    | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------

-- After the first COMPUTE INCREMENTAL STATS,
-- all partitions have stats. The first
-- COMPUTE INCREMENTAL STATS scans the whole
-- table, discarding any previous stats from
-- a traditional COMPUTE STATS statement.
compute incremental stats item_partitioned;
+-------------------------------------------+
| summary                                   |
+-------------------------------------------+
| Updated 10 partition(s) and 21 column(s). |
+-------------------------------------------+
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
```

```
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------

-- Add a new partition...
alter table item_partitioned add partition (i_category='Camping');
-- Add or replace files in HDFS outside of Impala,
-- rendering the stats for a partition obsolete.
!import_data_into_sports_partition.sh
refresh item_partitioned;
drop incremental stats item_partitioned partition (i_category='Sports');
-- Now some partitions have incremental stats
-- and some do not.
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Camping     | -1    | 1      | 408.02KB | NOT CACHED   | PARQUET | false
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 11     | 2.65MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------

-- After another COMPUTE INCREMENTAL STATS,
-- all partitions have incremental stats, and only the 2
-- partitions without incremental stats were scanned.
compute incremental stats item_partitioned;
+------------------------------------------+
| summary                                  |
+------------------------------------------+
| Updated 2 partition(s) and 21 column(s). |
+------------------------------------------+
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+------------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incremental stats
+-------------+-------+--------+----------+--------------+---------+------------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Camping     | 5328  | 1      | 408.02KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 11     | 2.65MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------------------
```

How Impala Works with Hadoop File Formats

Impala supports several familiar file formats used in Apache Hadoop. Impala can load and query data files produced by other Hadoop components such as Spark, and data files produced by Impala can be used by other components also. The following sections discuss the procedures, limitations, and performance considerations for using each file format with Impala.

The file format used for an Impala table has significant performance consequences. Some file formats include compression support that affects the size of data on the disk and, consequently, the amount of I/O and CPU resources required to deserialize data. The amounts of I/O and CPU resources required can be a limiting factor in query performance since querying often begins with moving and decompressing data. To reduce the potential impact of this part of the process, data is often compressed. By compressing data, a smaller total number of bytes are transferred from disk to memory. This reduces the amount of time taken to transfer the data, but a tradeoff occurs when the CPU decompresses the content.

For the file formats that Impala cannot write to, create the table from within Impala whenever possible and insert data using another component such as Hive or Spark. See the table below for specific file formats.

The following table lists the file formats that Impala supports.

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Parquet	Structured	Snappy, gzip, zstd; currently Snappy by default	Yes.	Yes: `CREATE TABLE`, `INSERT`, `LOAD DATA`, and query.
ORC	Structured	gzip, Snappy, LZO, LZ4; currently gzip by default	The ORC support is an experimental feature since CDH 6.1 / Impala 3.1 & Impala 2.12. To disable it, set `--enable_orc_scanner` to `false` when starting the cluster.	No. Import data by using `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive followed by `REFRESH` *`table_name`* in Impala.
Text	Unstructured	LZO, gzip, bzip2, Snappy	Yes. For `CREATE TABLE` with no `STORED AS` clause, the default file format is uncompressed text, with values separated by ASCII `0x01` characters (typically represented as Ctrl-A).	Yes if uncompressed. No if compressed. If LZO compression is used, you must create the table and load data in Hive. If other kinds of compression are used, you must load data through `LOAD DATA`, Hive, or manually in HDFS.
Avro	Structured	Snappy, gzip, deflate	Yes, in Impala 1.4.0 and higher. In lower versions, create the table using Hive.	No. Import data by using `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive followed by `REFRESH` *`table_name`* in Impala.
RCFile	Structured	Snappy, gzip, deflate, bzip2	Yes.	No. Import data by using `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive followed by `REFRESH` *`table_name`* in Impala.

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
SequenceFile	Structured	Snappy, gzip, deflate, bzip2	Yes.	No. Import data by using LOAD DATA on data files already in the right format, or use INSERT in Hive followed by REFRESH *table_name* in Impala.

Impala supports the following compression codecs:

Snappy

Recommended for its effective balance between compression ratio and decompression speed. Snappy compression is very fast, but gzip provides greater space savings. Supported for text, RC, Sequence, and Avro files in Impala 2.0 and higher.

Gzip

Recommended when achieving the highest level of compression (and therefore greatest disk-space savings) is desired. Supported for text, RC, Sequence and Avro files in Impala 2.0 and higher.

Deflate

Not supported for text files.

Bzip2

Supported for text, RC, and Sequence files in Impala 2.0 and higher.

LZO

For text files only. Impala can query LZO-compressed text tables, but currently cannot create them or insert data into them. You need to perform these operations in Hive.

Zstd

For Parquet files only.

Choosing the File Format for a Table

Different file formats and compression codecs work better for different data sets. Choosing the proper format for your data can yield performance improvements. Use the following considerations to decide which combination of file format and compression to use for a particular table:

- If you are working with existing files that are already in a supported file format, use the same format for the Impala table if performance is acceptable. If the original format does not yield acceptable query performance or resource usage, consider creating a new Impala table with different file format or compression characteristics, and doing a one-time conversion by rewriting the data to the new table.
- Text files are convenient to produce through many different tools, and are human-readable for ease of verification and debugging. Those characteristics are why text is the default format for an Impala CREATE TABLE statement. However, when performance and resource usage are the primary considerations, use one of the structured file formats that include metadata and built-in compression.

  A typical workflow might involve bringing data into an Impala table by copying CSV or TSV files into the appropriate data directory, and then using the INSERT ... SELECT syntax to rewrite the data into a table using a different, more compact file format.

Using Text Data Files with Impala Tables

Impala supports using text files as the storage format for input and output. Text files are a convenient format to use for interchange with other applications or scripts that produce or read delimited text files, such as CSV or TSV with commas or tabs for delimiters.

Text files are also very flexible in their column definitions. For example, a text file could have more fields than the Impala table, and those extra fields are ignored during queries; or it could have fewer fields than the Impala table, and those missing fields are treated as `NULL` values in queries. You could have fields that were treated as numbers or timestamps in a table, then use `ALTER TABLE ... REPLACE COLUMNS` to switch them to strings, or the reverse.

Table 2: Text Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Text	Unstructured	LZO, gzip, bzip2, Snappy	Yes. For `CREATE TABLE` with no `STORED AS` clause, the default file format is uncompressed text, with values separated by ASCII `0x01` characters (typically represented as Ctrl-A).	Yes if uncompressed. No if compressed. If LZO compression is used, you must create the table and load data in Hive. If other kinds of compression are used, you must load data through `LOAD DATA`, Hive, or manually in HDFS.

Query Performance for Impala Text Tables

Data stored in text format is relatively bulky, and not as efficient to query as binary formats such as Parquet. You typically use text tables with Impala if that is the format you receive the data and you do not have control over that process, or if you are a relatively new Hadoop user and not familiar with techniques to generate files in other formats. (Because the default format for `CREATE TABLE` is text, you might create your first Impala tables as text without giving performance much thought.) Either way, look for opportunities to use more efficient file formats for the tables used in your most performance-critical queries.

For frequently queried data, you might load the original text data files into one Impala table, then use an `INSERT` statement to transfer the data to another table that uses the Parquet file format; the data is converted automatically as it is stored in the destination table.

For more compact data, consider using LZO compression for the text files. LZO is the only compression codec that Impala supports for text data, because the “splittable” nature of LZO data files lets different nodes work on different parts of the same file in parallel. See Using LZO-Compressed Text Files on page 640 for details.

In Impala 2.0 and later, you can also use text data compressed in the gzip, bzip2, or Snappy formats. Because these compressed formats are not “splittable” in the way that LZO is, there is less opportunity for Impala to parallelize queries on them. Therefore, use these types of compressed data only for convenience if that is the format in which you receive the data. Prefer to use LZO compression for text data if you have the choice, or convert the data to Parquet using an `INSERT ... SELECT` statement to copy the original data into a Parquet table.

Note:

Impala supports bzip files created by the `bzip2` command, but not bzip files with multiple streams created by the `pbzip2` command. Impala decodes only the data from the first part of such files, leading to incomplete results.

The maximum size that Impala can accommodate for an individual bzip file is 1 GB (after uncompression).

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Creating Text Tables

To create a table using text data files:

If the exact format of the text data files (such as the delimiter character) is not significant, use the `CREATE TABLE` statement with no extra clauses at the end to create a text-format table. For example:

```
create table my_table(id int, s string, n int, t timestamp, b boolean);
```

The data files created by any `INSERT` statements will use the Ctrl-A character (hex 01) as a separator between each column value.

A common use case is to import existing text files into an Impala table. The syntax is more verbose; the significant part is the `FIELDS TERMINATED BY` clause, which must be preceded by the `ROW FORMAT DELIMITED` clause. The statement can end with a `STORED AS TEXTFILE` clause, but that clause is optional because text format tables are the default. For example:

```
create table csv(id int, s string, n int, t timestamp, b boolean)
  row format delimited
  fields terminated by ',';

create table tsv(id int, s string, n int, t timestamp, b boolean)
  row format delimited
  fields terminated by '\t';

create table pipe_separated(id int, s string, n int, t timestamp, b boolean)
  row format delimited
  fields terminated by '|'
  stored as textfile;
```

You can create tables with specific separator characters to import text files in familiar formats such as CSV, TSV, or pipe-separated. You can also use these tables to produce output data files, by copying data into them through the `INSERT ... SELECT` syntax and then extracting the data files from the Impala data directory.

In Impala 1.3.1 and higher, you can specify a delimiter character `'\0'` to use the ASCII 0 (`nul`) character for text tables:

```
create table nul_separated(id int, s string, n int, t timestamp, b boolean)
  row format delimited
  fields terminated by '\0'
  stored as textfile;
```



Note:

Do not surround string values with quotation marks in text data files that you construct. If you need to include the separator character inside a field value, for example to put a string value with a comma inside a CSV-format data file, specify an escape character on the `CREATE TABLE` statement with the `ESCAPED BY` clause, and insert that character immediately before any separator characters that need escaping.

Issue a `DESCRIBE FORMATTED` *`table_name`* statement to see the details of how each table is represented internally in Impala.

Complex type considerations: Although you can create tables in this file format using the complex types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher, currently, Impala can query these types only in Parquet tables. The one exception to the preceding rule is COUNT(*) queries on RCFile tables that include complex types. Such queries are allowed in CDH 5.8 / Impala 2.6 and higher.

Data Files for Text Tables

When Impala queries a table with data in text format, it consults all the data files in the data directory for that table, with some exceptions:

- Impala ignores any hidden files, that is, files whose names start with a dot or an underscore.
- Impala queries ignore files with extensions commonly used for temporary work files by Hadoop tools. Any files with extensions `.tmp` or `.copying` are not considered part of the Impala table. The suffix matching is case-insensitive, so for example Impala ignores both `.copying` and `.COPYING` suffixes.
- Impala uses suffixes to recognize when text data files are compressed text. For Impala to recognize the compressed text files, they must have the appropriate file extension corresponding to the compression codec, either `.gz`, `.bz2`, or `.snappy`. The extensions can be in uppercase or lowercase.
- Otherwise, the file names are not significant. When you put files into an HDFS directory through ETL jobs, or point Impala to an existing HDFS directory with the `CREATE EXTERNAL TABLE` statement, or move data files under external control with the `LOAD DATA` statement, Impala preserves the original file names.

File names for data produced through Impala `INSERT` statements are given unique names to avoid file name conflicts.

An `INSERT ... SELECT` statement produces one data file from each node that processes the `SELECT` part of the statement. An `INSERT ... VALUES` statement produces a separate data file for each statement; because Impala is more efficient querying a small number of huge files than a large number of tiny files, the `INSERT ... VALUES` syntax is not recommended for loading a substantial volume of data. If you find yourself with a table that is inefficient due to too many small data files, reorganize the data into a few large files by doing `INSERT ... SELECT` to transfer the data to a new table.

Special values within text data files:

- Impala recognizes the literal strings `inf` for infinity and `nan` for "Not a Number", for `FLOAT` and `DOUBLE` columns.
- Impala recognizes the literal string `\N` to represent `NULL`. When using Sqoop, specify the options `--null-non-string` and `--null-string` to ensure all `NULL` values are represented correctly in the Sqoop output files. `\N` needs to be escaped as in the below example:

```
--null-string '\\N' --null-non-string '\\N'
```

- By default, Sqoop writes `NULL` values using the string `null`, which causes a conversion error when such rows are evaluated by Impala. A workaround for existing tables and data files is to change the table properties through `ALTER TABLE` *`name`* `SET TBLPROPERTIES("serialization.null.format"="null")`.
- In CDH 5.8 / Impala 2.6 and higher, Impala can optionally skip an arbitrary number of header lines from text input files on HDFS based on the `skip.header.line.count` value in the `TBLPROPERTIES` field of the table metadata. For example:

```
create table header_line(first_name string, age int)
  row format delimited fields terminated by ',';

-- Back in the shell, load data into the table with commands such as:
-- cat >data.csv
-- Name,Age
-- Alice,25
-- Bob,19
-- hdfs dfs -put data.csv /user/hive/warehouse/header_line

refresh header_line;

-- Initially, the Name,Age header line is treated as a row of the table.
```

```
select * from header_line limit 10;
+------------+------+
| first_name | age  |
+------------+------+
| Name       | NULL |
| Alice      | 25   |
| Bob        | 19   |
+------------+------+

alter table header_line set tblproperties('skip.header.line.count'='1');

-- Once the table property is set, queries skip the specified number of lines
-- at the beginning of each text data file. Therefore, all the files in the table
-- should follow the same convention for header lines.
select * from header_line limit 10;
+------------+-----+
| first_name | age |
+------------+-----+
| Alice      | 25  |
| Bob        | 19  |
+------------+-----+
```

Loading Data into Impala Text Tables

To load an existing text file into an Impala text table, use the `LOAD DATA` statement and specify the path of the file in HDFS. That file is moved into the appropriate Impala data directory.

To load multiple existing text files into an Impala text table, use the `LOAD DATA` statement and specify the HDFS path of the directory containing the files. All non-hidden files are moved into the appropriate Impala data directory.

To convert data to text from any other file format supported by Impala, use a SQL statement such as:

```
-- Text table with default delimiter, the hex 01 character.
CREATE TABLE text_table AS SELECT * FROM other_file_format_table;

-- Text table with user-specified delimiter. Currently, you cannot specify
-- the delimiter as part of CREATE TABLE LIKE or CREATE TABLE AS SELECT.
-- But you can change an existing text table to have a different delimiter.
CREATE TABLE csv LIKE other_file_format_table;
ALTER TABLE csv SET SERDEPROPERTIES ('serialization.format=',', 'field.delim=',');
INSERT INTO csv SELECT * FROM other_file_format_table;
```

This can be a useful technique to see how Impala represents special values within a text-format data file. Use the `DESCRIBE FORMATTED` statement to see the HDFS directory where the data files are stored, then use Linux commands such as `hdfs dfs -ls` *`hdfs_directory`* and `hdfs dfs -cat` *`hdfs_file`* to display the contents of an Impala-created text file.

To create a few rows in a text table for test purposes, you can use the `INSERT ... VALUES` syntax:

```
INSERT INTO text_table VALUES ('string_literal',100,hex('hello world'));
```



Note: Because Impala and the HDFS infrastructure are optimized for multi-megabyte files, avoid the `INSERT ... VALUES` notation when you are inserting many rows. Each `INSERT ... VALUES` statement produces a new tiny file, leading to fragmentation and reduced performance. When creating any substantial volume of new data, use one of the bulk loading techniques such as `LOAD DATA` or `INSERT ... SELECT`. Or, use an HBase table for single-row `INSERT` operations, because HBase tables are not subject to the same fragmentation issues as tables stored on HDFS.

When you create a text file for use with an Impala text table, specify `\N` to represent a `NULL` value. For the differences between `NULL` and empty strings, see NULL on page 170.

If a text file has fewer fields than the columns in the corresponding Impala table, all the corresponding columns are set to `NULL` when the data in that file is read by an Impala query.

If a text file has more fields than the columns in the corresponding Impala table, the extra fields are ignored when the data in that file is read by an Impala query.

You can also use manual HDFS operations such as `hdfs dfs -put` or `hdfs dfs -cp` to put data files in the data directory for an Impala table. When you copy or move new data files into the HDFS directory for the Impala table, issue a REFRESH *table_name* statement in `impala-shell` before issuing the next query against that table, to make Impala recognize the newly added files.

Using LZO-Compressed Text Files

Impala supports using text data files that employ LZO compression. Cloudera recommends compressing text data files when practical. Impala queries are usually I/O-bound; reducing the amount of data read from disk typically speeds up a query, despite the extra CPU work to uncompress the data in memory.

Impala can work with LZO-compressed text files. LZO-compressed files are preferable to text files compressed by other codecs, because LZO-compressed files are “splittable”, meaning that different portions of a file can be uncompressed and processed independently by different nodes.

Impala does not currently support writing LZO-compressed text files.

Because Impala can query LZO-compressed files but currently cannot write them, you use Hive to do the initial CREATE TABLE and load the data, then switch back to Impala to run queries. For instructions on setting up LZO compression for Hive CREATE TABLE and INSERT statements, see the LZO page on the Hive wiki. Once you have created an LZO text table, you can also manually add LZO-compressed text files to it, produced by the `lzop` command or similar method.

Preparing to Use LZO-Compressed Text Files

Before using LZO-compressed tables in Impala, do the following one-time setup for each machine in the cluster. Install the necessary packages using either the Cloudera public repository, a private repository you establish, or by using packages. You must do these steps manually, whether or not the cluster is managed by the Cloudera Manager product.

1. **Prepare your systems to work with LZO by downloading and installing the appropriate libraries:**

   On systems managed by Cloudera Manager using parcels:

   See the setup instructions for the LZO parcel in the Cloudera Manager documentation for Cloudera Manager.

2. **Configure Impala to use LZO:**

   Use **one** of the following sets of commands to refresh your package management system's repository information, install the base LZO support for Hadoop, and install the LZO support for Impala.

   For RHEL/CentOS systems:

```
sudo yum update
sudo yum install hadoop-lzo
sudo yum install impala-lzo
```

   For SUSE systems:

```
$ sudo apt-get update
$ sudo zypper install hadoop-lzo
$ sudo zypper install impala-lzo
```

   For Debian/Ubuntu systems:

```
sudo zypper update
sudo apt-get install hadoop-lzo
sudo apt-get install impala-lzo
```



Note:

The level of the `impala-lzo` package is closely tied to the version of Impala you use. Any time you upgrade Impala, re-do the installation command for `impala-lzo` on each applicable machine to make sure you have the appropriate version of that package.

3. For `core-site.xml` on the client **and** server (that is, in the configuration directories for both Impala and Hadoop), append `com.hadoop.compression.lzo.LzopCodec` to the comma-separated list of codecs. For example:

```
<property>
  <name>io.compression.codecs</name>

<value>org.apache.hadoop.io.compress.DefaultCodec,org.apache.hadoop.io.compress.GzipCodec,


org.apache.hadoop.io.compress.BZip2Codec,org.apache.hadoop.io.compress.DeflateCodec,

org.apache.hadoop.io.compress.SnappyCodec,com.hadoop.compression.lzo.LzopCodec</value>
</property>
```



Note:

If this is the first time you have edited the Hadoop `core-site.xml` file, note that the `/etc/hadoop/conf` directory is typically a symbolic link, so the canonical `core-site.xml` might reside in a different directory:

```
$ ls -l /etc/hadoop
total 8
lrwxrwxrwx. 1 root root   29 Feb 26  2013 conf ->
/etc/alternatives/hadoop-conf
lrwxrwxrwx. 1 root root   10 Feb 26  2013 conf.dist -> conf.empty
drwxr-xr-x. 2 root root 4096 Feb 26  2013 conf.empty
drwxr-xr-x. 2 root root 4096 Oct 28 15:46 conf.pseudo
```

If the `io.compression.codecs` property is missing from `core-site.xml`, only add `com.hadoop.compression.lzo.LzopCodec` to the new property value, not all the names from the preceding example.

4. Restart the MapReduce and Impala services.

Creating LZO Compressed Text Tables

A table containing LZO-compressed text files must be created in Hive with the following storage clause:

```
STORED AS
    INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'
    OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
```

Also, certain Hive settings need to be in effect. For example:

```
hive> SET mapreduce.output.fileoutputformat.compress=true;
hive> SET hive.exec.compress.output=true;
hive> SET
mapreduce.output.fileoutputformat.compress.codec=com.hadoop.compression.lzo.LzopCodec;
hive> CREATE TABLE lzo_t (s string) STORED AS
  > INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'
  > OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';
hive> INSERT INTO TABLE lzo_t SELECT col1, col2 FROM uncompressed_text_table;
```

Once you have created LZO-compressed text tables, you can convert data stored in other tables (regardless of file format) by using the `INSERT ... SELECT` statement in Hive.

Files in an LZO-compressed table must use the `.lzo` extension. Examine the files in the HDFS data directory after doing the `INSERT` in Hive, to make sure the files have the right extension. If the required settings are not in place, you end up with regular uncompressed files, and Impala cannot access the table because it finds data files with the wrong (uncompressed) format.

After loading data into an LZO-compressed text table, index the files so that they can be split. You index the files by running a Java class, `com.hadoop.compression.lzo.DistributedLzoIndexer`, through the Linux command line. This Java class is included in the `hadoop-lzo` package.

Run the indexer using a command like the following:

```
$ hadoop jar /usr/lib/hadoop/lib/hadoop-lzo-version-gplextras.jar
  com.hadoop.compression.lzo.DistributedLzoIndexer /hdfs_location_of_table/
```



Note: If the path of the JAR file in the preceding example is not recognized, do a `find` command to locate `hadoop-lzo-*-gplextras.jar` and use that path.

Indexed files have the same name as the file they index, with the `.index` extension. If the data files are not indexed, Impala queries still work, but the queries read the data from remote DataNodes, which is very inefficient.

Once the LZO-compressed tables are created, and data is loaded and indexed, you can query them through Impala. As always, the first time you start `impala-shell` after creating a table in Hive, issue an `INVALIDATE METADATA` statement so that Impala recognizes the new table. (In Impala 1.2 and higher, you only have to run `INVALIDATE METADATA` on one node, rather than on all the Impala nodes.)

Using gzip, bzip2, or Snappy-Compressed Text Files

In Impala 2.0 and later, Impala supports using text data files that employ gzip, bzip2, or Snappy compression. These compression types are primarily for convenience within an existing ETL pipeline rather than maximum performance. Although it requires less I/O to read compressed text than the equivalent uncompressed text, files compressed by these codecs are not “splittable” and therefore cannot take full advantage of the Impala parallel query capability.

As each bzip2- or Snappy-compressed text file is processed, the node doing the work reads the entire file into memory and then decompresses it. Therefore, the node must have enough memory to hold both the compressed and uncompressed data from the text file. The memory required to hold the uncompressed data is difficult to estimate in advance, potentially causing problems on systems with low memory limits or with resource management enabled. In Impala 2.1 and higher, this memory overhead is reduced for gzip-compressed text files. The gzipped data is decompressed as it is read, rather than all at once.

To create a table to hold gzip, bzip2, or Snappy-compressed text, create a text table with no special compression options. Specify the delimiter and escape character if required, using the `ROW FORMAT` clause.

Because Impala can query compressed text files but currently cannot write them, produce the compressed text files outside Impala and use the `LOAD DATA` statement, manual HDFS commands to move them to the appropriate Impala data directory. (Or, you can use `CREATE EXTERNAL TABLE` and point the `LOCATION` attribute at a directory containing existing compressed text files.)

For Impala to recognize the compressed text files, they must have the appropriate file extension corresponding to the compression codec, either `.gz`, `.bz2`, or `.snappy`. The extensions can be in uppercase or lowercase.

The following example shows how you can create a regular text table, put different kinds of compressed and uncompressed files into it, and Impala automatically recognizes and decompresses each one based on their file extensions:

```
create table csv_compressed (a string, b string, c string)
  row format delimited fields terminated by ",";

insert into csv_compressed values
  ('one - uncompressed', 'two - uncompressed', 'three - uncompressed'),
  ('abc - uncompressed', 'xyz - uncompressed', '123 - uncompressed');
...make equivalent .gz, .bz2, and .snappy files and load them into same table directory...
```

```
select * from csv_compressed;
+--------------------+--------------------+----------------------+
| a                  | b                  | c                    |
+--------------------+--------------------+----------------------+
| one - snappy       | two - snappy       | three - snappy       |
| one - uncompressed | two - uncompressed | three - uncompressed |
| abc - uncompressed | xyz - uncompressed | 123 - uncompressed   |
| one - bz2          | two - bz2          | three - bz2          |
| abc - bz2          | xyz - bz2          | 123 - bz2            |
| one - gzip         | two - gzip         | three - gzip         |
| abc - gzip         | xyz - gzip         | 123 - gzip           |
+--------------------+--------------------+----------------------+

$ hdfs dfs -ls
'hdfs://127.0.0.1:8020/user/hive/warehouse/file_formats.db/csv_compressed/';
...truncated for readability...
75
hdfs://127.0.0.1:8020/user/hive/warehouse/file_formats.db/csv_compressed/csv_compressed.snappy
79
hdfs://127.0.0.1:8020/user/hive/warehouse/file_formats.db/csv_compressed/csv_compressed_bz2.csv.bz2
80
hdfs://127.0.0.1:8020/user/hive/warehouse/file_formats.db/csv_compressed/csv_compressed_gzip.csv.gz
116
hdfs://127.0.0.1:8020/user/hive/warehouse/file_formats.db/csv_compressed/dd414df64d67d49b_data.0.
```

Using the Parquet File Format with Impala Tables

Impala allows you to create, manage, and query Parquet tables. Parquet is a column-oriented binary file format intended to be highly efficient for the types of large-scale queries that Impala is best at. Parquet is especially good for queries scanning particular columns within a table, for example, to query “wide” tables with many columns, or to perform aggregation operations such as SUM( ) and AVG( ) that need to process most or all of the values from a column. Each Parquet data file written by Impala contains the values for a set of rows (referred to as the “row group”). Within a data file, the values from each column are organized so that they are all adjacent, enabling good compression for the values from that column. Queries against a Parquet table can retrieve and analyze these values from any column quickly and with minimal I/O.

See How Impala Works with Hadoop File Formats on page 634 for the summary of Parquet format support.

Creating Parquet Tables in Impala

To create a table named PARQUET_TABLE that uses the Parquet format, you would use a command like the following, substituting your own table name, column names, and data types:

```
[impala-host:21000] > create table parquet_table_name (x INT, y STRING) STORED AS PARQUET;
```

Or, to clone the column names and data types of an existing table:

```
[impala-host:21000] > create table parquet_table_name LIKE other_table_name STORED AS
PARQUET;
```

In Impala 1.4.0 and higher, you can derive column definitions from a raw Parquet data file, even without an existing Impala table. For example, you can create an external table pointing to an HDFS directory, and base the column definitions on one of the files in that directory:

```
CREATE EXTERNAL TABLE ingest_existing_files LIKE PARQUET
'/user/etl/destination/datafile1.dat'
  STORED AS PARQUET
  LOCATION '/user/etl/destination';
```

Or, you can refer to an existing data file and create a new empty table with suitable column definitions. Then you can use INSERT to create new data files or LOAD DATA to transfer existing data files into the new table.

```
CREATE TABLE columns_from_data_file LIKE PARQUET '/user/etl/destination/datafile1.dat'
  STORED AS PARQUET;
```

The default properties of the newly created table are the same as for any other CREATE TABLE statement. For example, the default file format is text; if you want the new table to use the Parquet file format, include the STORED AS PARQUET file also.

In this example, the new table is partitioned by year, month, and day. These partition key columns are not part of the data file, so you specify them in the CREATE TABLE statement:

```
CREATE TABLE columns_from_data_file LIKE PARQUET '/user/etl/destination/datafile1.dat'
  PARTITION (year INT, month TINYINT, day TINYINT)
  STORED AS PARQUET;
```

See CREATE TABLE Statement on page 234 for more details about the CREATE TABLE LIKE PARQUET syntax.

Once you have created a table, to insert data into that table, use a command similar to the following, again with your own table names:

```
[impala-host:21000] > insert overwrite table parquet_table_name select * from
other_table_name;
```

If the Parquet table has a different number of columns or different column names than the other table, specify the names of columns from the other table rather than * in the SELECT statement.

Loading Data into Parquet Tables

Choose from the following techniques for loading data into Parquet tables, depending on whether the original data is already in an Impala table, or exists as raw data files outside Impala.

If you already have data in an Impala or Hive table, perhaps in a different file format or partitioning scheme, you can transfer the data to a Parquet table using the Impala INSERT...SELECT syntax. You can convert, filter, repartition, and do other things to the data as part of this same INSERT statement. See Compressions for Parquet Data Files on page 647 for some examples showing how to insert data into Parquet tables.

When inserting into partitioned tables, especially using the Parquet file format, you can include a hint in the INSERT statement to fine-tune the overall performance of the operation and its resource usage. See Optimizer Hints in Impala for using hints in the INSERT statements.

Any INSERT statement for a Parquet table requires enough free space in the HDFS filesystem to write one block. Because Parquet data files use a block size of 1 GB by default, an INSERT might fail (even for a very small amount of data) if your HDFS is running low on space.

Avoid the INSERT...VALUES syntax for Parquet tables, because INSERT...VALUES produces a separate tiny data file for each INSERT...VALUES statement, and the strength of Parquet is in its handling of data (compressing, parallelizing, and so on) in large chunks.

If you have one or more Parquet data files produced outside of Impala, you can quickly make the data queryable through Impala by one of the following methods:

- The LOAD DATA statement moves a single data file or a directory full of data files into the data directory for an Impala table. It does no validation or conversion of the data. The original data files must be somewhere in HDFS, not the local filesystem.
- The CREATE TABLE statement with the LOCATION clause creates a table where the data continues to reside outside the Impala data directory. The original data files must be somewhere in HDFS, not the local filesystem. For extra safety, if the data is intended to be long-lived and reused by other applications, you can use the CREATE EXTERNAL TABLE syntax so that the data files are not deleted by an Impala DROP TABLE statement.
- If the Parquet table already exists, you can copy Parquet data files directly into it, then use the REFRESH statement to make Impala recognize the newly added data. Remember to preserve the block size of the Parquet data files

by using the hadoop distcp -pb command rather than a -put or -cp operation on the Parquet files. See Example of Copying Parquet Data Files on page 648 for an example of this kind of operation.

Note:

Currently, Impala always decodes the column data in Parquet files based on the ordinal position of the columns, not by looking up the position of each column based on its name. Parquet files produced outside of Impala must write column data in the same order as the columns are declared in the Impala table. Any optional columns that are omitted from the data files must be the rightmost columns in the Impala table definition.

If you created compressed Parquet files through some tool other than Impala, make sure that any compression codecs are supported in Parquet by Impala. For example, Impala does not currently support LZO compression in Parquet files. Also doublecheck that you used any recommended compatibility settings in the other tool, such as spark.sql.parquet.binaryAsString when writing Parquet files through Spark.

Recent versions of Sqoop can produce Parquet output files using the --as-parquetfile option.

If the data exists outside Impala and is in some other format, combine both of the preceding techniques. First, use a LOAD DATA or CREATE EXTERNAL TABLE ... LOCATION statement to bring the data into an Impala table that uses the appropriate file format. Then, use an INSERT...SELECT statement to copy the data to the Parquet table, converting to Parquet format as part of the process.

Loading data into Parquet tables is a memory-intensive operation, because the incoming data is buffered until it reaches one data block in size, then that chunk of data is organized and compressed in memory before being written out. The memory consumption can be larger when inserting data into partitioned Parquet tables, because a separate data file is written for each combination of partition key column values, potentially requiring several large chunks to be manipulated in memory at once.

When inserting into a partitioned Parquet table, Impala redistributes the data among the nodes to reduce memory consumption. You might still need to temporarily increase the memory dedicated to Impala during the insert operation, or break up the load operation into several INSERT statements, or both.

Note: All the preceding techniques assume that the data you are loading matches the structure of the destination table, including column order, column names, and partition layout. To transform or reorganize the data, start by loading the data into a Parquet table that matches the underlying structure of the data, then use one of the table-copying techniques such as CREATE TABLE AS SELECT or INSERT ... SELECT to reorder or rename columns, divide the data among multiple partitions, and so on. For example to take a single comprehensive Parquet data file and load it into a partitioned table, you would use an INSERT ... SELECT statement with dynamic partitioning to let Impala create separate data files with the appropriate partition values; for an example, see INSERT Statement on page 277.

Query Performance for Impala Parquet Tables

Query performance for Parquet tables depends on the number of columns needed to process the SELECT list and WHERE clauses of the query, the way data is divided into large data files with block size equal to file size, the reduction in I/O by reading the data for each column in compressed format, which data files can be skipped (for partitioned tables), and the CPU overhead of decompressing the data for each column.

For example, the following is an efficient query for a Parquet table:

```
select avg(income) from census_data where state = 'CA';
```

The query processes only 2 columns out of a large number of total columns. If the table is partitioned by the STATE column, it is even more efficient because the query only has to read and decode 1 column from each data file, and it

can read only the data files in the partition directory for the state `'CA'`, skipping the data files for all the other states, which will be physically located in other directories.

The following is a relatively inefficient query for a Parquet table:

```
select * from census_data;
```

Impala would have to read the entire contents of each large data file, and decompress the contents of each column for each row group, negating the I/O optimizations of the column-oriented format. This query might still be faster for a Parquet table than a table with some other file format, but it does not take advantage of the unique strengths of Parquet data files.

Impala can optimize queries on Parquet tables, especially join queries, better when statistics are available for all the tables. Issue the `COMPUTE STATS` statement for each table after substantial amounts of data are loaded into or appended to it. See COMPUTE STATS Statement on page 219 for details.

The runtime filtering feature, available in CDH 5.7 / Impala 2.5 and higher, works best with Parquet tables. The per-row filtering aspect only applies to Parquet tables. See Runtime Filtering for Impala Queries (CDH 5.7 or higher only) on page 588 for details.

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

In CDH 5.12 and higher, Parquet files written by Impala include embedded metadata specifying the minimum and maximum values for each column, within each row group and each data page within the row group. Impala-written Parquet files typically contain a single row group; a row group can contain many data pages. Impala uses this information (currently, only the metadata for each row group) when reading each Parquet data file during a query, to quickly determine whether each row group within the file potentially includes any rows that match the conditions in the `WHERE` clause. For example, if the column `X` within a particular Parquet file has a minimum value of 1 and a maximum value of 100, then a query including the clause `WHERE x > 200` can quickly determine that it is safe to skip that particular file, instead of scanning all the associated column values. This optimization technique is especially effective for tables that use the `SORT BY` clause for the columns most frequently checked in `WHERE` clauses, because any `INSERT` operation on such tables produces Parquet data files with relatively narrow ranges of column values within each file.

Partitioning for Parquet Tables

As explained in Partitioning for Impala Tables on page 625, partitioning is an important performance technique for Impala generally. This section explains some of the performance considerations for partitioned Parquet tables.

The Parquet file format is ideal for tables containing many columns, where most queries only refer to a small subset of the columns. As explained in How Parquet Data Files Are Organized on page 652, the physical layout of Parquet data files lets Impala read only a small fraction of the data for many queries. The performance benefits of this approach are amplified when you use Parquet tables in combination with partitioning. Impala can skip the data files for certain partitions entirely, based on the comparisons in the `WHERE` clause that refer to the partition key columns. For example, queries on partitioned tables often analyze data for time intervals based on columns such as `YEAR`, `MONTH`, and/or `DAY`, or for geographic regions. Remember that Parquet data files use a large block size, so when deciding how finely to partition the data, try to find a granularity where each partition contains 256 MB or more of data, rather than creating a large number of smaller files split among many partitions.

Inserting into a partitioned Parquet table can be a resource-intensive operation, because each Impala node could potentially be writing a separate data file to HDFS for each combination of different values for the partition key columns. The large number of simultaneous open files could exceed the HDFS “transceivers” limit. To avoid exceeding this limit, consider the following techniques:

- Load different subsets of data using separate INSERT statements with specific values for the PARTITION clause, such as PARTITION (year=2010).
- Increase the "transceivers" value for HDFS, sometimes spelled "xcievers" (sic). The property value in the hdfs-site.xml configuration file is dfs.datanode.max.transfer.threads. For example, if you were loading 12 years of data partitioned by year, month, and day, even a value of 4096 might not be high enough. This blog post explores the considerations for setting this value higher or lower, using HBase examples for illustration.
- Use the COMPUTE STATS statement to collect column statistics on the source table from which data is being copied, so that the Impala query can estimate the number of different values in the partition key columns and distribute the work accordingly.

Compressions for Parquet Data Files

When Impala writes Parquet data files using the INSERT statement, the underlying compression is controlled by the COMPRESSION_CODEC query option. (Prior to Impala 2.0, the query option name was PARQUET_COMPRESSION_CODEC.) The allowed values for this query option are snappy (the default), gzip, zstd, and none. The option value is not case-sensitive. If the option is set to an unrecognized value, all kinds of queries will fail due to the invalid option setting, not just queries involving Parquet tables.

Example of Parquet Table with Snappy Compression

By default, the underlying data files for a Parquet table are compressed with Snappy. The combination of fast compression and decompression makes it a good choice for many data sets. To ensure Snappy compression is used, for example after experimenting with other compression codecs, set the COMPRESSION_CODEC query option to snappy before inserting the data:

```
[localhost:21000] > create database parquet_compression;
[localhost:21000] > use parquet_compression;
[localhost:21000] > create table parquet_snappy like raw_text_data;
[localhost:21000] > set COMPRESSION_CODEC=snappy;
[localhost:21000] > insert into parquet_snappy select * from raw_text_data;
Inserted 1000000000 rows in 181.98s
```

Example of Parquet Table with GZip Compression

If you need more intensive compression (at the expense of more CPU cycles for uncompressing during queries), set the COMPRESSION_CODEC query option to gzip before inserting the data:

```
[localhost:21000] > create table parquet_gzip like raw_text_data;
[localhost:21000] > set COMPRESSION_CODEC=gzip;
[localhost:21000] > insert into parquet_gzip select * from raw_text_data;
Inserted 1000000000 rows in 1418.24s
```

Example of Uncompressed Parquet Table

If your data compresses very poorly, or you want to avoid the CPU overhead of compression and decompression entirely, set the COMPRESSION_CODEC query option to none before inserting the data:

```
[localhost:21000] > create table parquet_none like raw_text_data;
[localhost:21000] > set COMPRESSION_CODEC=none;
[localhost:21000] > insert into parquet_none select * from raw_text_data;
Inserted 1000000000 rows in 146.90s
```

Examples of Sizes and Speeds for Compressed Parquet Tables

Here are some examples showing differences in data sizes and query speeds for 1 billion rows of synthetic data, compressed with each kind of codec. As always, run similar tests with realistic data sets of your own. The actual compression ratios, and relative insert and query speeds, will vary depending on the characteristics of the actual data.

In this case, switching from Snappy to GZip compression shrinks the data by an additional 40% or so, while switching from Snappy compression to no compression expands the data also by about 40%:

```
$ hdfs dfs -du -h /user/hive/warehouse/parquet_compression.db
23.1 G  /user/hive/warehouse/parquet_compression.db/parquet_snappy
13.5 G  /user/hive/warehouse/parquet_compression.db/parquet_gzip
32.8 G  /user/hive/warehouse/parquet_compression.db/parquet_none
```

Because Parquet data files are typically large, each directory will have a different number of data files and the row groups will be arranged differently.

At the same time, the less aggressive the compression, the faster the data can be decompressed. In this case using a table with a billion rows, a query that evaluates all the values for a particular column runs faster with no compression than with Snappy compression, and faster with Snappy compression than with Gzip compression. Query performance depends on several other factors, so as always, run your own benchmarks with your own data to determine the ideal tradeoff between data size, CPU efficiency, and speed of insert and query operations.

```
[localhost:21000] > desc parquet_snappy;
Query finished, fetching results ...
+-----------+---------+---------+
| name      | type    | comment |
+-----------+---------+---------+
| id        | int     |         |
| val       | int     |         |
| zfill     | string  |         |
| name      | string  |         |
| assertion | boolean |         |
+-----------+---------+---------+
Returned 5 row(s) in 0.14s
[localhost:21000] > select avg(val) from parquet_snappy;
Query finished, fetching results ...
+-----------------+
| _c0             |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 4.29s
[localhost:21000] > select avg(val) from parquet_gzip;
Query finished, fetching results ...
+-----------------+
| _c0             |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 6.97s
[localhost:21000] > select avg(val) from parquet_none;
Query finished, fetching results ...
+-----------------+
| _c0             |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 3.67s
```

Example of Copying Parquet Data Files

Here is a final example, to illustrate how the data files using the various compression codecs are all compatible with each other for read operations. The metadata about the compression format is written into each data file, and can be decoded during queries regardless of the `COMPRESSION_CODEC` setting in effect at the time. In this example, we copy data files from the `PARQUET_SNAPPY`, `PARQUET_GZIP`, and `PARQUET_NONE` tables used in the previous examples, each containing 1 billion rows, all to the data directory of a new table `PARQUET_EVERYTHING`. A couple of sample queries demonstrate that the new table now contains 3 billion rows featuring a variety of compression codecs for the data files.

First, we create the table in Impala so that there is a destination directory in HDFS to put the data files:

```
[localhost:21000] > create table parquet_everything like parquet_snappy;
Query: create table parquet_everything like parquet_snappy
```

Then in the shell, we copy the relevant data files into the data directory for this new table. Rather than using `hdfs dfs -cp` as with typical files, we use `hadoop distcp -pb` to ensure that the special block size of the Parquet data files is preserved.

```
$ hadoop distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_snappy \
  /user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
$ hadoop distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_gzip  \
  /user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
$ hadoop distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_none  \
  /user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
```

Back in the `impala-shell` interpreter, we use the REFRESH statement to alert the Impala server to the new data files for this table, then we can run queries demonstrating that the data files represent 3 billion rows, and the values for one of the numeric columns match what was in the original smaller tables:

```
[localhost:21000] > refresh parquet_everything;
Query finished, fetching results ...

Returned 0 row(s) in 0.32s
[localhost:21000] > select count(*) from parquet_everything;
Query finished, fetching results ...
+------------+
| _c0        |
+------------+
| 3000000000 |
+------------+
Returned 1 row(s) in 8.18s
[localhost:21000] > select avg(val) from parquet_everything;
Query finished, fetching results ...
+-----------------+
| _c0             |
+-----------------+
| 250000.93577915 |
+-----------------+
Returned 1 row(s) in 13.35s
```

Parquet Tables for Impala Complex Types

In CDH 5.5 / Impala 2.3 and higher, Impala supports the complex types ARRAY, STRUCT, and MAP See Complex Types (CDH 5.5 or higher only) on page 139 for details. Because these data types are currently supported only for the Parquet file format, if you plan to use them, become familiar with the performance and storage aspects of Parquet first.

Exchanging Parquet Data Files with Other Hadoop Components

You can read and write Parquet data files from other CDH components.

Originally, it was not possible to create Parquet data through Impala and reuse that table within Hive. Now that Parquet support is available for Hive, reusing existing Impala Parquet data files in Hive requires updating the table metadata. Use the following command if you are already running Impala 1.1.1 or higher:

```
ALTER TABLE table_name SET FILEFORMAT PARQUET;
```

If you are running a level of Impala that is older than 1.1.1, do the metadata update through Hive:

```
ALTER TABLE table_name SET SERDE 'parquet.hive.serde.ParquetHiveSerDe';
ALTER TABLE table_name SET FILEFORMAT
```

```
  INPUTFORMAT "parquet.hive.DeprecatedParquetInputFormat"
  OUTPUTFORMAT "parquet.hive.DeprecatedParquetOutputFormat";
```

Impala 1.1.1 and higher can reuse Parquet data files created by Hive, without any action required.

Impala supports the scalar data types that you can encode in a Parquet data file, but not composite or nested types such as maps or arrays. In CDH 5.4 / Impala 2.2 and higher, Impala can query Parquet data files that include composite or nested types, as long as the query only refers to columns with scalar types.

If you copy Parquet data files between nodes, or even between different directories on the same node, make sure to preserve the block size by using the command `hadoop distcp -pb`. To verify that the block size was preserved, issue the command `hdfs fsck -blocks` *`HDFS_path_of_impala_table_dir`* and check that the average block size is at or near 256 MB (or whatever other size is defined by the `PARQUET_FILE_SIZE` query option).. (The `hadoop distcp` operation typically leaves some directories behind, with names matching `_distcp_logs_*`, that you can delete from the destination directory afterward.) Issue the command `hadoop distcp` for details about `distcp` command syntax.

Impala can query Parquet files that use the `PLAIN`, `PLAIN_DICTIONARY`, `BIT_PACKED`, and `RLE` encodings. Currently, Impala does not support `RLE_DICTIONARY` encoding. When creating files outside of Impala for use by Impala, make sure to use one of the supported encodings. In particular, for MapReduce jobs, `parquet.writer.version` must not be defined (especially as `PARQUET_2_0`) for writing the configurations of Parquet MR jobs. Data using the version 2.0 of Parquet writer might not be consumable by Impala, due to use of the `RLE_DICTIONARY` encoding. Use the default version of the Parquet writer and refrain from overriding the default writer version by setting the `parquet.writer.version` property or via `WriterVersion.PARQUET_2_0` in the Parquet API.

To examine the internal structure and data of Parquet files, you can use the `parquet-tools` command that comes with CDH. Make sure this command is in your `$PATH`. (Typically, it is symlinked from `/usr/bin`; sometimes, depending on your installation setup, you might need to locate it under a CDH-specific `bin` directory.) The arguments to this command let you perform operations such as:

- `cat`: Print a file's contents to standard out. In CDH 5.5 and higher, you can use the `-j` option to output JSON.
- `head`: Print the first few records of a file to standard output.
- `schema`: Print the Parquet schema for the file.
- `meta`: Print the file footer metadata, including key-value properties (like Avro schema), compression ratios, encodings, compression used, and row group information.
- `dump`: Print all data and metadata.

Use `parquet-tools -h` to see usage information for all the arguments. Here are some examples showing `parquet-tools` usage:

```
$ # Be careful doing this for a big file! Use parquet-tools head to be safe.
$ parquet-tools cat sample.parq
year = 1992
month = 1
day = 2
dayofweek = 4
dep_time = 748
crs_dep_time = 750
arr_time = 851
crs_arr_time = 846
carrier = US
flight_num = 53
actual_elapsed_time = 63
crs_elapsed_time = 56
arrdelay = 5
depdelay = -2
origin = CMH
dest = IND
distance = 182
cancelled = 0
diverted = 0

year = 1992
month = 1
```

```
day = 3
...
```

```
$ parquet-tools head -n 2 sample.parq
year = 1992
month = 1
day = 2
dayofweek = 4
dep_time = 748
crs_dep_time = 750
arr_time = 851
crs_arr_time = 846
carrier = US
flight_num = 53
actual_elapsed_time = 63
crs_elapsed_time = 56
arrdelay = 5
depdelay = -2
origin = CMH
dest = IND
distance = 182
cancelled = 0
diverted = 0

year = 1992
month = 1
day = 3
...
```

```
$ parquet-tools schema sample.parq
message schema {
  optional int32 year;
  optional int32 month;
  optional int32 day;
  optional int32 dayofweek;
  optional int32 dep_time;
  optional int32 crs_dep_time;
  optional int32 arr_time;
  optional int32 crs_arr_time;
  optional binary carrier;
  optional int32 flight_num;
...
```

```
$ parquet-tools meta sample.parq
creator:             impala version 2.2.0-cdh5.4.3 (build
517bb0f71cd604a00369254ac6d88394df83e0f6)

file schema:         schema
-------------------------------------------------------------------
year:                OPTIONAL INT32 R:0 D:1
month:               OPTIONAL INT32 R:0 D:1
day:                 OPTIONAL INT32 R:0 D:1
dayofweek:           OPTIONAL INT32 R:0 D:1
dep_time:            OPTIONAL INT32 R:0 D:1
crs_dep_time:        OPTIONAL INT32 R:0 D:1
arr_time:            OPTIONAL INT32 R:0 D:1
crs_arr_time:        OPTIONAL INT32 R:0 D:1
carrier:             OPTIONAL BINARY R:0 D:1
flight_num:          OPTIONAL INT32 R:0 D:1
...

row group 1:         RC:20636601 TS:265103674
-------------------------------------------------------------------
year:                 INT32 SNAPPY DO:4 FPO:35 SZ:10103/49723/4.92 VC:20636601
```

```
ENC:PLAIN_DICTIONARY,RLE,PLAIN
month:                INT32 SNAPPY DO:10147 FPO:10210 SZ:11380/35732/3.14 VC:20636601
ENC:PLAIN_DICTIONARY,RLE,PLAIN
day:                INT32 SNAPPY DO:21572 FPO:21714 SZ:3071658/9868452/3.21 VC:20636601
 ENC:PLAIN_DICTIONARY,RLE,PLAIN
dayofweek:            INT32 SNAPPY DO:3093276 FPO:3093319 SZ:2274375/5941876/2.61
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
dep_time:             INT32 SNAPPY DO:5367705 FPO:5373967 SZ:28281281/28573175/1.01
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
crs_dep_time:         INT32 SNAPPY DO:33649039 FPO:33654262 SZ:10220839/11574964/1.13
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
arr_time:             INT32 SNAPPY DO:43869935 FPO:43876489 SZ:28562410/28797767/1.01
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
crs_arr_time:         INT32 SNAPPY DO:72432398 FPO:72438151 SZ:10908972/12164626/1.12
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
carrier:              BINARY SNAPPY DO:83341427 FPO:83341558 SZ:114916/128611/1.12
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
flight_num:           INT32 SNAPPY DO:83456393 FPO:83488603 SZ:10216514/11474301/1.12
VC:20636601 ENC:PLAIN_DICTIONARY,RLE,PLAIN
...
```

How Parquet Data Files Are Organized

Although Parquet is a column-oriented file format, do not expect to find one data file for each column. Parquet keeps all the data for a row within the same data file, to ensure that the columns for a row are always available on the same node for processing. What Parquet does is to set a large HDFS block size and a matching maximum data file size, to ensure that I/O and network transfer requests apply to large batches of data.

Within that data file, the data for a set of rows is rearranged so that all the values from the first column are organized in one contiguous block, then all the values from the second column, and so on. Putting the values from the same column next to each other lets Impala use effective compression techniques on the values in that column.



Note:

Impala `INSERT` statements write Parquet data files using an HDFS block size that matches the data file size, to ensure that each data file is represented by a single HDFS block, and the entire file can be processed on a single node without requiring any remote reads.

If you create Parquet data files outside of Impala, such as through a MapReduce or Pig job, ensure that the HDFS block size is greater than or equal to the file size, so that the "one file per block" relationship is maintained. Set the `dfs.block.size` or the `dfs.blocksize` property large enough that each file fits within a single HDFS block, even if that size is larger than the normal HDFS block size.

If the block size is reset to a lower value during a file copy, you will see lower performance for queries involving those files, and the `PROFILE` statement will reveal that some I/O is being done suboptimally, through remote reads. See Example of Copying Parquet Data Files on page 648 for an example showing how to preserve the block size when copying Parquet data files.

When Impala retrieves or tests the data for a particular column, it opens all the data files, but only reads the portion of each file containing the values for that column. The column values are stored consecutively, minimizing the I/O required to process the values within a single column. If other columns are named in the `SELECT` list or `WHERE` clauses, the data for all columns in the same row is available within that same data file.

If an `INSERT` statement brings in less than one Parquet block's worth of data, the resulting data file is smaller than ideal. Thus, if you do split up an ETL job to use multiple `INSERT` statements, try to keep the volume of data for each `INSERT` statement to approximately 256 MB, or a multiple of 256 MB.

RLE and Dictionary Encoding for Parquet Data Files

Parquet uses some automatic compression techniques, such as run-length encoding (RLE) and dictionary encoding, based on analysis of the actual data values. Once the data values are encoded in a compact form, the encoded data can optionally be further compressed using a compression algorithm. Parquet data files created by Impala can use

Snappy, GZip, or no compression; the Parquet spec also allows LZO compression, but currently Impala does not support LZO-compressed Parquet files.

RLE and dictionary encoding are compression techniques that Impala applies automatically to groups of Parquet data values, in addition to any Snappy or GZip compression applied to the entire data files. These automatic optimizations can save you time and planning that are normally needed for a traditional data warehouse. For example, dictionary encoding reduces the need to create numeric IDs as abbreviations for longer string values.

Run-length encoding condenses sequences of repeated data values. For example, if many consecutive rows all contain the same value for a country code, those repeating values can be represented by the value followed by a count of how many times it appears consecutively.

Dictionary encoding takes the different values present in a column, and represents each one in compact 2-byte form rather than the original value, which could be several bytes. (Additional compression is applied to the compacted values, for extra space savings.) This type of encoding applies when the number of different values for a column is less than 2**16 (16,384). It does not apply to columns of data type `BOOLEAN`, which are already very short. `TIMESTAMP` columns sometimes have a unique value for each row, in which case they can quickly exceed the 2**16 limit on distinct values. The 2**16 limit on different values within a column is reset for each data file, so if several different data files each contained 10,000 different city names, the city name column in each data file could still be condensed using dictionary encoding.

Compacting Data Files for Parquet Tables

If you reuse existing table structures or ETL processes for Parquet tables, you might encounter a “many small files” situation, which is suboptimal for query efficiency. For example, statements like these might produce inefficiently organized data files:

```
-- In an N-node cluster, each node produces a data file
-- for the INSERT operation. If you have less than
-- N GB of data to copy, some files are likely to be
-- much smaller than the default Parquet block size.
insert into parquet_table select * from text_table;

-- Even if this operation involves an overall large amount of data,
-- when split up by year/month/day, each partition might only
-- receive a small amount of data. Then the data files for
-- the partition might be divided between the N nodes in the cluster.
-- A multi-gigabyte copy operation might produce files of only
-- a few MB each.
insert into partitioned_parquet_table partition (year, month, day)
  select year, month, day, url, referer, user_agent, http_code, response_time
  from web_stats;
```

Here are techniques to help you produce large data files in Parquet `INSERT` operations, and to compact existing too-small data files:

- When inserting into a partitioned Parquet table, use statically partitioned `INSERT` statements where the partition key values are specified as constant values. Ideally, use a separate `INSERT` statement for each partition.
- You might set the `NUM_NODES` option to 1 briefly, during `INSERT` or `CREATE TABLE AS SELECT` statements. Normally, those statements produce one or more data files per data node. If the write operation involves small amounts of data, a Parquet table, and/or a partitioned table, the default behavior could produce many small files when intuitively you might expect only a single output file. `SET NUM_NODES=1` turns off the “distributed” aspect of the write operation, making it more likely to produce only one or a few data files.
- Be prepared to reduce the number of partition key columns from what you are used to with traditional analytic database systems.
- Do not expect Impala-written Parquet files to fill up the entire Parquet block size. Impala estimates on the conservative side when figuring out how much data to write to each Parquet file. Typically, the of uncompressed data in memory is substantially reduced on disk by the compression and encoding techniques in the Parquet file format. The final data file size varies depending on the compressibility of the data. Therefore, it is not an indication of a problem if 256 MB of text data is turned into 2 Parquet data files, each less than 256 MB.

- If you accidentally end up with a table with many small data files, consider using one or more of the preceding techniques and copying all the data into a new Parquet table, either through `CREATE TABLE AS SELECT` or `INSERT ... SELECT` statements.

  To avoid rewriting queries to change table names, you can adopt a convention of always running important queries against a view. Changing the view definition immediately switches any subsequent queries to use the new underlying tables:

```
create view production_table as select * from table_with_many_small_files;
-- CTAS or INSERT...SELECT all the data into a more efficient layout...
alter view production_table as select * from table_with_few_big_files;
select * from production_table where c1 = 100 and c2 < 50 and ...;
```

Schema Evolution for Parquet Tables

Schema evolution refers to using the statement `ALTER TABLE ... REPLACE COLUMNS` to change the names, data type, or number of columns in a table. You can perform schema evolution for Parquet tables as follows:

- The Impala `ALTER TABLE` statement never changes any data files in the tables. From the Impala side, schema evolution involves interpreting the same data files in terms of a new table definition. Some types of schema changes make sense and are represented correctly. Other types of changes cannot be represented in a sensible way, and produce special result values or conversion errors during queries.
- The `INSERT` statement always creates data using the latest table definition. You might end up with data files with different numbers of columns or internal data representations if you do a sequence of `INSERT` and `ALTER TABLE ... REPLACE COLUMNS` statements.
- If you use `ALTER TABLE ... REPLACE COLUMNS` to define additional columns at the end, when the original data files are used in a query, these final columns are considered to be all `NULL` values.
- If you use `ALTER TABLE ... REPLACE COLUMNS` to define fewer columns than before, when the original data files are used in a query, the unused columns still present in the data file are ignored.
- Parquet represents the `TINYINT`, `SMALLINT`, and `INT` types the same internally, all stored in 32-bit integers.
  - That means it is easy to promote a `TINYINT` column to `SMALLINT` or `INT`, or a `SMALLINT` column to `INT`. The numbers are represented exactly the same in the data file, and the columns being promoted would not contain any out-of-range values.
  - If you change any of these column types to a smaller type, any values that are out-of-range for the new type are returned incorrectly, typically as negative numbers.
  - You cannot change a `TINYINT`, `SMALLINT`, or `INT` column to `BIGINT`, or the other way around. Although the `ALTER TABLE` succeeds, any attempt to query those columns results in conversion errors.
  - Any other type conversion for columns produces a conversion error during queries. For example, `INT` to `STRING`, `FLOAT` to `DOUBLE`, `TIMESTAMP` to `STRING`, `DECIMAL(9,0)` to `DECIMAL(5,2)`, and so on.

You might find that you have Parquet files where the columns do not line up in the same order as in your Impala table. For example, you might have a Parquet file that was part of a table with columns `C1,C2,C3,C4`, and now you want to reuse the same Parquet file in a table with columns `C4,C2`. By default, Impala expects the columns in the data file to appear in the same order as the columns defined for the table, making it impractical to do some kinds of file reuse or schema evolution. In CDH 5.8 / Impala 2.6 and higher, the query option `PARQUET_FALLBACK_SCHEMA_RESOLUTION=name` lets Impala resolve columns by name, and therefore handle out-of-order or extra columns in the data file. For example:

```
create database schema_evolution;
use schema_evolution;
create table t1 (c1 int, c2 boolean, c3 string, c4 timestamp)
  stored as parquet;
```

```
insert into t1 values
  (1, true, 'yes', now()),
  (2, false, 'no', now() + interval 1 day);

select * from t1;
+----+-------+-----+-------------------------------+
| c1 | c2    | c3  | c4                            |
+----+-------+-----+-------------------------------+
| 1  | true  | yes | 2016-06-28 14:53:26.554369000 |
| 2  | false | no  | 2016-06-29 14:53:26.554369000 |
+----+-------+-----+-------------------------------+

desc formatted t1;
...
| Location:   | /user/hive/warehouse/schema_evolution.db/t1 |
...

-- Make T2 have the same data file as in T1, including 2
-- unused columns and column order different than T2 expects.
load data inpath '/user/hive/warehouse/schema_evolution.db/t1'
  into table t2;
+-----------------------------------------------------------+
| summary                                                   |
+-----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 1 |
+-----------------------------------------------------------+

-- 'position' is the default setting.
-- Impala cannot read the Parquet file if the column order does not match.
set PARQUET_FALLBACK_SCHEMA_RESOLUTION=position;
PARQUET_FALLBACK_SCHEMA_RESOLUTION set to position

select * from t2;
WARNINGS:
File 'schema_evolution.db/t2/45331705_data.0.parq'
has an incompatible Parquet schema for column 'schema_evolution.t2.c4'.
Column type: TIMESTAMP, Parquet schema: optional int32 c1 [i:0 d:1 r:0]

File 'schema_evolution.db/t2/45331705_data.0.parq'
has an incompatible Parquet schema for column 'schema_evolution.t2.c4'.
Column type: TIMESTAMP, Parquet schema: optional int32 c1 [i:0 d:1 r:0]

-- With the 'name' setting, Impala can read the Parquet data files
-- despite mismatching column order.
set PARQUET_FALLBACK_SCHEMA_RESOLUTION=name;
PARQUET_FALLBACK_SCHEMA_RESOLUTION set to name

select * from t2;
+-------------------------------+-------+
| c4                            | c2    |
+-------------------------------+-------+
| 2016-06-28 14:53:26.554369000 | true  |
| 2016-06-29 14:53:26.554369000 | false |
+-------------------------------+-------+
```

See PARQUET_FALLBACK_SCHEMA_RESOLUTION Query Option (CDH 5.8 or higher only) on page 353 for more details.

Data Type Considerations for Parquet Tables

The Parquet format defines a set of data types whose names differ from the names of the corresponding Impala data types. If you are preparing Parquet files using other Hadoop components such as Pig or MapReduce, you might need to work with the type names defined by Parquet. The following tables list the Parquet-defined types and the equivalent types in Impala.

Primitive types

Parquet type	Impala type
BINARY	STRING

Parquet type	Impala type
BOOLEAN	BOOLEAN
DOUBLE	DOUBLE
FLOAT	FLOAT
INT32	INT
INT64	BIGINT
INT96	TIMESTAMP

Logical types

Parquet uses type annotations to extend the types that it can store, by specifying how the primitive types should be interpreted.

Parquet primitive type and annotation	Impala type
BINARY annotated with the UTF8 OriginalType	STRING
BINARY annotated with the STRING LogicalType	STRING
BINARY annotated with the ENUM OriginalType	STRING
BINARY annotated with the DECIMAL OriginalType	DECIMAL
INT64 annotated with the TIMESTAMP_MILLIS OriginalType	TIMESTAMP (in CDH 6.2 or higher) or BIGINT (for backward compatibility)
INT64 annotated with the TIMESTAMP_MICROS OriginalType	TIMESTAMP (in CDH 6.2 or higher) or BIGINT (for backward compatibility)
INT64 annotated with the TIMESTAMP LogicalType	TIMESTAMP (in CDH 6.2 or higher) or BIGINT (for backward compatibility)

Complex types:

For the complex types (ARRAY, MAP, and STRUCT) available in CDH 5.5 / Impala 2.3 and higher, Impala only supports queries against those types in Parquet tables.

Using the ORC File Format with Impala Tables

Impala can read ORC data files as an experimental feature since Impala 3.1.

To enable the feature, set `--enable_orc_scanner` to `true` when starting the cluster.



Note: Impala's ORC support is not yet production quality. Parquet remains the preferred file format for use with Impala and offers significantly better performance and a more complete set of features.

Table 3: ORC Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
ORC	Structured	gzip, Snappy, LZO, LZ4; currently gzip by default	The ORC support is an experimental feature since CDH 6.1 / Impala 3.1 & Impala 2.12. To disable it, set `--enable_orc_scanner` to `false` when starting the cluster.	No. Import data by using `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive followed by `REFRESH` *`table_name`* in Impala.

Creating ORC Tables and Loading Data

If you do not have an existing data file to use, begin by creating one in the appropriate format.

To create an ORC table:

In the `impala-shell` interpreter, issue a command similar to:

```
CREATE TABLE orc_table (column_specs) STORED AS ORC;
```

Because Impala can query some kinds of tables that it cannot currently write to, after creating tables of certain file formats, you might use the Hive shell to load the data. See How Impala Works with Hadoop File Formats on page 634 for details. After loading data into a table through Hive or other mechanism outside of Impala, issue a `REFRESH` *`table_name`* statement the next time you connect to the Impala node, before querying the table, to make Impala recognize the new data.

For example, here is how you might create some ORC tables in Impala (by specifying the columns explicitly, or cloning the structure of another table), load data through Hive, and query them through Impala:

```
$ impala-shell -i localhost
[localhost:21000] default> CREATE TABLE orc_table (x INT) STORED AS ORC;
[localhost:21000] default> CREATE TABLE orc_clone LIKE some_other_table STORED AS ORC;
[localhost:21000] default> quit;

$ hive
hive> INSERT INTO TABLE orc_table SELECT x FROM some_other_table;
3 Rows loaded to orc_table
Time taken: 4.169 seconds
hive> quit;

$ impala-shell -i localhost
[localhost:21000] default> SELECT * FROM orc_table;
Fetched 0 row(s) in 0.11s
[localhost:21000] default> -- Make Impala recognize the data loaded through Hive;
[localhost:21000] default> REFRESH orc_table;
[localhost:21000] default> SELECT * FROM orc_table;
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Fetched 3 row(s) in 0.11s
```

Enabling Compression for ORC Tables

ORC tables are in zlib (Deflate in Impala) compression in default. You may want to use Snappy or LZO compression on existing tables for different balance between compression ratio and decompression speed. In Hive-1.1.0, the supported

compressions for ORC tables are NONE, ZLIB, SNAPPY and LZO. For example, to enable Snappy compression, you would specify the following additional settings when loading data through the Hive shell:

```
hive> SET hive.exec.compress.output=true;
hive> SET orc.compress=SNAPPY;
hive> INSERT OVERWRITE TABLE new_table SELECT * FROM old_table;
```

If you are converting partitioned tables, you must complete additional steps. In such a case, specify additional settings similar to the following:

```
hive> CREATE TABLE new_table (your_cols) PARTITIONED BY (partition_cols) STORED AS
new_format;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE new_table PARTITION(comma_separated_partition_cols) SELECT
 * FROM old_table;
```

Remember that Hive does not require that you specify a source format for it. Consider the case of converting a table with two partition columns called `year` and `month` to a Snappy compressed ORC table. Combining the components outlined previously to complete this table conversion, you would specify settings similar to the following:

```
hive> CREATE TABLE tbl_orc (int_col INT, string_col STRING) STORED AS ORC;
hive> SET hive.exec.compress.output=true;
hive> SET orc.compress=SNAPPY;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_orc SELECT * FROM tbl;
```

To complete a similar process for a table that includes partitions, you would specify settings similar to the following:

```
hive> CREATE TABLE tbl_orc (int_col INT, string_col STRING) PARTITIONED BY (year INT)
STORED AS ORC;
hive> SET hive.exec.compress.output=true;
hive> SET orc.compress=SNAPPY;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_orc PARTITION(year) SELECT * FROM tbl;
```



Note:

The compression type is specified in the following command:

```
SET orc.compress=SNAPPY;
```

You could elect to specify alternative codecs such as `NONE`, `GZIP`, `LZO` here.

Query Performance for Impala ORC Tables

In general, expect query performance with ORC tables to be faster than with tables using text data, but slower than with Parquet tables since there're bunch of optimizations for Parquet. See Using the Parquet File Format with Impala Tables on page 643 for information about using the Parquet file format for high-performance analytic queries.

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Data Type Considerations for ORC Tables

The ORC format defines a set of data types whose names differ from the names of the corresponding Impala data types. If you are preparing ORC files using other Hadoop components such as Pig or MapReduce, you might need to work with the type names defined by ORC. The following figure lists the ORC-defined types and the equivalent types in Impala.

Primitive types:

```
BINARY -> STRING
BOOLEAN -> BOOLEAN
DOUBLE -> DOUBLE
FLOAT -> FLOAT
TINYINT -> TINYINT
SMALLINT -> SMALLINT
INT -> INT
BIGINT -> BIGINT
TIMESTAMP -> TIMESTAMP
DATE (not supported)
```

Complex types:

Complex types are currently not supported on ORC. However, queries materializing only scalar type columns are allowed:

```
$ hive
hive> CREATE TABLE orc_nested_table (id INT, a ARRAY<INT>) STORED AS ORC;
hive> INSERT INTO TABLE orc_nested_table SELECT 1, ARRAY(1,2,3);
OK
Time taken: 2.629 seconds
hive> quit;

$ impala-shell -i localhost
[localhost:21000] default> INVALIDATE METADATA orc_nested_table;
[localhost:21000] default> SELECT 1 FROM orc_nested_table t, t.a;
ERROR: NotImplementedException: Scan of table 't' in format 'ORC' is not supported
because the table has a column 'a' with a complex type 'ARRAY<INT>'.
Complex types are supported for these file formats: PARQUET.

[localhost:21000] default> SELECT COUNT(*) FROM orc_nested_table;
+----------+
| count(*) |
+----------+
| 1        |
+----------+
Fetched 1 row(s) in 0.12s

[localhost:21000] default> SELECT id FROM orc_nested_table;
+----+
| id |
+----+
| 1  |
+----+
Fetched 1 row(s) in 0.12s
```

Using the Avro File Format with Impala Tables

Impala supports using tables whose data files use the Avro file format. Impala can query Avro tables. In Impala 1.4.0 and higher, Impala can create Avro tables, but cannot insert data into them. For insert operations, use Hive, then switch back to Impala to run queries.

Table 4: Avro Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Avro	Structured	Snappy, gzip, deflate	Yes, in Impala 1.4.0 and higher. In lower versions, create the table using Hive.	No. Import data by using LOAD DATA on data files already in the right format, or use INSERT in Hive followed by REFRESH *table_name* in Impala.

Creating Avro Tables

To create a new table using the Avro file format, issue the CREATE TABLE statement through Impala with the STORED AS AVRO clause, or through Hive. If you create the table through Impala, you must include column definitions that match the fields specified in the Avro schema. With Hive, you can omit the columns and just specify the Avro schema.

In CDH 5.5 / Impala 2.3 and higher, the CREATE TABLE for Avro tables can include SQL-style column definitions rather than specifying Avro notation through the TBLPROPERTIES clause. Impala issues warning messages if there are any mismatches between the types specified in the SQL column definitions and the underlying types; for example, any TINYINT or SMALLINT columns are treated as INT in the underlying Avro files, and therefore are displayed as INT in any DESCRIBE or SHOW CREATE TABLE output.



Note:

Currently, Avro tables cannot contain TIMESTAMP columns. If you need to store date and time values in Avro tables, as a workaround you can use a STRING representation of the values, convert the values to BIGINT with the UNIX_TIMESTAMP( ) function, or create separate numeric columns for individual date and time fields using the EXTRACT( ) function.

The following examples demonstrate creating an Avro table in Impala, using either an inline column specification or one taken from a JSON file stored in HDFS:

```
[localhost:21000] > CREATE TABLE avro_only_sql_columns
                  > (
                  >   id INT,
                  >   bool_col BOOLEAN,
                  >   tinyint_col TINYINT, /* Gets promoted to INT */
                  >   smallint_col SMALLINT, /* Gets promoted to INT */
                  >   int_col INT,
                  >   bigint_col BIGINT,
                  >   float_col FLOAT,
                  >   double_col DOUBLE,
                  >   date_string_col STRING,
                  >   string_col STRING
                  > )
                  > STORED AS AVRO;

[localhost:21000] > CREATE TABLE impala_avro_table
                  > (bool_col BOOLEAN, int_col INT, long_col BIGINT, float_col FLOAT,
double_col DOUBLE, string_col STRING, nullable_int INT)
                  > STORED AS AVRO
                  > TBLPROPERTIES ('avro.schema.literal'='{
                  >    "name": "my_record",
                  >    "type": "record",
                  >    "fields": [
                  >       {"name":"bool_col", "type":"boolean"},
                  >       {"name":"int_col", "type":"int"},
                  >       {"name":"long_col", "type":"long"},
                  >       {"name":"float_col", "type":"float"},
                  >       {"name":"double_col", "type":"double"},
                  >       {"name":"string_col", "type":"string"},
                  >       {"name": "nullable_int", "type": ["null", "int"]}]}');
```

```
[localhost:21000] > CREATE TABLE avro_examples_of_all_types (
                  >     id INT,
                  >     bool_col BOOLEAN,
                  >     tinyint_col TINYINT,
                  >     smallint_col SMALLINT,
                  >     int_col INT,
                  >     bigint_col BIGINT,
                  >     float_col FLOAT,
                  >     double_col DOUBLE,
                  >     date_string_col STRING,
                  >     string_col STRING
                  >   )
                  >   STORED AS AVRO
                  >   TBLPROPERTIES
('avro.schema.url'='hdfs://localhost:8020/avro_schemas/alltypes.json');
```

The following example demonstrates creating an Avro table in Hive:

```
hive> CREATE TABLE hive_avro_table
    > ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
   > STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

    > OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
    > TBLPROPERTIES ('avro.schema.literal'='{
    >    "name": "my_record",
    >    "type": "record",
    >    "fields": [
    >       {"name":"bool_col", "type":"boolean"},
    >       {"name":"int_col", "type":"int"},
    >       {"name":"long_col", "type":"long"},
    >       {"name":"float_col", "type":"float"},
    >       {"name":"double_col", "type":"double"},
    >       {"name":"string_col", "type":"string"},
    >       {"name": "nullable_int", "type": ["null", "int"]}]}');
```

Each field of the record becomes a column of the table. Note that any other information, such as the record name, is ignored.

Note: For nullable Avro columns, make sure to put the `"null"` entry before the actual type name. In Impala, all columns are nullable; Impala currently does not have a `NOT NULL` clause. Any non-nullable property is only enforced on the Avro side.

Most column types map directly from Avro to Impala under the same names. These are the exceptions and special cases to consider:

- The `DECIMAL` type is defined in Avro as a `BYTE` type with the `logicalType` property set to `"decimal"` and a specified precision and scale.
- The Avro `long` type maps to `BIGINT` in Impala.

If you create the table through Hive, switch back to `impala-shell` and issue an `INVALIDATE METADATA` *`table_name`* statement. Then you can run queries for that table through `impala-shell`.

In rare instances, a mismatch could occur between the Avro schema and the column definitions in the metastore database. In CDH 5.5 / Impala 2.3 and higher, Impala checks for such inconsistencies during a `CREATE TABLE` statement and each time it loads the metadata for a table (for example, after `INVALIDATE METADATA`). Impala uses the following rules to determine how to treat mismatching columns, a process known as ***schema reconciliation***:

- If there is a mismatch in the number of columns, Impala uses the column definitions from the Avro schema.
- If there is a mismatch in column name or type, Impala uses the column definition from the Avro schema. Because a `CHAR` or `VARCHAR` column in Impala maps to an Avro `STRING`, this case is not considered a mismatch and the column is preserved as `CHAR` or `VARCHAR` in the reconciled schema. Prior to CDH 5.9 / Impala 2.7 the column name and comment for such `CHAR` and `VARCHAR` columns was also taken from the SQL column definition. In CDH 5.9 /

Impala 2.7 and higher, the column name and comment from the Avro schema file take precedence for such columns, and only the CHAR or VARCHAR type is preserved from the SQL column definition.

- An Impala TIMESTAMP column definition maps to an Avro STRING and is presented as a STRING in the reconciled schema, because Avro has no binary TIMESTAMP representation. As a result, no Avro table can have a TIMESTAMP column; this restriction is the same as in earlier Impala releases.

Complex type considerations: Although you can create tables in this file format using the complex types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher, currently, Impala can query these types only in Parquet tables. The one exception to the preceding rule is COUNT(*) queries on RCFile tables that include complex types. Such queries are allowed in CDH 5.8 / Impala 2.6 and higher.

Using a Hive-Created Avro Table in Impala

If you have an Avro table created through Hive, you can use it in Impala as long as it contains only Impala-compatible data types. It cannot contain:

- Complex types: `array`, `map`, `record`, `struct`, `union` other than `[`*`supported_type`*`,null]` or `[null,`*`supported_type`*`]`
- The Avro-specific types `enum`, `bytes`, and `fixed`
- Any scalar type other than those listed in Data Types on page 101

Because Impala and Hive share the same metastore database, Impala can directly access the table definitions and data for tables that were created in Hive.

If you create an Avro table in Hive, issue an `INVALIDATE METADATA` the next time you connect to Impala through `impala-shell`. This is a one-time operation to make Impala aware of the new table. You can issue the statement while connected to any Impala node, and the catalog service broadcasts the change to all other Impala nodes.

If you load new data into an Avro table through Hive, either through a Hive `LOAD DATA` or `INSERT` statement, or by manually copying or moving files into the data directory for the table, issue a `REFRESH` *`table_name`* statement the next time you connect to Impala through `impala-shell`. You can issue the statement while connected to any Impala node, and the catalog service broadcasts the change to all other Impala nodes. If you issue the `LOAD DATA` statement through Impala, you do not need a `REFRESH` afterward.

Impala only supports fields of type `boolean`, `int`, `long`, `float`, `double`, and `string`, or unions of these types with null; for example, `["string", "null"]`. Unions with `null` essentially create a nullable type.

Specifying the Avro Schema through JSON

While you can embed a schema directly in your `CREATE TABLE` statement, as shown above, column width restrictions in the Hive metastore limit the length of schema you can specify. If you encounter problems with long schema literals, try storing your schema as a `JSON` file in HDFS instead. Specify your schema in HDFS using table properties similar to the following:

```
tblproperties ('avro.schema.url'='hdfs//your-name-node:port/path/to/schema.json');
```

Loading Data into an Avro Table

Currently, Impala cannot write Avro data files. Therefore, an Avro table cannot be used as the destination of an Impala `INSERT` statement or `CREATE TABLE AS SELECT`.

To copy data from another table, issue any `INSERT` statements through Hive. For information about loading data into Avro tables through Hive, see the Avro page on the Hive wiki.

If you already have data files in Avro format, you can also issue `LOAD DATA` in either Impala or Hive. Impala can move existing Avro data files into an Avro table, it just cannot create new Avro data files.

Enabling Compression for Avro Tables

To enable compression for Avro tables, specify settings in the Hive shell to enable compression and to specify a codec, then issue a `CREATE TABLE` statement as in the preceding examples. Impala supports the `snappy` and `deflate` codecs for Avro tables.

For example:

```
hive> set hive.exec.compress.output=true;
hive> set avro.output.codec=snappy;
```

How Impala Handles Avro Schema Evolution

Starting in Impala 1.1, Impala can deal with Avro data files that employ ***schema evolution***, where different data files within the same table use slightly different type definitions. (You would perform the schema evolution operation by issuing an `ALTER TABLE` statement in the Hive shell.) The old and new types for any changed columns must be compatible, for example a column might start as an `int` and later change to a `bigint` or `float`.

As with any other tables where the definitions are changed or data is added outside of the current `impalad` node, ensure that Impala loads the latest metadata for the table if the Avro schema is modified through Hive. Issue a `REFRESH` *`table_name`* or `INVALIDATE METADATA` *`table_name`* statement. `REFRESH` reloads the metadata immediately, `INVALIDATE METADATA` reloads the metadata the next time the table is accessed.

When Avro data files or columns are not consulted during a query, Impala does not check for consistency. Thus, if you issue `SELECT c1, c2 FROM t1`, Impala does not return any error if the column `c3` changed in an incompatible way. If a query retrieves data from some partitions but not others, Impala does not check the data files for the unused partitions.

In the Hive DDL statements, you can specify an `avro.schema.literal` table property (if the schema definition is short) or an `avro.schema.url` property (if the schema definition is long, or to allow convenient editing for the definition).

For example, running the following SQL code in the Hive shell creates a table using the Avro file format and puts some sample data into it:

```
CREATE TABLE avro_table (a string, b string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
TBLPROPERTIES (
  'avro.schema.literal'='{
    "type": "record",
    "name": "my_record",
    "fields": [
      {"name": "a", "type": "int"},
      {"name": "b", "type": "string"}
    ]}');

INSERT OVERWRITE TABLE avro_table SELECT 1, "avro" FROM functional.alltypes LIMIT 1;
```

Once the Avro table is created and contains data, you can query it through the `impala-shell` command:

```
[localhost:21000] > select * from avro_table;
+---+------+
| a | b    |
+---+------+
| 1 | avro |
+---+------+
```

Now in the Hive shell, you change the type of a column and add a new column with a default value:

```
-- Promote column "a" from INT to FLOAT (no need to update Avro schema)
ALTER TABLE avro_table CHANGE A A FLOAT;
```

```
-- Add column "c" with default
ALTER TABLE avro_table ADD COLUMNS (c int);
ALTER TABLE avro_table SET TBLPROPERTIES (
  'avro.schema.literal'='{
    "type": "record",
    "name": "my_record",
    "fields": [
      {"name": "a", "type": "int"},
      {"name": "b", "type": "string"},
      {"name": "c", "type": "int", "default": 10}
    ]}');
```

Once again in `impala-shell`, you can query the Avro table based on its latest schema definition. Because the table metadata was changed outside of Impala, you issue a `REFRESH` statement first so that Impala has up-to-date metadata for the table.

```
[localhost:21000] > refresh avro_table;
[localhost:21000] > select * from avro_table;
+---+------+----+
| a | b    | c  |
+---+------+----+
| 1 | avro | 10 |
+---+------+----+
```

Data Type Considerations for Avro Tables

The Avro format defines a set of data types whose names differ from the names of the corresponding Impala data types. If you are preparing Avro files using other Hadoop components such as Pig or MapReduce, you might need to work with the type names defined by Avro. The following figure lists the Avro-defined types and the equivalent types in Impala.

```
Primitive Types (Avro -> Impala)
--------------------------------
STRING -> STRING
STRING -> CHAR
STRING -> VARCHAR
INT -> INT
BOOLEAN -> BOOLEAN
LONG ->  BIGINT
FLOAT ->  FLOAT
DOUBLE -> DOUBLE

Logical Types
-------------
BYTES + logicalType = "decimal" -> DECIMAL

Avro Types with No Impala Equivalent
------------------------------------
RECORD, MAP, ARRAY, UNION,  ENUM, FIXED, NULL

Impala Types with No Avro Equivalent
------------------------------------
TIMESTAMP
```

The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In CDH 5.7 / Impala 2.5 and higher, Impala truncates `CHAR` and `VARCHAR` values in Avro tables to (2**31)-1 bytes. If a query encounters a `STRING` value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Query Performance for Impala Avro Tables

In general, expect query performance with Avro tables to be faster than with tables using text data, but slower than with Parquet tables. See Using the Parquet File Format with Impala Tables on page 643 for information about using the Parquet file format for high-performance analytic queries.

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Using the RCFile File Format with Impala Tables

Impala supports using RCFile data files.

Table 5: RCFile Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
RCFile	Structured	Snappy, gzip, deflate, bzip2	Yes.	No. Import data by using LOAD DATA on data files already in the right format, or use INSERT in Hive followed by REFRESH *table_name* in Impala.

Creating RCFile Tables and Loading Data

If you do not have an existing data file to use, begin by creating one in the appropriate format.

To create an RCFile table:

In the `impala-shell` interpreter, issue a command similar to:

```
create table rcfile_table (column_specs) stored as rcfile;
```

Because Impala can query some kinds of tables that it cannot currently write to, after creating tables of certain file formats, you might use the Hive shell to load the data. See How Impala Works with Hadoop File Formats on page 634 for details. After loading data into a table through Hive or other mechanism outside of Impala, issue a REFRESH *table_name* statement the next time you connect to the Impala node, before querying the table, to make Impala recognize the new data.

For example, here is how you might create some RCFile tables in Impala (by specifying the columns explicitly, or cloning the structure of another table), load data through Hive, and query them through Impala:

```
$ impala-shell -i localhost
[localhost:21000] > create table rcfile_table (x int) stored as rcfile;
[localhost:21000] > create table rcfile_clone like some_other_table stored as rcfile;
[localhost:21000] > quit;

$ hive
hive> insert into table rcfile_table select x from some_other_table;
3 Rows loaded to rcfile_table
Time taken: 19.015 seconds
hive> quit;

$ impala-shell -i localhost
[localhost:21000] > select * from rcfile_table;
Returned 0 row(s) in 0.23s
[localhost:21000] > -- Make Impala recognize the data loaded through Hive;
[localhost:21000] > refresh rcfile_table;
[localhost:21000] > select * from rcfile_table;
+---+
| x |
+---+
```

```
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.23s
```

Complex type considerations: Although you can create tables in this file format using the complex types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher, currently, Impala can query these types only in Parquet tables. The one exception to the preceding rule is COUNT(*) queries on RCFile tables that include complex types. Such queries are allowed in CDH 5.8 / Impala 2.6 and higher.

Enabling Compression for RCFile Tables

You may want to enable compression on existing tables. Enabling compression provides performance gains in most cases and is supported for RCFile tables. For example, to enable Snappy compression, you would specify the following additional settings when loading data through the Hive shell:

```
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> INSERT OVERWRITE TABLE new_table SELECT * FROM old_table;
```

If you are converting partitioned tables, you must complete additional steps. In such a case, specify additional settings similar to the following:

```
hive> CREATE TABLE new_table (your_cols) PARTITIONED BY (partition_cols) STORED AS
new_format;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE new_table PARTITION(comma_separated_partition_cols) SELECT
 * FROM old_table;
```

Remember that Hive does not require that you specify a source format for it. Consider the case of converting a table with two partition columns called year and month to a Snappy compressed RCFile. Combining the components outlined previously to complete this table conversion, you would specify settings similar to the following:

```
hive> CREATE TABLE tbl_rc (int_col INT, string_col STRING) STORED AS RCFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_rc SELECT * FROM tbl;
```

To complete a similar process for a table that includes partitions, you would specify settings similar to the following:

```
hive> CREATE TABLE tbl_rc (int_col INT, string_col STRING) PARTITIONED BY (year INT)
STORED AS RCFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_rc PARTITION(year) SELECT * FROM tbl;
```

Note:

The compression type is specified in the following command:

```
SET
mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

You could elect to specify alternative codecs such as `GzipCodec` here.

Query Performance for Impala RCFile Tables

In general, expect query performance with RCFile tables to be faster than with tables using text data, but slower than with Parquet tables. See Using the Parquet File Format with Impala Tables on page 643 for information about using the Parquet file format for high-performance analytic queries.

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Using the SequenceFile File Format with Impala Tables

Impala supports using SequenceFile data files.

Table 6: SequenceFile Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
SequenceFile	Structured	Snappy, gzip, deflate, bzip2	Yes.	No. Import data by using LOAD DATA on data files already in the right format, or use INSERT in Hive followed by REFRESH *table_name* in Impala.

Creating SequenceFile Tables and Loading Data

If you do not have an existing data file to use, begin by creating one in the appropriate format.

To create a SequenceFile table:

In the `impala-shell` interpreter, issue a command similar to:

```
create table sequencefile_table (column_specs) stored as sequencefile;
```

Because Impala can query some kinds of tables that it cannot currently write to, after creating tables of certain file formats, you might use the Hive shell to load the data. See How Impala Works with Hadoop File Formats on page 634 for details. After loading data into a table through Hive or other mechanism outside of Impala, issue a REFRESH *table_name* statement the next time you connect to the Impala node, before querying the table, to make Impala recognize the new data.

For example, here is how you might create some SequenceFile tables in Impala (by specifying the columns explicitly, or cloning the structure of another table), load data through Hive, and query them through Impala:

```
$ impala-shell -i localhost
[localhost:21000] > create table seqfile_table (x int) stored as sequencefile;
[localhost:21000] > create table seqfile_clone like some_other_table stored as
sequencefile;
[localhost:21000] > quit;

$ hive
hive> insert into table seqfile_table select x from some_other_table;
3 Rows loaded to seqfile_table
Time taken: 19.047 seconds
hive> quit;

$ impala-shell -i localhost
[localhost:21000] > select * from seqfile_table;
Returned 0 row(s) in 0.23s
[localhost:21000] > -- Make Impala recognize the data loaded through Hive;
[localhost:21000] > refresh seqfile_table;
[localhost:21000] > select * from seqfile_table;
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.23s
```

Complex type considerations: Although you can create tables in this file format using the complex types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher, currently, Impala can query these types only in Parquet tables. The one exception to the preceding rule is COUNT(*) queries on RCFile tables that include complex types. Such queries are allowed in CDH 5.8 / Impala 2.6 and higher.

Enabling Compression for SequenceFile Tables

You may want to enable compression on existing tables. Enabling compression provides performance gains in most cases and is supported for SequenceFile tables. For example, to enable Snappy compression, you would specify the following additional settings when loading data through the Hive shell:

```
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> insert overwrite table new_table select * from old_table;
```

If you are converting partitioned tables, you must complete additional steps. In such a case, specify additional settings similar to the following:

```
hive> create table new_table (your_cols) partitioned by (partition_cols) stored as
new_format;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> insert overwrite table new_table partition(comma_separated_partition_cols) select
 * from old_table;
```

Remember that Hive does not require that you specify a source format for it. Consider the case of converting a table with two partition columns called year and month to a Snappy compressed SequenceFile. Combining the components outlined previously to complete this table conversion, you would specify settings similar to the following:

```
hive> create table TBL_SEQ (int_col int, string_col string) STORED AS SEQUENCEFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

```
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_seq SELECT * FROM tbl;
```

To complete a similar process for a table that includes partitions, you would specify settings similar to the following:

```
hive> CREATE TABLE tbl_seq (int_col INT, string_col STRING) PARTITIONED BY (year INT)
STORED AS SEQUENCEFILE;
hive> SET hive.exec.compress.output=true;
hive> SET mapred.max.split.size=256000000;
hive> SET mapred.output.compression.type=BLOCK;
hive> SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
hive> SET hive.exec.dynamic.partition.mode=nonstrict;
hive> SET hive.exec.dynamic.partition=true;
hive> INSERT OVERWRITE TABLE tbl_seq PARTITION(year) SELECT * FROM tbl;
```



Note:

The compression type is specified in the following command:

```
SET
mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;
```

You could elect to specify alternative codecs such as `GzipCodec` here.

Query Performance for Impala SequenceFile Tables

In general, expect query performance with SequenceFile tables to be faster than with tables using text data, but slower than with Parquet tables. See Using the Parquet File Format with Impala Tables on page 643 for information about using the Parquet file format for high-performance analytic queries.

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Using Impala to Query Kudu Tables

You can use Impala to query tables stored by Apache Kudu. This capability allows convenient access to a storage system that is tuned for different kinds of workloads than the default with Impala.

By default, Impala tables are stored on HDFS using data files with various file formats. HDFS files are ideal for bulk loads (append operations) and queries using full-table scans, but do not support in-place updates or deletes. Kudu is an alternative storage engine used by Impala which can do both in-place updates (for mixed read/write workloads) and fast scans (for data-warehouse/analytic operations). Using Kudu tables with Impala can simplify the ETL pipeline by avoiding extra steps to segregate and reorganize newly arrived data.

Certain Impala SQL statements and clauses, such as `DELETE`, `UPDATE`, `UPSERT`, and `PRIMARY KEY` work only with Kudu tables. Other statements and clauses, such as `LOAD DATA`, `TRUNCATE TABLE`, and `INSERT OVERWRITE`, are not applicable to Kudu tables.

Benefits of Using Kudu Tables with Impala

The combination of Kudu and Impala works best for tables where scan performance is important, but data arrives continuously, in small batches, or needs to be updated without being completely replaced. HDFS-backed tables can require substantial overhead to replace or reorganize data files as new data arrives. Impala can perform efficient lookups and scans within Kudu tables, and Impala can also perform update or delete operations efficiently. You can also use the Kudu Java, C++, and Python APIs to do ingestion or transformation operations outside of Impala, and Impala can query the current data at any time.

Configuring Impala for Use with Kudu

The `-kudu_master_hosts` configuration property must be set correctly for the `impalad` daemon, for `CREATE TABLE ... STORED AS KUDU` statements to connect to the appropriate Kudu server. Typically, the required value for this setting is *`kudu_host`*`:7051`. In a high-availability Kudu deployment, specify the names of multiple Kudu hosts separated by commas.

If the `-kudu_master_hosts` configuration property is not set, you can still associate the appropriate value for each table by specifying a `TBLPROPERTIES('kudu.master_addresses')` clause in the `CREATE TABLE` statement or changing the `TBLPROPERTIES('kudu.master_addresses')` value with an `ALTER TABLE` statement.

If you are using Cloudera Manager, for each Impala service, navigate to the **Configuration** tab and specify the Kudu service you want to use in the **Kudu Service** field.

Cluster Topology for Kudu Tables

With HDFS-backed tables, you are typically concerned with the number of DataNodes in the cluster, how many and how large HDFS data files are read during a query, and therefore the amount of work performed by each DataNode and the network communication to combine intermediate results and produce the final result set.

With Kudu tables, the topology considerations are different, because:

- The underlying storage is managed and organized by Kudu, not represented as HDFS data files.
- Kudu handles some of the underlying mechanics of partitioning the data. You can specify the partitioning scheme with combinations of hash and range partitioning, so that you can decide how much effort to expend to manage the partitions as new data arrives. For example, you can construct partitions that apply to date ranges rather than a separate partition for each day or each hour.
- Data is physically divided based on units of storage called ***tablets***. Tablets are stored by ***tablet servers***. Each tablet server can store multiple tablets, and each tablet is replicated across multiple tablet servers, managed automatically

by Kudu. Where practical, co-locate the tablet servers on the same hosts as the Impala daemons, although that is not required.

Kudu Replication Factor

By default, Kudu tables created through Impala use a tablet replication factor of 3. To change the replication factor for a Kudu table, specify the replication factor using the TBLPROPERTIES in the CREATE TABLE Statement statement as below where *n* is the replication factor you want to use:

```
TBLPROPERTIES ('kudu.num_tablet_replicas' = 'n')
```

The number of replicas for a Kudu table must be odd.

Altering the `kudu.num_tablet_replicas` property after table creation currently has no effect.

Impala DDL Enhancements for Kudu Tables (CREATE TABLE and ALTER TABLE)

You can use the Impala `CREATE TABLE` and `ALTER TABLE` statements to create and fine-tune the characteristics of Kudu tables. Because Kudu tables have features and properties that do not apply to other kinds of Impala tables, familiarize yourself with Kudu-related concepts and syntax first. For the general syntax of the `CREATE TABLE` statement for Kudu tables, see CREATE TABLE Statement on page 234.

Primary Key Columns for Kudu Tables

Kudu tables introduce the notion of primary keys to Impala for the first time. The primary key is made up of one or more columns, whose values are combined and used as a lookup key during queries. The tuple represented by these columns must be unique and cannot contain any `NULL` values, and can never be updated once inserted. For a Kudu table, all the partition key columns must come from the set of primary key columns.

The primary key has both physical and logical aspects:

- On the physical side, it is used to map the data values to particular tablets for fast retrieval. Because the tuples formed by the primary key values are unique, the primary key columns are typically highly selective.
- On the logical side, the uniqueness constraint allows you to avoid duplicate data in a table. For example, if an `INSERT` operation fails partway through, only some of the new rows might be present in the table. You can re-run the same `INSERT`, and only the missing rows will be added. Or if data in the table is stale, you can run an `UPSERT` statement that brings the data up to date, without the possibility of creating duplicate copies of existing rows.



Note:

Impala only allows `PRIMARY KEY` clauses and `NOT NULL` constraints on columns for Kudu tables. These constraints are enforced on the Kudu side.

Kudu-Specific Column Attributes for CREATE TABLE

For the general syntax of the `CREATE TABLE` statement for Kudu tables, see CREATE TABLE Statement on page 234. The following sections provide more detail for some of the Kudu-specific keywords you can use in column definitions.

The column list in a `CREATE TABLE` statement can include the following attributes, which only apply to Kudu tables:

```
  PRIMARY KEY
| [NOT] NULL
| ENCODING codec
| COMPRESSION algorithm
```

```
| DEFAULT constant_expression
| BLOCK_SIZE number
```

See the following sections for details about each column attribute.

PRIMARY KEY Attribute

The primary key for a Kudu table is a column, or set of columns, that uniquely identifies every row. The primary key value also is used as the natural sort order for the values from the table. The primary key value for each row is based on the combination of values for the columns.

Because all of the primary key columns must have non-null values, specifying a column in the `PRIMARY KEY` clause implicitly adds the `NOT NULL` attribute to that column.

The primary key columns must be the first ones specified in the `CREATE TABLE` statement. For a single-column primary key, you can include a `PRIMARY KEY` attribute inline with the column definition. For a multi-column primary key, you include a `PRIMARY KEY (c1, c2, ...)` clause as a separate entry at the end of the column list.

You can specify the `PRIMARY KEY` attribute either inline in a single column definition, or as a separate clause at the end of the column list:

```
CREATE TABLE pk_inline
(
  col1 BIGINT PRIMARY KEY,
  col2 STRING,
  col3 BOOLEAN
) PARTITION BY HASH(col1) PARTITIONS 2 STORED AS KUDU;

CREATE TABLE pk_at_end
(
  col1 BIGINT,
  col2 STRING,
  col3 BOOLEAN,
  PRIMARY KEY (col1)
) PARTITION BY HASH(col1) PARTITIONS 2 STORED AS KUDU;
```

When the primary key is a single column, these two forms are equivalent. If the primary key consists of more than one column, you must specify the primary key using a separate entry in the column list:

```
CREATE TABLE pk_multiple_columns
(
  col1 BIGINT,
  col2 STRING,
  col3 BOOLEAN,
  PRIMARY KEY (col1, col2)
) PARTITION BY HASH(col2) PARTITIONS 2 STORED AS KUDU;
```

The `SHOW CREATE TABLE` statement always represents the `PRIMARY KEY` specification as a separate item in the column list:

```
CREATE TABLE inline_pk_rewritten (id BIGINT PRIMARY KEY, s STRING)
  PARTITION BY HASH(id) PARTITIONS 2 STORED AS KUDU;

SHOW CREATE TABLE inline_pk_rewritten;
+-------------------------------------------------------------------------------+
| result                                                                        |
+-------------------------------------------------------------------------------+
| CREATE TABLE user.inline_pk_rewritten (                                       |
|   id BIGINT NOT NULL ENCODING AUTO_ENCODING COMPRESSION DEFAULT_COMPRESSION,  |
|   s STRING NULL ENCODING AUTO_ENCODING COMPRESSION DEFAULT_COMPRESSION,       |
|   PRIMARY KEY (id)                                                            |
| )                                                                             |
| PARTITION BY HASH (id) PARTITIONS 2                                           |
| STORED AS KUDU                                                                |
```

```
| TBLPROPERTIES ('kudu.master_addresses'='host.example.com')                    |
+-------------------------------------------------------------------------------+
```

The notion of primary key only applies to Kudu tables. Every Kudu table requires a primary key. The primary key consists of one or more columns. You must specify any primary key columns first in the column list.

The contents of the primary key columns cannot be changed by an UPDATE or UPSERT statement. Including too many columns in the primary key (more than 5 or 6) can also reduce the performance of write operations. Therefore, pick the most selective and most frequently tested non-null columns for the primary key specification. If a column must always have a value, but that value might change later, leave it out of the primary key and use a NOT NULL clause for that column instead. If an existing row has an incorrect or outdated key column value, delete the old row and insert an entirely new row with the correct primary key.

NULL | NOT NULL Attribute

For Kudu tables, you can specify which columns can contain nulls or not. This constraint offers an extra level of consistency enforcement for Kudu tables. If an application requires a field to always be specified, include a NOT NULL clause in the corresponding column definition, and Kudu prevents rows from being inserted with a NULL in that column.

For example, a table containing geographic information might require the latitude and longitude coordinates to always be specified. Other attributes might be allowed to be NULL. For example, a location might not have a designated place name, its altitude might be unimportant, and its population might be initially unknown, to be filled in later.

Because all of the primary key columns must have non-null values, specifying a column in the PRIMARY KEY clause implicitly adds the NOT NULL attribute to that column.

For non-Kudu tables, Impala allows any column to contain NULL values, because it is not practical to enforce a “not null” constraint on HDFS data files that could be prepared using external tools and ETL processes.

```
CREATE TABLE required_columns
(
  id BIGINT PRIMARY KEY,
  latitude DOUBLE NOT NULL,
  longitude DOUBLE NOT NULL,
  place_name STRING,
  altitude DOUBLE,
  population BIGINT
) PARTITION BY HASH(id) PARTITIONS 2 STORED AS KUDU;
```

During performance optimization, Kudu can use the knowledge that nulls are not allowed to skip certain checks on each input row, speeding up queries and join operations. Therefore, specify NOT NULL constraints when appropriate.

The NULL clause is the default condition for all columns that are not part of the primary key. You can omit it, or specify it to clarify that you have made a conscious design decision to allow nulls in a column.

Because primary key columns cannot contain any NULL values, the NOT NULL clause is not required for the primary key columns, but you might still specify it to make your code self-describing.

DEFAULT Attribute

You can specify a default value for columns in Kudu tables. The default value can be any constant expression, for example, a combination of literal values, arithmetic and string operations. It cannot contain references to columns or non-deterministic function calls.

The following example shows different kinds of expressions for the DEFAULT clause. The requirement to use a constant value means that you can fill in a placeholder value such as NULL, empty string, 0, -1, 'N/A' and so on, but you cannot reference functions or column names. Therefore, you cannot use DEFAULT to do things such as automatically making an uppercase copy of a string value, storing Boolean values based on tests of other columns, or add or subtract one from another column representing a sequence number.

```
CREATE TABLE default_vals
```

```
(
  id BIGINT PRIMARY KEY,
  name STRING NOT NULL DEFAULT 'unknown',
  address STRING DEFAULT upper('no fixed address'),
  age INT DEFAULT -1,
  earthling BOOLEAN DEFAULT TRUE,
  planet_of_origin STRING DEFAULT 'Earth',
  optional_col STRING DEFAULT NULL
) PARTITION BY HASH(id) PARTITIONS 2 STORED AS KUDU;
```



Note:

When designing an entirely new schema, prefer to use `NULL` as the placeholder for any unknown or missing values, because that is the universal convention among database systems. Null values can be stored efficiently, and easily checked with the `IS NULL` or `IS NOT NULL` operators. The `DEFAULT` attribute is appropriate when ingesting data that already has an established convention for representing unknown or missing values, or where the vast majority of rows have some common non-null value.

ENCODING Attribute

Each column in a Kudu table can optionally use an encoding, a low-overhead form of compression that reduces the size on disk, then requires additional CPU cycles to reconstruct the original values during queries. Typically, highly compressible data benefits from the reduced I/O to read the data back from disk.

The encoding keywords that Impala recognizes are:

- `AUTO_ENCODING`: use the default encoding based on the column type, which are bitshuffle for the numeric type columns and dictionary for the string type columns.
- `PLAIN_ENCODING`: leave the value in its original binary format.
- `RLE`: compress repeated values (when sorted in primary key order) by including a count.
- `DICT_ENCODING`: when the number of different string values is low, replace the original string with a numeric ID.
- `BIT_SHUFFLE`: rearrange the bits of the values to efficiently compress sequences of values that are identical or vary only slightly based on primary key order. The resulting encoded data is also compressed with LZ4.
- `PREFIX_ENCODING`: compress common prefixes in string values; mainly for use internally within Kudu.

The following example shows the Impala keywords representing the encoding types. (The Impala keywords match the symbolic names used within Kudu.) For usage guidelines on the different kinds of encoding, see the Kudu documentation. The `DESCRIBE` output shows how the encoding is reported after the table is created, and that omitting the encoding (in this case, for the `ID` column) is the same as specifying `DEFAULT_ENCODING`.

```
CREATE TABLE various_encodings
(
  id BIGINT PRIMARY KEY,
  c1 BIGINT ENCODING PLAIN_ENCODING,
  c2 BIGINT ENCODING AUTO_ENCODING,
  c3 TINYINT ENCODING BIT_SHUFFLE,
  c4 DOUBLE ENCODING BIT_SHUFFLE,
  c5 BOOLEAN ENCODING RLE,
  c6 STRING ENCODING DICT_ENCODING,
  c7 STRING ENCODING PREFIX_ENCODING
) PARTITION BY HASH(id) PARTITIONS 2 STORED AS KUDU;

-- Some columns are omitted from the output for readability.
describe various_encodings;
+------+---------+-------------+----------+-----------------+
| name | type    | primary_key | nullable | encoding        |
+------+---------+-------------+----------+-----------------+
| id   | bigint  | true        | false    | AUTO_ENCODING   |
```

```
| c1   | bigint  | false       | true     | PLAIN_ENCODING  |
| c2   | bigint  | false       | true     | AUTO_ENCODING   |
| c3   | tinyint | false       | true     | BIT_SHUFFLE     |
| c4   | double  | false       | true     | BIT_SHUFFLE     |
| c5   | boolean | false       | true     | RLE             |
| c6   | string  | false       | true     | DICT_ENCODING   |
| c7   | string  | false       | true     | PREFIX_ENCODING |
+------+---------+-------------+----------+-----------------+
```

COMPRESSION Attribute

You can specify a compression algorithm to use for each column in a Kudu table. This attribute imposes more CPU overhead when retrieving the values than the `ENCODING` attribute does. Therefore, use it primarily for columns with long strings that do not benefit much from the less-expensive `ENCODING` attribute.

The choices for `COMPRESSION` are `LZ4`, `SNAPPY`, and `ZLIB`.

> **Note:**
>
> Columns that use the `BITSHUFFLE` encoding are already compressed using `LZ4`, and so typically do not need any additional `COMPRESSION` attribute.

The following example shows design considerations for several `STRING` columns with different distribution characteristics, leading to choices for both the `ENCODING` and `COMPRESSION` attributes. The `country` values come from a specific set of strings, therefore this column is a good candidate for dictionary encoding. The `post_id` column contains an ascending sequence of integers, where several leading bits are likely to be all zeroes, therefore this column is a good candidate for bitshuffle encoding. The body column and the corresponding columns for translated versions tend to be long unique strings that are not practical to use with any of the encoding schemes, therefore they employ the `COMPRESSION` attribute instead. The ideal compression codec in each case would require some experimentation to determine how much space savings it provided and how much CPU overhead it added, based on real-world data.

```
CREATE TABLE blog_posts
(
  user_id STRING ENCODING DICT_ENCODING,
  post_id BIGINT ENCODING BIT_SHUFFLE,
  subject STRING ENCODING PLAIN_ENCODING,
  body STRING COMPRESSION LZ4,
  spanish_translation STRING COMPRESSION SNAPPY,
  esperanto_translation STRING COMPRESSION ZLIB,
  PRIMARY KEY (user_id, post_id)
) PARTITION BY HASH(user_id, post_id) PARTITIONS 2 STORED AS KUDU;
```

BLOCK_SIZE Attribute

Although Kudu does not use HDFS files internally, and thus is not affected by the HDFS block size, it does have an underlying unit of I/O called the ***block size***. The `BLOCK_SIZE` attribute lets you set the block size for any column.

> **Note:** The block size attribute is a relatively advanced feature. This is an unsupported feature and is considered experimental.

Partitioning for Kudu Tables

Kudu tables use special mechanisms to distribute data among the underlying tablet servers. Although we refer to such tables as partitioned tables, they are distinguished from traditional Impala partitioned tables by use of different clauses on the `CREATE TABLE` statement. Kudu tables use `PARTITION BY`, `HASH`, `RANGE`, and range specification clauses rather than the `PARTITIONED BY` clause for HDFS-backed tables, which specifies only a column name and creates a new partition for each different value.

For background information and architectural details about the Kudu partitioning mechanism, see the Kudu white paper, section 3.2.



Note:

The Impala DDL syntax for Kudu tables is different than in early Kudu versions, which used an experimental fork of the Impala code. For example, the DISTRIBUTE BY clause is now PARTITION BY, the INTO *n* BUCKETS clause is now PARTITIONS *n* and the range partitioning syntax is reworked to replace the SPLIT ROWS clause with more expressive syntax involving comparison operators.

Hash Partitioning

Hash partitioning is the simplest type of partitioning for Kudu tables. For hash-partitioned Kudu tables, inserted rows are divided up between a fixed number of "buckets" by applying a hash function to the values of the columns specified in the HASH clause. Hashing ensures that rows with similar values are evenly distributed, instead of clumping together all in the same bucket. Spreading new rows across the buckets this way lets insertion operations work in parallel across multiple tablet servers. Separating the hashed values can impose additional overhead on queries, where queries with range-based predicates might have to read multiple tablets to retrieve all the relevant values.

```
-- 1M rows with 50 hash partitions = approximately 20,000 rows per partition.
-- The values in each partition are not sequential, but rather based on a hash function.
-- Rows 1, 99999, and 123456 might be in the same partition.
CREATE TABLE million_rows (id string primary key, s string)
  PARTITION BY HASH(id) PARTITIONS 50
  STORED AS KUDU;

-- Because the ID values are unique, we expect the rows to be roughly
-- evenly distributed between the buckets in the destination table.
INSERT INTO million_rows SELECT * FROM billion_rows ORDER BY id LIMIT 1e6;
```



Note:

The largest number of buckets that you can create with a PARTITIONS clause varies depending on the number of tablet servers in the cluster, while the smallest is 2. For simplicity, some of the simple CREATE TABLE statements throughout this section use PARTITIONS 2 to illustrate the minimum requirements for a Kudu table. For large tables, prefer to use roughly 10 partitions per server in the cluster.

Range Partitioning

Range partitioning lets you specify partitioning precisely, based on single values or ranges of values within one or more columns. You add one or more RANGE clauses to the CREATE TABLE statement, following the PARTITION BY clause.

Range-partitioned Kudu tables use one or more range clauses, which include a combination of constant expressions, VALUE or VALUES keywords, and comparison operators. (This syntax replaces the SPLIT ROWS clause used with early Kudu versions.) For the full syntax, see CREATE TABLE Statement on page 234.

```
-- 50 buckets, all for IDs beginning with a lowercase letter.
-- Having only a single range enforces the allowed range of values
-- but does not add any extra parallelism.
create table million_rows_one_range (id string primary key, s string)
  partition by hash(id) partitions 50,
  range (partition 'a' <= values < '{')
  stored as kudu;

-- 50 buckets for IDs beginning with a lowercase letter
-- plus 50 buckets for IDs beginning with an uppercase letter.
-- Total number of buckets = number in the PARTITIONS clause x number of ranges.
-- We are still enforcing constraints on the primary key values
```

```
-- allowed in the table, and the 2 ranges provide better parallelism
-- as rows are inserted or the table is scanned.
create table million_rows_two_ranges (id string primary key, s string)
  partition by hash(id) partitions 50,
  range (partition 'a' <= values < '{', partition 'A' <= values < '[')
  stored as kudu;

-- Same as previous table, with an extra range covering the single key value '00000'.
create table million_rows_three_ranges (id string primary key, s string)
  partition by hash(id) partitions 50,
  range (partition 'a' <= values < '{', partition 'A' <= values < '[', partition value
 = '00000')
  stored as kudu;

-- The range partitioning can be displayed with a SHOW command in impala-shell.
show range partitions million_rows_three_ranges;
+---------------------+
| RANGE (id)          |
+---------------------+
| VALUE = "00000"     |
| "A" <= VALUES < "[" |
| "a" <= VALUES < "{" |
+---------------------+
```



Note:

When defining ranges, be careful to avoid “fencepost errors” where values at the extreme ends might be included or omitted by accident. For example, in the tables defined in the preceding code listings, the range `"a" <= VALUES < "{"` ensures that any values starting with z, such as za or zzz or zzz-ZZZ, are all included, by using a less-than operator for the smallest value after all the values starting with z.

For range-partitioned Kudu tables, an appropriate range must exist before a data value can be created in the table. Any INSERT, UPDATE, or UPSERT statements fail if they try to create column values that fall outside the specified ranges. The error checking for ranges is performed on the Kudu side; Impala passes the specified range information to Kudu, and passes back any error or warning if the ranges are not valid. (A nonsensical range specification causes an error for a DDL statement, but only a warning for a DML statement.)

Ranges can be non-contiguous:

```
partition by range (year) (partition 1885 <= values <= 1889, partition 1893 <= values
<= 1897)

partition by range (letter_grade) (partition value = 'A', partition value = 'B',
  partition value = 'C', partition value = 'D', partition value = 'F')
```

The `ALTER TABLE` statement with the `ADD PARTITION` or `DROP PARTITION` clauses can be used to add or remove ranges from an existing Kudu table.

```
ALTER TABLE foo ADD PARTITION 30 <= VALUES < 50;
ALTER TABLE foo DROP PARTITION 1 <= VALUES < 5;
```

When a range is added, the new range must not overlap with any of the previous ranges; that is, it can only fill in gaps within the previous ranges.

```
alter table test_scores add range partition value = 'E';
```

```
alter table year_ranges add range partition 1890 <= values < 1893;
```

When a range is removed, all the associated rows in the table are deleted. (This is true whether the table is internal or external.)

```
alter table test_scores drop range partition value = 'E';
alter table year_ranges drop range partition 1890 <= values < 1893;
```

Kudu tables can also use a combination of hash and range partitioning.

```
partition by hash (school) partitions 10,
  range (letter_grade) (partition value = 'A', partition value = 'B',
    partition value = 'C', partition value = 'D', partition value = 'F')
```

Working with Partitioning in Kudu Tables

To see the current partitioning scheme for a Kudu table, you can use the `SHOW CREATE TABLE` statement or the `SHOW PARTITIONS` statement. The `CREATE TABLE` syntax displayed by this statement includes all the hash, range, or both clauses that reflect the original table structure plus any subsequent `ALTER TABLE` statements that changed the table structure.

To see the underlying buckets and partitions for a Kudu table, use the `SHOW TABLE STATS` or `SHOW PARTITIONS` statement.

Handling Date, Time, or Timestamp Data with Kudu

In CDH 5.12 / Impala 2.9 and higher, you can include `TIMESTAMP` columns in Kudu tables, instead of representing the date and time as a `BIGINT` value. The behavior of `TIMESTAMP` for Kudu tables has some special considerations:

- Any nanoseconds in the original 96-bit value produced by Impala are not stored, because Kudu represents date/time columns using 64-bit values. The nanosecond portion of the value is rounded, not truncated. Therefore, a `TIMESTAMP` value that you store in a Kudu table might not be bit-for-bit identical to the value returned by a query.
- The conversion between the Impala 96-bit representation and the Kudu 64-bit representation introduces some performance overhead when reading or writing `TIMESTAMP` columns. You can minimize the overhead during writes by performing inserts through the Kudu API. Because the overhead during reads applies to each query, you might continue to use a `BIGINT` column to represent date/time values in performance-critical applications.
- The Impala `TIMESTAMP` type has a narrower range for years than the underlying Kudu data type. Impala can represent years 1400-9999. If year values outside this range are written to a Kudu table by a non-Impala client, Impala returns `NULL` by default when reading those `TIMESTAMP` values during a query. Or, if the `ABORT_ON_ERROR` query option is enabled, the query fails when it encounters a value with an out-of-range year.

```
--- Make a table representing a date/time value as TIMESTAMP.
-- The strings representing the partition bounds are automatically
-- cast to TIMESTAMP values.
create table native_timestamp(id bigint, when_exactly timestamp, event string, primary
 key (id, when_exactly))
  partition by hash (id) partitions 20,
  range (when_exactly)
  (
    partition '2015-01-01' <= values < '2016-01-01',
    partition '2016-01-01' <= values < '2017-01-01',
    partition '2017-01-01' <= values < '2018-01-01'
  )
  stored as kudu;
```

```
insert into native_timestamp values (12345, now(), 'Working on doc examples');

select * from native_timestamp;
+-------+-------------------------------+-------------------------+
| id    | when_exactly                  | event                   |
+-------+-------------------------------+-------------------------+
| 12345 | 2017-05-31 16:27:42.667542000 | Working on doc examples |
+-------+-------------------------------+-------------------------+
```

Because Kudu tables have some performance overhead to convert TIMESTAMP columns to the Impala 96-bit internal representation, for performance-critical applications you might store date/time information as the number of seconds, milliseconds, or microseconds since the Unix epoch date of January 1, 1970. Specify the column as BIGINT in the Impala CREATE TABLE statement, corresponding to an 8-byte integer (an int64) in the underlying Kudu table). Then use Impala date/time conversion functions as necessary to produce a numeric, TIMESTAMP, or STRING value depending on the context.

For example, the unix_timestamp() function returns an integer result representing the number of seconds past the epoch. The now() function produces a TIMESTAMP representing the current date and time, which can be passed as an argument to unix_timestamp(). And string literals representing dates and date/times can be cast to TIMESTAMP, and from there converted to numeric values. The following examples show how you might store a date/time column as BIGINT in a Kudu table, but still use string literals and TIMESTAMP values for convenience.

```
-- now() returns a TIMESTAMP and shows the format for string literals you can cast to
TIMESTAMP.
select now();
+-------------------------------+
| now()                         |
+-------------------------------+
| 2017-01-25 23:50:10.132385000 |
+-------------------------------+

-- unix_timestamp() accepts either a TIMESTAMP or an equivalent string literal.
select unix_timestamp(now());
+------------------+
| unix_timestamp() |
+------------------+
| 1485386670       |
+------------------+

select unix_timestamp('2017-01-01');
+------------------------------+
| unix_timestamp('2017-01-01') |
+------------------------------+
| 1483228800                   |
+------------------------------+

-- Make a table representing a date/time value as BIGINT.
-- Construct 1 range partition and 20 associated hash partitions for each year.
-- Use date/time conversion functions to express the ranges as human-readable dates.
create table time_series(id bigint, when_exactly bigint, event string, primary key (id,
 when_exactly))
  partition by hash (id) partitions 20,
  range (when_exactly)
  (
    partition unix_timestamp('2015-01-01') <= values < unix_timestamp('2016-01-01'),
    partition unix_timestamp('2016-01-01') <= values < unix_timestamp('2017-01-01'),
    partition unix_timestamp('2017-01-01') <= values < unix_timestamp('2018-01-01')
  )
  stored as kudu;

-- On insert, we can transform a human-readable date/time into a numeric value.
insert into time_series values (12345, unix_timestamp('2017-01-25 23:24:56'), 'Working
 on doc examples');

-- On retrieval, we can examine the numeric date/time value or turn it back into a string
 for readability.
select id, when_exactly, from_unixtime(when_exactly) as 'human-readable date/time',
```

```
event
  from time_series order by when_exactly limit 100;
+-------+-------------+-------------------------+-------------------------+
| id    | when_exactly | human-readable date/time | event                   |
+-------+-------------+-------------------------+-------------------------+
| 12345 | 1485386696   | 2017-01-25 23:24:56      | Working on doc examples |
+-------+-------------+-------------------------+-------------------------+
```



Note:

If you do high-precision arithmetic involving numeric date/time values, when dividing millisecond values by 1000, or microsecond values by 1 million, always cast the integer numerator to a `DECIMAL` with sufficient precision and scale to avoid any rounding or loss of precision.

```
-- 1 million and 1 microseconds = 1.000001 seconds.
select microseconds,
  cast (microseconds as decimal(20,7)) / 1e6 as fractional_seconds
  from table_with_microsecond_column;
+-------------+----------------------+
| microseconds | fractional_seconds   |
+-------------+----------------------+
| 1000001      | 1.000001000000000000 |
+-------------+----------------------+
```

How Impala Handles Kudu Metadata



Note: This section only applies the Kudu services that are not integrated with the Hive Metastore (HMS).

By default, much of the metadata for Kudu tables is handled by the underlying storage layer. Kudu tables have less reliance on the Metastore database, and require less metadata caching on the Impala side. For example, information about partitions in Kudu tables is managed by Kudu, and Impala does not cache any block locality metadata for Kudu tables. If the Kudu service is not integrated with the Hive Metastore, Impala will manage Kudu table metadata in the Hive Metastore.

The `REFRESH` and `INVALIDATE METADATA` statements are needed less frequently for Kudu tables than for HDFS-backed tables. Neither statement is needed when data is added to, removed, or updated in a Kudu table, even if the changes are made directly to Kudu through a client program using the Kudu API. Run `REFRESH` *`table_name`* or `INVALIDATE METADATA` *`table_name`* for a Kudu table only after making a change to the Kudu table schema, such as adding or dropping a column.

Because Kudu manages the metadata for its own tables separately from the metastore database, there is a table name stored in the metastore database for Impala to use, and a table name on the Kudu side, and these names can be modified independently through `ALTER TABLE` statements.

To avoid potential name conflicts, the prefix `impala::` and the Impala database name are encoded into the underlying Kudu table name:

```
create database some_database;
use some_database;

create table table_name_demo (x int primary key, y int)
  partition by hash (x) partitions 2 stored as kudu;

describe formatted table_name_demo;
...
```

```
kudu.table_name  | impala::some_database.table_name_demo
```

See Kudu Tables on page 198 for examples of how to change the name of the Impala table in the metastore database, the name of the underlying Kudu table, or both.

Working with Kudu Integrated with Hive Metastore

Starting from Kudu 1.10 and Impala 3.3 in CDH 6.3, Impala supports Kudu services integrated with the Hive Metastore (HMS). See the HMS integration documentation for more details on Kudu's Hive Metastore integration.

The following are some of the changes you need to consider when working with Kudu services integrated with the HMS.

- When Kudu is integrated with the Hive Metastore, Impala must be configured to use the same HMS as Kudu.
- Since there may be no one-to-one mapping between Kudu tables and external tables, only internal tables are automatically synchronized.
- When you create a table in Kudu, Kudu will create an HMS entry for that table with the internal table type.
- When the Kudu service is integrated with the HMS, internal table entries will be created automatically in the HMS when tables are created in Kudu without Impala. To access these tables through Impala, run `INVALIDATE METADATA` statement so Impala picks up the latest metadata.

Loading Data into Kudu Tables

Kudu tables are well-suited to use cases where data arrives continuously, in small or moderate volumes. To bring data into Kudu tables, use the Impala `INSERT` and `UPSERT` statements. The `LOAD DATA` statement does not apply to Kudu tables.

Because Kudu manages its own storage layer that is optimized for smaller block sizes than HDFS, and performs its own housekeeping to keep data evenly distributed, it is not subject to the "many small files" issue and does not need explicit reorganization and compaction as the data grows over time. The partitions within a Kudu table can be specified to cover a variety of possible data distributions, instead of hardcoding a new partition for each new day, hour, and so on, which can lead to inefficient, hard-to-scale, and hard-to-manage partition schemes with HDFS tables.

Your strategy for performing ETL or bulk updates on Kudu tables should take into account the limitations on consistency for DML operations.

Make `INSERT`, `UPDATE`, and `UPSERT` operations ***idempotent***: that is, able to be applied multiple times and still produce an identical result.

If a bulk operation is in danger of exceeding capacity limits due to timeouts or high memory usage, split it into a series of smaller operations.

Avoid running concurrent ETL operations where the end results depend on precise ordering. In particular, do not rely on an `INSERT ... SELECT` statement that selects from the same table into which it is inserting, unless you include extra conditions in the `WHERE` clause to avoid reading the newly inserted rows within the same statement.

Because relationships between tables cannot be enforced by Impala and Kudu, and cannot be committed or rolled back together, do not expect transactional semantics for multi-table operations.

Impala DML Support for Kudu Tables (INSERT, UPDATE, DELETE, UPSERT)

Impala supports certain DML statements for Kudu tables only. The `UPDATE` and `DELETE` statements let you modify data within Kudu tables without rewriting substantial amounts of table data. The `UPSERT` statement acts as a combination of `INSERT` and `UPDATE`, inserting rows where the primary key does not already exist, and updating the non-primary key columns where the primary key does already exist in the table.

The INSERT statement for Kudu tables honors the unique and NOT NULL requirements for the primary key columns.

Because Impala and Kudu do not support transactions, the effects of any INSERT, UPDATE, or DELETE statement are immediately visible. For example, you cannot do a sequence of UPDATE statements and only make the changes visible after all the statements are finished. Also, if a DML statement fails partway through, any rows that were already inserted, deleted, or changed remain in the table; there is no rollback mechanism to undo the changes.

In particular, an INSERT ... SELECT statement that refers to the table being inserted into might insert more rows than expected, because the SELECT part of the statement sees some of the new rows being inserted and processes them again.



Note:

The LOAD DATA statement, which involves manipulation of HDFS data files, does not apply to Kudu tables.

Starting from CDH 5.12 / Impala 2.9, the INSERT or UPSERT operations into Kudu tables automatically add an exchange and a sort node to the plan that partitions and sorts the rows according to the partitioning/primary key scheme of the target table (unless the number of rows to be inserted is small enough to trigger single node execution). Since Kudu partitions and sorts rows on write, pre-partitioning and sorting takes some of the load off of Kudu and helps large INSERT operations to complete without timing out. However, this default behavior may slow down the end-to-end performance of the INSERT or UPSERT operations. Starting from CDH 5.13 / Impala 2.10, you can use the /* +NOCLUSTERED */ and /* +NOSHUFFLE */ hints together to disable partitioning and sorting before the rows are sent to Kudu. Additionally, since sorting may consume a large amount of memory, consider setting the MEM_LIMIT query option for those queries.

Consistency Considerations for Kudu Tables

Kudu tables have consistency characteristics such as uniqueness, controlled by the primary key columns, and non-nullable columns. The emphasis for consistency is on preventing duplicate or incomplete data from being stored in a table.

Currently, Kudu does not enforce strong consistency for order of operations, total success or total failure of a multi-row statement, or data that is read while a write operation is in progress. Changes are applied atomically to each row, but not applied as a single unit to all rows affected by a multi-row DML statement. That is, Kudu does not currently have atomic multi-row statements or isolation between statements.

If some rows are rejected during a DML operation because of a mismatch with duplicate primary key values, NOT NULL constraints, and so on, the statement succeeds with a warning. Impala still inserts, deletes, or updates the other rows that are not affected by the constraint violation.

Consequently, the number of rows affected by a DML operation on a Kudu table might be different than you expect.

Because there is no strong consistency guarantee for information being inserted into, deleted from, or updated across multiple tables simultaneously, consider denormalizing the data where practical. That is, if you run separate INSERT statements to insert related rows into two different tables, one INSERT might fail while the other succeeds, leaving the data in an inconsistent state. Even if both inserts succeed, a join query might happen during the interval between the completion of the first and second statements, and the query would encounter incomplete inconsistent data. Denormalizing the data into a single wide table can reduce the possibility of inconsistency due to multi-table operations.

Information about the number of rows affected by a DML operation is reported in impala-shell output, and in the PROFILE output, but is not currently reported to HiveServer2 clients such as JDBC or ODBC applications.

Security Considerations for Kudu Tables

Security for Kudu tables involves:

- Sentry authorization.

Access to Kudu tables must be granted to and revoked from roles with the following considerations:

- Only users with the ALL privilege on SERVER can create external Kudu tables.
- The ALL privileges on SERVER is required to specify the kudu.master_addresses property in the CREATE TABLE statements for managed tables as well as external tables.
- Access to Kudu tables is enforced at the table level and at the column level.
- The SELECT- and INSERT-specific permissions are supported.
- The DELETE, UPDATE, and UPSERT operations require the ALL privilege.

Because non-SQL APIs can access Kudu data without going through Sentry authorization, currently the Sentry support is considered preliminary and subject to change.

- Kerberos authentication. See for details.
- TLS encryption. See for details.
- Lineage tracking.
- Auditing.
- Redaction of sensitive information from log files.

Impala Query Performance for Kudu Tables

For queries involving Kudu tables, Impala can delegate much of the work of filtering the result set to Kudu, avoiding some of the I/O involved in full table scans of tables containing HDFS data files. This type of optimization is especially effective for partitioned Kudu tables, where the Impala query WHERE clause refers to one or more primary key columns that are also used as partition key columns. For example, if a partitioned Kudu table uses a HASH clause for col1 and a RANGE clause for col2, a query using a clause such as WHERE col1 IN (1,2,3) AND col2 > 100 can determine exactly which tablet servers contain relevant data, and therefore parallelize the query very efficiently.

In CDH 5.14 / Impala 2.11 and higher, Impala can push down additional information to optimize join queries involving Kudu tables. If the join clause contains predicates of the form *column = expression*, after Impala constructs a hash table of possible matching values for the join columns from the bigger table (either an HDFS table or a Kudu table), Impala can "push down" the minimum and maximum matching column values to Kudu, so that Kudu can more efficiently locate matching rows in the second (smaller) table. These min/max filters are affected by the RUNTIME_FILTER_MODE, RUNTIME_FILTER_WAIT_TIME_MS, and DISABLE_ROW_RUNTIME_FILTERING query options; the min/max filters are not affected by the RUNTIME_BLOOM_FILTER_SIZE, RUNTIME_FILTER_MIN_SIZE, RUNTIME_FILTER_MAX_SIZE, and MAX_NUM_RUNTIME_FILTERS query options.

See EXPLAIN Statement on page 271 for examples of evaluating the effectiveness of the predicate pushdown for a specific query against a Kudu table.

The TABLESAMPLE clause of the SELECT statement does not apply to a table reference derived from a view, a subquery, or anything other than a real base table. This clause only works for tables backed by HDFS or HDFS-like data files, therefore it does not apply to Kudu or HBase tables.

Using Impala to Query HBase Tables

You can use Impala to query HBase tables. This is useful for accessing any of your existing HBase tables via SQL and performing analytics over them. HDFS and Kudu tables are preferred over HBase for analytic workloads and offer superior performance. Kudu supports efficient inserts, updates and deletes of small numbers of rows and can replace HBase for most analytics-oriented use cases. See Using Impala to Query Kudu Tables on page 670 for information on using Impala with Kudu.

From the perspective of an Impala user, coming from an RDBMS background, HBase is a kind of key-value store where the value consists of multiple fields. The key is mapped to one column in the Impala table, and the various fields of the value are mapped to the other columns in the Impala table.

For background information on HBase, see the Apache HBase documentation. This is a snapshot of the Apache HBase site (including documentation) for the level of HBase that comes with CDH.

Overview of Using HBase with Impala

When you use Impala with HBase:

- You create the tables on the Impala side using the Hive shell, because the Impala `CREATE TABLE` statement currently does not support custom SerDes and some other syntax needed for these tables:
  - You designate it as an HBase table using the `STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'` clause on the Hive `CREATE TABLE` statement.
  - You map these specially created tables to corresponding tables that exist in HBase, with the clause `TBLPROPERTIES("hbase.table.name" = "`*`table_name_in_hbase`*`")` on the Hive `CREATE TABLE` statement.
  - See Examples of Querying HBase Tables from Impala on page 690 for a full example.
- You define the column corresponding to the HBase row key as a string with the `#string` keyword, or map it to a `STRING` column.
- Because Impala and Hive share the same metastore database, once you create the table in Hive, you can query or insert into it through Impala. (After creating a new table through Hive, issue the `INVALIDATE METADATA` statement in `impala-shell` to make Impala aware of the new table.)
- You issue queries against the Impala tables. For efficient queries, use `WHERE` clause to find a single key value or a range of key values wherever practical, by testing the Impala column corresponding to the HBase row key. Avoid queries that do full-table scans, which are efficient for regular Impala tables but inefficient in HBase.

To work with an HBase table from Impala, ensure that the `impala` user has read/write privileges for the HBase table, using the `GRANT` command in the HBase shell. For details about HBase security, see the Security chapter in the Apache HBase documentation.

Configuring HBase for Use with Impala

HBase works out of the box with Impala. There is no mandatory configuration needed to use these two components together.

To avoid delays if HBase is unavailable during Impala startup or after an `INVALIDATE METADATA` statement, set timeout values similar to the following in `/etc/impala/conf/hbase-site.xml` (for environments not managed by Cloudera Manager):

```
<property>
  <name>hbase.client.retries.number</name>
  <value>3</value>
```

```
</property>
<property>
  <name>hbase.rpc.timeout</name>
  <value>3000</value>
</property>
```

Currently, Cloudera Manager does not have an Impala-only override for HBase settings, so any HBase configuration change you make through Cloudera Manager would take affect for all HBase applications. Therefore, this change is not recommended on systems managed by Cloudera Manager.

Supported Data Types for HBase Columns

To understand how Impala column data types are mapped to fields in HBase, you should have some background knowledge about HBase first. You set up the mapping by running the `CREATE TABLE` statement in the Hive shell. See the Hive wiki for a starting point, and Examples of Querying HBase Tables from Impala on page 690 for examples.

HBase works as a kind of z“bit bucket”, in the sense that HBase does not enforce any typing for the key or value fields. All the type enforcement is done on the Impala side.

For best performance of Impala queries against HBase tables, most queries will perform comparisons in the `WHERE` clause against the column that corresponds to the HBase row key. When creating the table through the Hive shell, use the `STRING` data type for the column that corresponds to the HBase row key. Impala can translate predicates (through operators such as =, <, and `BETWEEN`) against this column into fast lookups in HBase, but this optimization (“predicate pushdown”) only works when that column is defined as `STRING`.

Starting in Impala 1.1, Impala also supports reading and writing to columns that are defined in the Hive `CREATE TABLE` statement using binary data types, represented in the Hive table definition using the `#binary` keyword, often abbreviated as #b. Defining numeric columns as binary can reduce the overall data volume in the HBase tables. You should still define the column that corresponds to the HBase row key as a `STRING`, to allow fast lookups using those columns.

Performance Considerations for the Impala-HBase Integration

To understand the performance characteristics of SQL queries against data stored in HBase, you should have some background knowledge about how HBase interacts with SQL-oriented systems first. See the Hive wiki for a starting point; because Impala shares the same metastore database as Hive, the information about mapping columns from Hive tables to HBase tables is generally applicable to Impala too.

Impala uses the HBase client API via Java Native Interface (JNI) to query data stored in HBase. This querying does not read HFiles directly. The extra communication overhead makes it important to choose what data to store in HBase or in HDFS, and construct efficient queries that can retrieve the HBase data efficiently:

- Use HBase table for queries that return a single row or a small range of rows, not queries that perform a full table scan of an entire table. (If a query has a HBase table and no `WHERE` clause referencing that table, that is a strong indicator that it is an inefficient query for an HBase table.)
- HBase may offer acceptable performance for storing small dimension tables where the table is small enough that executing a full table scan for every query is efficient enough. However, Kudu is almost always a superior alternative for storing dimension tables. HDFS tables are also appropriate for dimension tables that do not need to support update queries, delete queries or insert queries with small numbers of rows.

Query predicates are applied to row keys as start and stop keys, thereby limiting the scope of a particular lookup. If row keys are not mapped to string columns, then ordering is typically incorrect and comparison operations do not work. For example, if row keys are not mapped to string columns, evaluating for greater than (>) or less than (<) cannot be completed.

Predicates on non-key columns can be sent to HBase to scan as `SingleColumnValueFilters`, providing some performance gains. In such a case, HBase returns fewer rows than if those same predicates were applied using Impala. While there is some improvement, it is not as great when start and stop rows are used. This is because the number of

rows that HBase must examine is not limited as it is when start and stop rows are used. As long as the row key predicate only applies to a single row, HBase will locate and return that row. Conversely, if a non-key predicate is used, even if it only applies to a single row, HBase must still scan the entire table to find the correct result.

Interpreting EXPLAIN Output for HBase Queries

For example, here are some queries against the following Impala table, which is mapped to an HBase table. The examples show excerpts from the output of the `EXPLAIN` statement, demonstrating what things to look for to indicate an efficient or inefficient query against an HBase table.

The first column (`cust_id`) was specified as the key column in the `CREATE EXTERNAL TABLE` statement; for performance, it is important to declare this column as `STRING`. Other columns, such as `BIRTH_YEAR` and `NEVER_LOGGED_ON`, are also declared as `STRING`, rather than their "natural" types of `INT` or `BOOLEAN`, because Impala can optimize those types more effectively in HBase tables. For comparison, we leave one column, `YEAR_REGISTERED`, as `INT` to show that filtering on this column is inefficient.

```
describe hbase_table;
Query: describe hbase_table
+----------------------+--------+---------+
| name                 | type   | comment |
+----------------------+--------+---------+
| cust_id              | string |         |
| birth_year           | string |         |
| never_logged_on      | string |         |
| private_email_address | string |         |
| year_registered      | int    |         |
+----------------------+--------+---------+
```

The best case for performance involves a single row lookup using an equality comparison on the column defined as the row key:

```
explain select count(*) from hbase_table where cust_id = 'some_user@example.com';
+-----------------------------------------------------------------------------------------+
| Explain String                                                                          |
+-----------------------------------------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=1.01GB VCores=1                                 |
| WARNING: The following tables are missing relevant table and/or column statistics.      |
| hbase.hbase_table                                                                       |
|                                                                                         |
| 03:AGGREGATE [MERGE FINALIZE]                                                           |
| |  output: sum(count(*))                                                                |
| |                                                                                       |
| 02:EXCHANGE [PARTITION=UNPARTITIONED]                                                   |
| |                                                                                       |
| 01:AGGREGATE                                                                            |
| |  output: count(*)                                                                     |
| |                                                                                       |
| 00:SCAN HBASE [hbase.hbase_table]                                                       |
|    start key: some_user@example.com                                                     |
|    stop key: some_user@example.com\0                                                    |
+-----------------------------------------------------------------------------------------+
```

Another type of efficient query involves a range lookup on the row key column, using SQL operators such as greater than (or equal), less than (or equal), or `BETWEEN`. This example also includes an equality test on a non-key column; because that column is a `STRING`, Impala can let HBase perform that test, indicated by the `hbase filters:` line in the `EXPLAIN` output. Doing the filtering within HBase is more efficient than transmitting all the data to Impala and doing the filtering on the Impala side.

```
explain select count(*) from hbase_table where cust_id between 'a' and 'b'
  and never_logged_on = 'true';
+-----------------------------------------------------------------------------------------+
| Explain String                                                                          |
+-----------------------------------------------------------------------------------------+
...

| 01:AGGREGATE                                                                            |
| |  output: count(*)                                                                     |
```

```
| |                                                                                  |
| 00:SCAN HBASE [hbase.hbase_table]                                                  |
|    start key: a                                                                    |
|    stop key: b\0                                                                   |
|    hbase filters: cols:never_logged_on EQUAL 'true'                                |
+------------------------------------------------------------------------------------+
```

The query is less efficient if Impala has to evaluate any of the predicates, because Impala must scan the entire HBase table. Impala can only push down predicates to HBase for columns declared as `STRING`. This example tests a column declared as `INT`, and the `predicates:` line in the `EXPLAIN` output indicates that the test is performed after the data is transmitted to Impala.

```
explain select count(*) from hbase_table where year_registered = 2010;
+------------------------------------------------------------------------------------+
| Explain String                                                                     |
+------------------------------------------------------------------------------------+
...

| 01:AGGREGATE                                                                       |
| |  output: count(*)                                                                |
| |                                                                                  |
| 00:SCAN HBASE [hbase.hbase_table]                                                  |
|    predicates: year_registered = 2010                                              |
+------------------------------------------------------------------------------------+
```

The same inefficiency applies if the key column is compared to any non-constant value. Here, even though the key column is a `STRING`, and is tested using an equality operator, Impala must scan the entire HBase table because the key column is compared to another column value rather than a constant.

```
explain select count(*) from hbase_table where cust_id = private_email_address;
+------------------------------------------------------------------------------------+
| Explain String                                                                     |
+------------------------------------------------------------------------------------+
...

| 01:AGGREGATE                                                                       |
| |  output: count(*)                                                                |
| |                                                                                  |
| 00:SCAN HBASE [hbase.hbase_table]                                                  |
|    predicates: cust_id = private_email_address                                     |
+------------------------------------------------------------------------------------+
```

Currently, tests on the row key using `OR` or `IN` clauses are not optimized into direct lookups either. Such limitations might be lifted in the future, so always check the `EXPLAIN` output to be sure whether a particular SQL construct results in an efficient query or not for HBase tables.

```
explain select count(*) from hbase_table where
  cust_id = 'some_user@example.com' or cust_id = 'other_user@example.com';
+------------------------------------------------------------------------------------+
| Explain String
   |
+------------------------------------------------------------------------------------+
...

| 01:AGGREGATE
   |
| |  output: count(*)
   |
| |
   |
| 00:SCAN HBASE [hbase.hbase_table]
   |
|    predicates: cust_id = 'some_user@example.com' OR cust_id = 'other_user@example.com'
 |
+------------------------------------------------------------------------------------+

explain select count(*) from hbase_table where
  cust_id in ('some_user@example.com', 'other_user@example.com');
```

```
+-----------------------------------------------------------------------------------+
| Explain String                                                                    |
+-----------------------------------------------------------------------------------+
...

| 01:AGGREGATE                                                                      |
| |  output: count(*)                                                               |
| |                                                                                 |
| 00:SCAN HBASE [hbase.hbase_table]                                                 |
|    predicates: cust_id IN ('some_user@example.com', 'other_user@example.com')     |
+-----------------------------------------------------------------------------------+
```

Either rewrite into separate queries for each value and combine the results in the application, or combine the single-row queries using UNION ALL:

```
select count(*) from hbase_table where cust_id = 'some_user@example.com';
select count(*) from hbase_table where cust_id = 'other_user@example.com';

explain
  select count(*) from hbase_table where cust_id = 'some_user@example.com'
  union all
  select count(*) from hbase_table where cust_id = 'other_user@example.com';
+-----------------------------------------------------------------------------------+
| Explain String                                                                    |
+-----------------------------------------------------------------------------------+
...

| |  04:AGGREGATE                                                                   |
| |  |  output: count(*)                                                            |
| |  |                                                                              |
| |  03:SCAN HBASE [hbase.hbase_table]                                              |
| |     start key: other_user@example.com                                           |
| |     stop key: other_user@example.com\0                                          |
| |                                                                                 |
| 10:MERGE                                                                          |
...

| 02:AGGREGATE                                                                      |
| |  output: count(*)                                                               |
| |                                                                                 |
| 01:SCAN HBASE [hbase.hbase_table]                                                 |
|    start key: some_user@example.com                                               |
|    stop key: some_user@example.com\0                                              |
+-----------------------------------------------------------------------------------+
```

Configuration Options for Java HBase Applications

If you have an HBase Java application that calls the `setCacheBlocks` or `setCaching` methods of the class org.apache.hadoop.hbase.client.Scan, you can set these same caching behaviors through Impala query options, to control the memory pressure on the HBase RegionServer. For example, when doing queries in HBase that result in full-table scans (which by default are inefficient for HBase), you can reduce memory usage and speed up the queries by turning off the `HBASE_CACHE_BLOCKS` setting and specifying a large number for the `HBASE_CACHING` setting.

To set these options, issue commands like the following in `impala-shell`:

```
-- Same as calling setCacheBlocks(true) or setCacheBlocks(false).
set hbase_cache_blocks=true;
set hbase_cache_blocks=false;

-- Same as calling setCaching(rows).
set hbase_caching=1000;
```

Or update the `impalad` defaults file `/etc/default/impala` and include settings for `HBASE_CACHE_BLOCKS` and/or `HBASE_CACHING` in the `-default_query_options` setting for `IMPALA_SERVER_ARGS`. See Modifying Impala Startup Options for details.



Note: In Impala 2.0 and later, these options are settable through the JDBC or ODBC interfaces using the SET statement.

Use Cases for Querying HBase through Impala

The following are representative use cases for using Impala to query HBase tables:

- Using HBase to store rapidly incrementing counters, such as how many times a web page has been viewed, or on a social network, how many connections a user has or how many votes a post received. HBase is efficient for capturing such changeable data: the append-only storage mechanism is efficient for writing each change to disk, and a query always returns the latest value. An application could query specific totals like these from HBase, and combine the results with a broader set of data queried from Impala.
- Storing very wide tables in HBase. Wide tables have many columns, possibly thousands, typically recording many attributes for an important subject such as a user of an online service. These tables are also often sparse, that is, most of the columns values are `NULL`, 0, `false`, empty string, or other blank or placeholder value. (For example, any particular web site user might have never used some site feature, filled in a certain field in their profile, visited a particular part of the site, and so on.) A typical query against this kind of table is to look up a single row to retrieve all the information about a specific subject, rather than summing, averaging, or filtering millions of rows as in typical Impala-managed tables.

Loading Data into an HBase Table

The Impala `INSERT` statement works for HBase tables. The `INSERT ... VALUES` syntax is ideally suited to HBase tables, because inserting a single row is an efficient operation for an HBase table. (For regular Impala tables, with data files in HDFS, the tiny data files produced by `INSERT ... VALUES` are extremely inefficient, so you would not use that technique with tables containing any significant data volume.)

When you use the `INSERT ... SELECT` syntax, the result in the HBase table could be fewer rows than you expect. HBase only stores the most recent version of each unique row key, so if an `INSERT ... SELECT` statement copies over multiple rows containing the same value for the key column, subsequent queries will only return one row with each key column value:

Although Impala does not have an `UPDATE` statement, you can achieve the same effect by doing successive `INSERT` statements using the same value for the key column each time:

Limitations and Restrictions of the Impala and HBase Integration

The Impala integration with HBase has the following limitations and restrictions, some inherited from the integration between HBase and Hive, and some unique to Impala:

- If you issue a `DROP TABLE` for an internal (Impala-managed) table that is mapped to an HBase table, the underlying table is not removed in HBase. The Hive `DROP TABLE` statement also removes the HBase table in this case.
- The `INSERT OVERWRITE` statement is not available for HBase tables. You can insert new data, or modify an existing row by inserting a new row with the same key value, but not replace the entire contents of the table. You can do an `INSERT OVERWRITE` in Hive if you need this capability.
- If you issue a `CREATE TABLE LIKE` statement for a table mapped to an HBase table, the new table is also an HBase table, but inherits the same underlying HBase table name as the original. The new table is effectively an alias for the old one, not a new table with identical column structure. Avoid using `CREATE TABLE LIKE` for HBase tables, to avoid any confusion.
- Copying data into an HBase table using the Impala `INSERT ... SELECT` syntax might produce fewer new rows than are in the query result set. If the result set contains multiple rows with the same value for the key column,

each row supercedes any previous rows with the same key value. Because the order of the inserted rows is unpredictable, you cannot rely on this technique to preserve the "latest" version of a particular key value.

- Because the complex data types (ARRAY, STRUCT, and MAP) available in CDH 5.5 / Impala 2.3 and higher are currently only supported in Parquet tables, you cannot use these types in HBase tables that are queried through Impala.
- The LOAD DATA statement cannot be used with HBase tables.
- The TABLESAMPLE clause of the SELECT statement does not apply to a table reference derived from a view, a subquery, or anything other than a real base table. This clause only works for tables backed by HDFS or HDFS-like data files, therefore it does not apply to Kudu or HBase tables.

Examples of Querying HBase Tables from Impala

The following examples create an HBase table with four column families, create a corresponding table through Hive, then insert and query the table through Impala.

In HBase shell, the table name is quoted in CREATE and DROP statements. Tables created in HBase begin in "enabled" state; before dropping them through the HBase shell, you must issue a `disable 'table_name'` statement.

```
$ hbase shell
15/02/10 16:07:45
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.94.2-cdh4.2.0, rUnknown, Fri Feb 15 11:51:18 PST 2013

hbase(main):001:0> create 'hbasealltypessmall', 'boolsCF', 'intsCF', 'floatsCF',
'stringsCF'
0 row(s) in 4.6520 seconds

=> Hbase::Table - hbasealltypessmall
hbase(main):006:0> quit
```

Issue the following CREATE TABLE statement in the Hive shell. (The Impala CREATE TABLE statement currently does not support the STORED BY clause, so you switch into Hive to create the table, then back to Impala and the `impala-shell` interpreter to issue the queries.)

This example creates an external table mapped to the HBase table, usable by both Impala and Hive. It is defined as an external table so that when dropped by Impala or Hive, the original HBase table is not touched at all.

The WITH SERDEPROPERTIES clause specifies that the first column (ID) represents the row key, and maps the remaining columns of the SQL table to HBase column families. The mapping relies on the ordinal order of the columns in the table, not the column names in the CREATE TABLE statement. The first column is defined to be the lookup key; the STRING data type produces the fastest key-based lookups for HBase tables.



Note: For Impala with HBase tables, the most important aspect to ensure good performance is to use a STRING column as the row key, as shown in this example.

```
$ hive
...
hive> use hbase;
OK
Time taken: 4.095 seconds
hive> CREATE EXTERNAL TABLE hbasestringids (
    >   id string,
    >   bool_col boolean,
    >   tinyint_col tinyint,
    >   smallint_col smallint,
    >   int_col int,
    >   bigint_col bigint,
    >   float_col float,
```

```
    >   double_col double,
    >   date_string_col string,
    >   string_col string,
    >   timestamp_col timestamp)
    > STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
    > WITH SERDEPROPERTIES (
    >   "hbase.columns.mapping" =
    >
":key,boolsCF:bool_col,intsCF:tinyint_col,intsCF:smallint_col,intsCF:int_col,intsCF:\
    >   bigint_col,floatsCF:float_col,floatsCF:double_col,stringsCF:date_string_col,\
    >   stringsCF:string_col,stringsCF:timestamp_col"
    > )
    > TBLPROPERTIES("hbase.table.name" = "hbasealltypessmall");
OK
Time taken: 2.879 seconds
hive> quit;
```

Once you have established the mapping to an HBase table, you can issue DML statements and queries from Impala. The following example shows a series of `INSERT` statements followed by a query. The ideal kind of query from a performance standpoint retrieves a row from the table based on a row key mapped to a string column. An initial `INVALIDATE METADATA` *`table_name`* statement makes the table created through Hive visible to Impala.

```
$ impala-shell -i localhost -d hbase
Starting Impala Shell without Kerberos authentication
Connected to localhost:21000
...
Query: use `hbase`
[localhost:21000] > invalidate metadata hbasestringids;
Fetched 0 row(s) in 0.09s
[localhost:21000] > desc hbasestringids;
+-----------------+-----------+---------+
| name            | type      | comment |
+-----------------+-----------+---------+
| id              | string    |         |
| bool_col        | boolean   |         |
| double_col      | double    |         |
| float_col       | float     |         |
| bigint_col      | bigint    |         |
| int_col         | int       |         |
| smallint_col    | smallint  |         |
| tinyint_col     | tinyint   |         |
| date_string_col | string    |         |
| string_col      | string    |         |
| timestamp_col   | timestamp |         |
+-----------------+-----------+---------+
Fetched 11 row(s) in 0.02s
[localhost:21000] > insert into hbasestringids values
('0001',true,3.141,9.94,1234567,32768,4000,76,'2014-12-31','Hello world',now());
Inserted 1 row(s) in 0.26s
[localhost:21000] > insert into hbasestringids values
('0002',false,2.004,6.196,1500,8000,129,127,'2014-01-01','Foo bar',now());
Inserted 1 row(s) in 0.12s
[localhost:21000] > select * from hbasestringids where id = '0001';
+---+------+---------+-----------+---------+-----+---------+---------+-----------+---------+-------------------+
| id   | bool_col | double_col | float_col        | bigint_col | int_col | smallint_col
 | tinyint_col | date_string_col | string_col  | timestamp_col                |
+---+------+---------+-----------+---------+-----+---------+---------+-----------+---------+-------------------+
| 0001 | true     | 3.141      | 9.939999580383301 | 1234567    | 32768   | 4000
   | 76          | 2014-12-31      | Hello world | 2015-02-10 16:36:59.764838000 |
+---+------+---------+-----------+---------+-----+---------+---------+-----------+---------+-------------------+
Fetched 1 row(s) in 0.54s
```



Note: After you create a table in Hive, such as the HBase mapping table in this example, issue an `INVALIDATE METADATA` *`table_name`* statement the next time you connect to Impala, make Impala aware of the new table. (Prior to Impala 1.2.4, you could not specify the table name if Impala was not aware of the table yet; in Impala 1.2.4 and higher, specifying the table name avoids reloading the metadata for other tables that are not changed.)

Using Impala with the Amazon S3 Filesystem



Important:

In CDH 5.8 / Impala 2.6 and higher, Impala supports both queries (SELECT) and DML (INSERT, LOAD DATA, CREATE TABLE AS SELECT) for data residing on Amazon S3. With the inclusion of write support, the Impala support for S3 is now considered ready for production use.

You can use Impala to query data residing on the Amazon S3 filesystem. This capability allows convenient access to a storage system that is remotely managed, accessible from anywhere, and integrated with various cloud-based services. Impala can query files in any supported file format from S3. The S3 storage location can be for an entire table, or individual partitions in a partitioned table.

The default Impala tables use data files stored on HDFS, which are ideal for bulk loads and queries using full-table scans. In contrast, queries against S3 data are less performant, making S3 suitable for holding "cold" data that is only queried occasionally, while more frequently accessed "hot" data resides in HDFS. In a partitioned table, you can set the LOCATION attribute for individual partitions to put some partitions on HDFS and others on S3, typically depending on the age of the data.

See Specifying Impala Credentials to Access Data in S3 for information about configuring Impala to use Amazon S3 filesystem.

How Impala SQL Statements Work with S3

Impala SQL statements work with data on S3 as follows:

- The CREATE TABLE Statement on page 234 or ALTER TABLE Statement on page 205 statements can specify that a table resides on the S3 filesystem by encoding an s3a:// prefix for the LOCATION property. ALTER TABLE can also set the LOCATION property for an individual partition, so that some data in a table resides on S3 and other data in the same table resides on HDFS.
- Once a table or partition is designated as residing on S3, the SELECT Statement on page 295 statement transparently accesses the data files from the appropriate storage layer.
- If the S3 table is an internal table, the DROP TABLE Statement on page 268 statement removes the corresponding data files from S3 when the table is dropped.
- The TRUNCATE TABLE Statement (CDH 5.5 or higher only) on page 381 statement always removes the corresponding data files from S3 when the table is truncated.
- The LOAD DATA Statement on page 288 can move data files residing in HDFS into an S3 table.
- The INSERT Statement on page 277 statement, or the CREATE TABLE AS SELECT form of the CREATE TABLE statement, can copy data from an HDFS table or another S3 table into an S3 table. The S3_SKIP_INSERT_STAGING Query Option (CDH 5.8 or higher only) on page 358 query option chooses whether or not to use a fast code path for these write operations to S3, with the tradeoff of potential inconsistency in the case of a failure during the statement.

For usage information about Impala SQL statements with S3 tables, see Creating Impala Databases, Tables, and Partitions for Data Stored on S3 on page 694 and Using Impala DML Statements for S3 Data on page 693.

Loading Data into S3 for Impala Queries

If your ETL pipeline involves moving data into S3 and then querying through Impala, you can either use Impala DML statements to create, move, or copy the data, or use the same data loading techniques as you would for non-Impala data.

Using Impala DML Statements for S3 Data

In CDH 5.8 / Impala 2.6 and higher, the Impala DML statements (`INSERT`, `LOAD DATA`, and `CREATE TABLE AS SELECT`) can write data into a table or partition that resides in the Amazon Simple Storage Service (S3). The syntax of the DML statements is the same as for any other tables, because the S3 location for tables and partitions is specified by an `s3a://` prefix in the `LOCATION` attribute of `CREATE TABLE` or `ALTER TABLE` statements. If you bring data into S3 using the normal S3 transfer mechanisms instead of Impala DML statements, issue a `REFRESH` statement for the table before using Impala to query the S3 data.

Because of differences between S3 and traditional filesystems, DML operations for S3 tables can take longer than for tables on HDFS. For example, both the `LOAD DATA` statement and the final stage of the `INSERT` and `CREATE TABLE AS SELECT` statements involve moving files from one directory to another. (In the case of `INSERT` and `CREATE TABLE AS SELECT`, the files are moved from a temporary staging directory to the final destination directory.) Because S3 does not support a "rename" operation for existing objects, in these cases Impala actually copies the data files from one location to another and then removes the original files. In CDH 5.8 / Impala 2.6, the `S3_SKIP_INSERT_STAGING` query option provides a way to speed up `INSERT` statements for S3 tables and partitions, with the tradeoff that a problem during statement execution could leave data in an inconsistent state. It does not apply to `INSERT OVERWRITE` or `LOAD DATA` statements. See S3_SKIP_INSERT_STAGING Query Option (CDH 5.8 or higher only) on page 358 for details.

Manually Loading Data into Impala Tables on S3

As an alternative, or on earlier Impala releases without DML support for S3, you can use the Amazon-provided methods to bring data files into S3 for querying through Impala. See the Amazon S3 web site for details.



Important:

For best compatibility with the S3 write support in CDH 5.8 / Impala 2.6 and higher:

- Use native Hadoop techniques to create data files in S3 for querying through Impala.
- Use the `PURGE` clause of `DROP TABLE` when dropping internal (managed) tables.

By default, when you drop an internal (managed) table, the data files are moved to the HDFS trashcan. This operation is expensive for tables that reside on the Amazon S3 filesystem. Therefore, for S3 tables, prefer to use `DROP TABLE table_name PURGE` rather than the default `DROP TABLE` statement. The `PURGE` clause makes Impala delete the data files immediately, skipping the HDFS trashcan. For the `PURGE` clause to work effectively, you must originally create the data files on S3 using one of the tools from the Hadoop ecosystem, such as `hadoop fs -cp`, or `INSERT` in Impala or Hive.

Alternative file creation techniques (less compatible with the `PURGE` clause) include:

- The Amazon AWS / S3 web interface to upload from a web browser.
- The Amazon AWS CLI to manipulate files from the command line.
- Other S3-enabled software, such as the S3Tools client software.

After you upload data files to a location already mapped to an Impala table or partition, or if you delete files in S3 from such a location, issue the `REFRESH table_name` statement to make Impala aware of the new set of data files.

Creating Impala Databases, Tables, and Partitions for Data Stored on S3

Impala reads data for a table or partition from S3 based on the `LOCATION` attribute for the table or partition. Specify the S3 details in the `LOCATION` clause of a `CREATE TABLE` or `ALTER TABLE` statement. The notation for the `LOCATION` clause is `s3a://bucket_name/path/to/file`. The filesystem prefix is always `s3a://` because Impala does not support the `s3://` or `s3n://` prefixes.

For a partitioned table, either specify a separate `LOCATION` clause for each new partition, or specify a base `LOCATION` for the table and set up a directory structure in S3 to mirror the way Impala partitioned tables are structured in HDFS. Although, strictly speaking, S3 filenames do not have directory paths, Impala treats S3 filenames with `/` characters the same as HDFS pathnames that include directories.

You point a nonpartitioned table or an individual partition at S3 by specifying a single directory path in S3, which could be any arbitrary directory. To replicate the structure of an entire Impala partitioned table or database in S3 requires more care, with directories and subdirectories nested and named to match the equivalent directory tree in HDFS. Consider setting up an empty staging area if necessary in HDFS, and recording the complete directory structure so that you can replicate it in S3.

For convenience when working with multiple tables with data files stored in S3, you can create a database with a `LOCATION` attribute pointing to an S3 path. Specify a URL of the form `s3a://bucket/root/path/for/database` for the `LOCATION` attribute of the database. Any tables created inside that database automatically create directories underneath the one specified by the database `LOCATION` attribute.

For example, the following session creates a partitioned table where only a single partition resides on S3. The partitions for years 2013 and 2014 are located on HDFS. The partition for year 2015 includes a `LOCATION` attribute with an `s3a://` URL, and so refers to data residing on S3, under a specific path underneath the bucket `impala-demo`.

```
[localhost:21000] > create database db_on_hdfs;
[localhost:21000] > use db_on_hdfs;
[localhost:21000] > create table mostly_on_hdfs (x int) partitioned by (year int);
[localhost:21000] > alter table mostly_on_hdfs add partition (year=2013);
[localhost:21000] > alter table mostly_on_hdfs add partition (year=2014);
[localhost:21000] > alter table mostly_on_hdfs add partition (year=2015)
                  >   location 's3a://impala-demo/dir1/dir2/dir3/t1';
```

The following session creates a database and two partitioned tables residing entirely on S3, one partitioned by a single column and the other partitioned by multiple columns. Because a `LOCATION` attribute with an `s3a://` URL is specified for the database, the tables inside that database are automatically created on S3 underneath the database directory. To see the names of the associated subdirectories, including the partition key values, we use an S3 client tool to examine how the directory structure is organized on S3. For example, Impala partition directories such as `month=1` do not include leading zeroes, which sometimes appear in partition directories created through Hive.

```
[localhost:21000] > create database db_on_s3 location 's3a://impala-demo/dir1/dir2/dir3';
[localhost:21000] > use db_on_s3;

[localhost:21000] > create table partitioned_on_s3 (x int) partitioned by (year int);
[localhost:21000] > alter table partitioned_on_s3 add partition (year=2013);
[localhost:21000] > alter table partitioned_on_s3 add partition (year=2014);
[localhost:21000] > alter table partitioned_on_s3 add partition (year=2015);

[localhost:21000] > !aws s3 ls s3://impala-demo/dir1/dir2/dir3 --recursive;
2015-03-17 13:56:34          0 dir1/dir2/dir3/
2015-03-17 16:43:28          0 dir1/dir2/dir3/partitioned_on_s3/
2015-03-17 16:43:49          0 dir1/dir2/dir3/partitioned_on_s3/year=2013/
2015-03-17 16:43:53          0 dir1/dir2/dir3/partitioned_on_s3/year=2014/
2015-03-17 16:43:58          0 dir1/dir2/dir3/partitioned_on_s3/year=2015/

[localhost:21000] > create table partitioned_multiple_keys (x int)
                  >   partitioned by (year smallint, month tinyint, day tinyint);
[localhost:21000] > alter table partitioned_multiple_keys
                  >   add partition (year=2015,month=1,day=1);
[localhost:21000] > alter table partitioned_multiple_keys
                  >   add partition (year=2015,month=1,day=31);
[localhost:21000] > alter table partitioned_multiple_keys
```

```
                  >   add partition (year=2015,month=2,day=28);

[localhost:21000] > !aws s3 ls s3://impala-demo/dir1/dir2/dir3 --recursive;
2015-03-17 13:56:34          0 dir1/dir2/dir3/
2015-03-17 16:47:13          0 dir1/dir2/dir3/partitioned_multiple_keys/
2015-03-17 16:47:44          0
dir1/dir2/dir3/partitioned_multiple_keys/year=2015/month=1/day=1/
2015-03-17 16:47:50          0
dir1/dir2/dir3/partitioned_multiple_keys/year=2015/month=1/day=31/
2015-03-17 16:47:57          0
dir1/dir2/dir3/partitioned_multiple_keys/year=2015/month=2/day=28/
2015-03-17 16:43:28          0 dir1/dir2/dir3/partitioned_on_s3/
2015-03-17 16:43:49          0 dir1/dir2/dir3/partitioned_on_s3/year=2013/
2015-03-17 16:43:53          0 dir1/dir2/dir3/partitioned_on_s3/year=2014/
2015-03-17 16:43:58          0 dir1/dir2/dir3/partitioned_on_s3/year=2015/
```

The CREATE DATABASE and CREATE TABLE statements create the associated directory paths if they do not already exist. You can specify multiple levels of directories, and the CREATE statement creates all appropriate levels, similar to using `mkdir -p`.

Use the standard S3 file upload methods to actually put the data files into the right locations. You can also put the directory paths and data files in place before creating the associated Impala databases or tables, and Impala automatically uses the data from the appropriate location after the associated databases and tables are created.

You can switch whether an existing table or partition points to data in HDFS or S3. For example, if you have an Impala table or partition pointing to data files in HDFS or S3, and you later transfer those data files to the other filesystem, use an ALTER TABLE statement to adjust the LOCATION attribute of the corresponding table or partition to reflect that change. Because Impala does not have an ALTER DATABASE statement, this location-switching technique is not practical for entire databases that have a custom LOCATION attribute.

Internal and External Tables Located on S3

Just as with tables located on HDFS storage, you can designate S3-based tables as either internal (managed by Impala) or external, by using the syntax CREATE TABLE or CREATE EXTERNAL TABLE respectively. When you drop an internal table, the files associated with the table are removed, even if they are on S3 storage. When you drop an external table, the files associated with the table are left alone, and are still available for access by other tools or components. See Overview of Impala Tables on page 196 for details.

If the data on S3 is intended to be long-lived and accessed by other tools in addition to Impala, create any associated S3 tables with the CREATE EXTERNAL TABLE syntax, so that the files are not deleted from S3 when the table is dropped.

If the data on S3 is only needed for querying by Impala and can be safely discarded once the Impala workflow is complete, create the associated S3 tables using the CREATE TABLE syntax, so that dropping the table also deletes the corresponding data files on S3.

For example, this session creates a table in S3 with the same column layout as a table in HDFS, then examines the S3 table and queries some data from it. The table in S3 works the same as a table in HDFS as far as the expected file format of the data, table and column statistics, and other table properties. The only indication that it is not an HDFS table is the `s3a://` URL in the LOCATION property. Many data files can reside in the S3 directory, and their combined contents form the table data. Because the data in this example is uploaded after the table is created, a REFRESH statement prompts Impala to update its cached information about the data files.

```
[localhost:21000] > create table usa_cities_s3 like usa_cities location
's3a://impala-demo/usa_cities';
[localhost:21000] > desc usa_cities_s3;
+-------+----------+---------+
| name  | type     | comment |
+-------+----------+---------+
| id    | smallint |         |
| city  | string   |         |
| state | string   |         |
+-------+----------+---------+
```

```
-- Now from a web browser, upload the same data file(s) to S3 as in the HDFS table,
-- under the relevant bucket and path. If you already have the data in S3, you would
-- point the table LOCATION at an existing path.

[localhost:21000] > refresh usa_cities_s3;
[localhost:21000] > select count(*) from usa_cities_s3;
+----------+
| count(*) |
+----------+
| 289      |
+----------+
[localhost:21000] > select distinct state from sample_data_s3 limit 5;
+----------------------+
| state                |
+----------------------+
| Louisiana            |
| Minnesota            |
| Georgia              |
| Alaska               |
| Ohio                 |
+----------------------+
[localhost:21000] > desc formatted usa_cities_s3;
+------------------------------+------------------------------+---------+
| name                         | type                         | comment |
+------------------------------+------------------------------+---------+
| # col_name                   | data_type                    | comment |
|                              | NULL                         | NULL    |
| id                           | smallint                     | NULL    |
| city                         | string                       | NULL    |
| state                        | string                       | NULL    |
|                              | NULL                         | NULL    |
| # Detailed Table Information | NULL                         | NULL    |
| Database:                    | s3_testing                   | NULL    |
| Owner:                       | jrussell                     | NULL    |
| CreateTime:                  | Mon Mar 16 11:36:25 PDT 2015 | NULL    |
| LastAccessTime:              | UNKNOWN                      | NULL    |
| Protect Mode:                | None                         | NULL    |
| Retention:                   | 0                            | NULL    |
| Location:                    | s3a://impala-demo/usa_cities | NULL    |
| Table Type:                  | MANAGED_TABLE                | NULL    |
...
+------------------------------+------------------------------+---------+
```

In this case, we have already uploaded a Parquet file with a million rows of data to the `sample_data` directory underneath the `impala-demo` bucket on S3. This session creates a table with matching column settings pointing to the corresponding location in S3, then queries the table. Because the data is already in place on S3 when the table is created, no REFRESH statement is required.

```
[localhost:21000] > create table sample_data_s3
                  > (id int, id bigint, val int, zerofill string,
                  > name string, assertion boolean, city string, state string)
                  > stored as parquet location 's3a://impala-demo/sample_data';
[localhost:21000] > select count(*) from sample_data_s3;;
+----------+
| count(*) |
+----------+
| 1000000  |
+----------+
[localhost:21000] > select count(*) howmany, assertion from sample_data_s3 group by
assertion;
+---------+-----------+
| howmany | assertion |
+---------+-----------+
| 667149  | true      |
| 332851  | false     |
+---------+-----------+
```

Running and Tuning Impala Queries for Data Stored on S3

Once the appropriate `LOCATION` attributes are set up at the table or partition level, you query data stored in S3 exactly the same as data stored on HDFS or in HBase:

- Queries against S3 data support all the same file formats as for HDFS data.
- Tables can be unpartitioned or partitioned. For partitioned tables, either manually construct paths in S3 corresponding to the HDFS directories representing partition key values, or use `ALTER TABLE ... ADD PARTITION` to set up the appropriate paths in S3.
- HDFS and HBase tables can be joined to S3 tables, or S3 tables can be joined with each other.
- Authorization using the Sentry framework to control access to databases, tables, or columns works the same whether the data is in HDFS or in S3.
- The `catalogd` daemon caches metadata for both HDFS and S3 tables. Use `REFRESH` and `INVALIDATE METADATA` for S3 tables in the same situations where you would issue those statements for HDFS tables.
- Queries against S3 tables are subject to the same kinds of admission control and resource management as HDFS tables.
- Metadata about S3 tables is stored in the same metastore database as for HDFS tables.
- You can set up views referring to S3 tables, the same as for HDFS tables.
- The `COMPUTE STATS`, `SHOW TABLE STATS`, and `SHOW COLUMN STATS` statements work for S3 tables also.

Understanding and Tuning Impala Query Performance for S3 Data

Although Impala queries for data stored in S3 might be less performant than queries against the equivalent data stored in HDFS, you can still do some tuning. Here are techniques you can use to interpret explain plans and profiles for queries against S3 data, and tips to achieve the best performance possible for such queries.

All else being equal, performance is expected to be lower for queries running against data on S3 rather than HDFS. The actual mechanics of the `SELECT` statement are somewhat different when the data is in S3. Although the work is still distributed across the datanodes of the cluster, Impala might parallelize the work for a distributed query differently for data on HDFS and S3. S3 does not have the same block notion as HDFS, so Impala uses heuristics to determine how to split up large S3 files for processing in parallel. Because all hosts can access any S3 data file with equal efficiency, the distribution of work might be different than for HDFS data, where the data blocks are physically read using short-circuit local reads by hosts that contain the appropriate block replicas. Although the I/O to read the S3 data might be spread evenly across the hosts of the cluster, the fact that all data is initially retrieved across the network means that the overall query performance is likely to be lower for S3 data than for HDFS data.

In CDH 5.8 / Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting `fs.s3a.block.size` in the `core-site.xml` configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase `fs.s3a.block.size` to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase `fs.s3a.block.size` to 268435456 (256 MB) to match the row group size produced by Impala.

Because of differences between S3 and traditional filesystems, DML operations for S3 tables can take longer than for tables on HDFS. For example, both the `LOAD DATA` statement and the final stage of the `INSERT` and `CREATE TABLE AS SELECT` statements involve moving files from one directory to another. (In the case of `INSERT` and `CREATE TABLE AS SELECT`, the files are moved from a temporary staging directory to the final destination directory.) Because S3 does not support a "rename" operation for existing objects, in these cases Impala actually copies the data files from one location to another and then removes the original files. In CDH 5.8 / Impala 2.6, the `S3_SKIP_INSERT_STAGING` query option provides a way to speed up `INSERT` statements for S3 tables and partitions, with the tradeoff that a problem during statement execution could leave data in an inconsistent state. It does not apply to `INSERT OVERWRITE` or `LOAD DATA` statements. See S3_SKIP_INSERT_STAGING Query Option (CDH 5.8 or higher only) on page 358 for details.

When optimizing aspects of for complex queries such as the join order, Impala treats tables on HDFS and S3 the same way. Therefore, follow all the same tuning recommendations for S3 tables as for HDFS ones, such as using the COMPUTE STATS statement to help Impala construct accurate estimates of row counts and cardinality. See Tuning Impala for Performance on page 565 for details.

In query profile reports, the numbers for `BytesReadLocal`, `BytesReadShortCircuit`, `BytesReadDataNodeCached`, and `BytesReadRemoteUnexpected` are blank because those metrics come from HDFS. If you do see any indications that a query against an S3 table performed "remote read" operations, do not be alarmed. That is expected because, by definition, all the I/O for S3 tables involves remote reads.

Restrictions on Impala Support for S3

Impala requires that the default filesystem for the cluster be HDFS. You cannot use S3 as the only filesystem in the cluster.

Prior to CDH 5.8 / Impala 2.6 Impala could not perform DML operations (`INSERT`, `LOAD DATA`, or `CREATE TABLE AS SELECT`) where the destination is a table or partition located on an S3 filesystem. This restriction is lifted in CDH 5.8 / Impala 2.6 and higher.

Impala does not support the old `s3://` block-based and `s3n://` filesystem schemes, only `s3a://`.

Although S3 is often used to store JSON-formatted data, the current Impala support for S3 does not include directly querying JSON data. For Impala queries, use data files in one of the file formats listed in How Impala Works with Hadoop File Formats on page 634. If you have data in JSON format, you can prepare a flattened version of that data for querying by Impala as part of your ETL cycle.

You cannot use the `ALTER TABLE ... SET CACHED` statement for tables or partitions that are located in S3.

Best Practices for Using Impala with S3

The following guidelines represent best practices derived from testing and field experience with Impala on S3:

- Any reference to an S3 location must be fully qualified. (This rule applies when S3 is not designated as the default filesystem.)
- Set the safety valve `fs.s3a.connection.maximum` to 1500 for `impalad`.
- Set safety valve `fs.s3a.block.size` to 134217728 (128 MB in bytes) if most Parquet files queried by Impala were written by Hive or ParquetMR jobs. Set the block size to 268435456 (256 MB in bytes) if most Parquet files queried by Impala were written by Impala.
- `DROP TABLE .. PURGE` is much faster than the default `DROP TABLE`. The same applies to `ALTER TABLE ... DROP PARTITION PURGE` versus the default `DROP PARTITION` operation. However, due to the eventually consistent nature of S3, the files for that table or partition could remain for some unbounded time when using `PURGE`. The default `DROP TABLE/PARTITION` is slow because Impala copies the files to the HDFS trash folder, and Impala waits until all the data is moved. `DROP TABLE/PARTITION .. PURGE` is a fast delete operation, and the Impala statement finishes quickly even though the change might not have propagated fully throughout S3.
- `INSERT` statements are faster than `INSERT OVERWRITE` for S3. The query option `S3_SKIP_INSERT_STAGING`, which is set to `true` by default, skips the staging step for regular `INSERT` (but not `INSERT OVERWRITE`). This makes the operation much faster, but consistency is not guaranteed: if a node fails during execution, the table could end up with inconsistent data. Set this option to `false` if stronger consistency is required, however this setting will make the `INSERT` operations slower.
- Too many files in a table can make metadata loading and updating slow on S3. If too many requests are made to S3, S3 has a back-off mechanism and responds slower than usual. You might have many small files because of:
  - Too many partitions due to over-granular partitioning. Prefer partitions with many megabytes of data, so that even a query against a single partition can be parallelized effectively.

– Many small `INSERT` queries. Prefer bulk `INSERT`s so that more data is written to fewer files.

Specifying Impala Credentials to Access Data in S3 with Cloudera Manager

Cloudera recommends that you use Cloudera Manager to specify Impala credentials to access data in Amazon S3. If you are not using Cloudera Manager, see Specifying Impala Credentials to Access Data in S3 from the command line.

To configure access to data stored in S3 for Impala with Cloudera Manager, use one of the following authentication types:

- **IAM Role-based Authentication**

  Amazon Identity & Access Management (IAM). You must set up IAM role-based authentication in Amazon. See Amazon documentation. This authentication method is best suited for environments where there is a single user, or where all cluster users can have the same privileges to data in S3. See How to Configure AWS Credentials for information about using IAM role-based authentication with Cloudera Manager.

- **Access Key Authentication**

  For environments where you have multiple users or multi-tenancy, use an AWS access key and an AWS secret key that you obtain from Amazon. See Amazon documentation. For this scenario, you must enable the Sentry service and Kerberos to use the **S3 Connector** service. Cloudera Manager stores your AWS credentials securely and does not store them in world-readable locations. If you can use the Sentry service and Kerberos, see the following sections to add your AWS credentials to Cloudera Manager and to manage them:

  – Adding AWS Credentials
  – Managing AWS Credentials



Note: If you cannot use the Sentry service or Kerberos in your environment, see the next section, Specifying Impala Credentials on Clusters Not Secured by Sentry or Kerberos on page 699.

Specifying Impala Credentials on Clusters Not Secured by Sentry or Kerberos

If you cannot use the Sentry service or Kerberos in your environment, specify Impala credentials in the **Cluster-wide Advanced Configuration Snippet (Safety Valve) for core-site.xml**. For example:

```
<property>
   <name>fs.s3a.access.key</name>
   <value>your_access_key</value>
</property>
<property>
   <name>fs.s3a.secret.key</name>
   <value>your_secret_key</value>
</property>
```

Specifying your credentials in this safety valve does not require Kerberos or the Sentry service, but it is not as secure. After specifying the credentials, restart both the Impala and Hive services. Restarting Hive is required because operations such as Impala queries and `CREATE TABLE` statements go through the Hive metastore.

Specifying Impala Credentials to Access Data in S3

To allow Impala to access data in S3, specify values for the following configuration settings in your `core-site.xml` file:

```
<property>
```

```
   <name>fs.s3a.access.key</name>
   <value>your_access_key</value>
</property>
<property>
   <name>fs.s3a.secret.key</name>
   <value>your_secret_key</value>
</property>
```

After specifying the credentials, restart both the Impala and Hive services. Restarting Hive is required because operations such as Impala queries and CREATE TABLE statements go through the Hive metastore.



Important:

Although you can specify the access key ID and secret key as part of the `s3a://` URL in the `LOCATION` attribute, doing so makes this sensitive information visible in many places, such as `DESCRIBE FORMATTED` output and Impala log files. Therefore, specify this information centrally in the `core-site.xml` file, and restrict read access to that file to only trusted users.

Using Impala with the Azure Data Lake Store (ADLS)

You can use Impala to query data residing on the Azure Data Lake Store (ADLS) filesystem. This capability allows convenient access to a storage system that is remotely managed, accessible from anywhere, and integrated with various cloud-based services. Impala can query files in any supported file format from ADLS. The ADLS storage location can be for an entire table or individual partitions in a partitioned table.

The default Impala tables use data files stored on HDFS, which are ideal for bulk loads and queries using full-table scans. In contrast, queries against ADLS data are less performant, making ADLS suitable for holding "cold" data that is only queried occasionally, while more frequently accessed "hot" data resides in HDFS. In a partitioned table, you can set the `LOCATION` attribute for individual partitions to put some partitions on HDFS and others on ADLS, typically depending on the age of the data.

Starting in CDH 6.1, Impala supports ADLS Gen2 filesystem, Azure Blob File System (ABFS).

Prerequisites

These procedures presume that you have already set up an Azure account, configured an ADLS store, and configured your Hadoop cluster with appropriate credentials to be able to access ADLS. See the following resources for information:

- Get started with Azure Data Lake Store using the Azure Portal
- Azure Data Lake Storage Gen2
- Hadoop Azure Data Lake Support

How Impala SQL Statements Work with ADLS

Impala SQL statements work with data on ADLS as follows.

- The CREATE TABLE Statement on page 234 or ALTER TABLE Statement on page 205 statements can specify that a table resides on the ADLS filesystem by using one of the following ADLS prefixes in the `LOCATION` property.
  - For ADLS Gen1: `adl://`
  - For ADLS Gen2: `abfs://` or `abfss://`

  `ALTER TABLE` can also set the `LOCATION` property for an individual partition, so that some data in a table resides on ADLS and other data in the same table resides on HDFS.

  See Creating Impala Databases, Tables, and Partitions for Data Stored on ADLS on page 703 for usage information.
- Once a table or partition is designated as residing on ADLS, the SELECT Statement on page 295 statement transparently accesses the data files from the appropriate storage layer.
- If the ADLS table is an internal table, the DROP TABLE Statement on page 268 statement removes the corresponding data files from ADLS when the table is dropped.
- The TRUNCATE TABLE Statement (CDH 5.5 or higher only) on page 381 statement always removes the corresponding data files from ADLS when the table is truncated.
- The LOAD DATA Statement on page 288 can move data files residing in HDFS into an ADLS table.
- The INSERT Statement on page 277, or the `CREATE TABLE AS SELECT` form of the `CREATE TABLE` statement, can copy data from an HDFS table or another ADLS table into an ADLS table.

For usage information about Impala SQL statements with ADLS tables, see Using Impala DML Statements for ADLS Data on page 703.

Specifying Impala Credentials to Access Data in ADLS

You can configure credentials to access ADLS in Cloudera Manager or in plain text.

When you configure credentials using Cloudera Manager, it provides a more secure way to access ADLS using credentials that are not stored in plain-text files. See Configuring ADLS Access Using Cloudera Manager for the steps to configure ADLS credentials using Cloudera Manager.

Important: Cloudera recommends that you only use the plain text method for accessing ADLS in development environments or other environments where security is not a concern.

To allow Impala to access data in ADLS using credentials in plain text, specify values for the following configuration settings in your `core-site.xml` file:

For ADLS Gen1:

```
<property>
   <name>dfs.adls.oauth2.access.token.provider.type</name>
   <value>ClientCredential</value>
</property>
<property>
   <name>dfs.adls.oauth2.client.id</name>
   <value>your_client_id</value>
</property>
<property>
   <name>dfs.adls.oauth2.credential</name>
   <value>your_client_secret</value>
</property>
<property>
   <name>dfs.adls.oauth2.refresh.url</name>
   <value>https://login.windows.net/your_azure_tenant_id/oauth2/token</value>
</property>
```

For ADLS Gen2:

```
 <property>
    <name>fs.azure.account.auth.type</name>
    <value>OAuth</value>
  </property>

  <property>
    <name>fs.azure.account.oauth.provider.type</name>
    <value>org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider</value>
  </property>

  <property>
    <name>fs.azure.account.oauth2.client.id</name>
    <value>your_client_id</value>
  </property>

  <property>
    <name>fs.azure.account.oauth2.client.secret</name>
    <value>your_client_secret</value>
  </property>

  <property>
    <name>fs.azure.account.oauth2.client.endpoint</name>
    <value>https://login.microsoftonline.com/your_azure_tenant_id/oauth2/token</value>
  </property>
```

Note:

Check if your Hadoop distribution or cluster management tool includes support for filling in and distributing credentials across the cluster in an automated way.

After specifying the credentials, restart both the Impala and Hive services. Restarting Hive is required because Impala DDL statements, such as the CREATE TABLE statements, go through the Hive metastore.

Loading Data into ADLS for Impala Queries

If your ETL pipeline involves moving data into ADLS and then querying through Impala, you can either use Impala DML statements to create, move, or copy the data, or use the same data loading techniques as you would for non-Impala data.

Using Impala DML Statements for ADLS Data

In CDH 5.12 / Impala 2.9 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Azure Data Lake Store (ADLS). ADLS Gen2 is supported in CDH 6.1 and higher.

In theCREATE TABLE or ALTER TABLE statements, specify the ADLS location for tables and partitions with the `adl://` prefix for ADLS Gen1 and `abfs://` or `abfss://` for ADLS Gen2 in the LOCATION attribute.

If you bring data into ADLS using the normal ADLS transfer mechanisms instead of Impala DML statements, issue a REFRESH statement for the table before using Impala to query the ADLS data.

Manually Loading Data into Impala Tables on ADLS

As an alternative, you can use the Microsoft-provided methods to bring data files into ADLS for querying through Impala. See the Microsoft ADLS documentation for details.

After you upload data files to a location already mapped to an Impala table or partition, or if you delete files in ADLS from such a location, issue the REFRESH *table_name* statement to make Impala aware of the new set of data files.

Creating Impala Databases, Tables, and Partitions for Data Stored on ADLS

Impala reads data for a table or partition from ADLS based on the LOCATION attribute for the table or partition. Specify the ADLS details in the LOCATION clause of a CREATE TABLE or ALTER TABLE statement. The syntax for the LOCATION clause is:

- For ADLS Gen1:

```
adl://account.azuredatalakestore.net/path/file
```

- For ADLS Gen2:

```
abfs://container@account.dfs.core.windows.net/path/file
```

or

```
abfss://container@account.dfs.core.windows.net/path/file
```

container denotes the parent location that holds the files and folders, which is the Containers in the Azure Storage Blobs service.

account is the name given for your storage account.



Note:

By default, TLS is enabled both with `abfs://` and `abfss://`.

When you set the `fs.azure.always.use.https=false` property, TLS is disabled with `abfs://`, and TLS is enabled with `abfss://`

For a partitioned table, either specify a separate `LOCATION` clause for each new partition, or specify a base `LOCATION` for the table and set up a directory structure in ADLS to mirror the way Impala partitioned tables are structured in HDFS. Although, strictly speaking, ADLS filenames do not have directory paths, Impala treats ADLS filenames with `/` characters the same as HDFS pathnames that include directories.

To point a nonpartitioned table or an individual partition at ADLS, specify a single directory path in ADLS, which could be any arbitrary directory. To replicate the structure of an entire Impala partitioned table or database in ADLS requires more care, with directories and subdirectories nested and named to match the equivalent directory tree in HDFS. Consider setting up an empty staging area if necessary in HDFS, and recording the complete directory structure so that you can replicate it in ADLS.

For example, the following session creates a partitioned table where only a single partition resides on ADLS. The partitions for years 2013 and 2014 are located on HDFS. The partition for year 2015 includes a `LOCATION` attribute with an `adl://` URL, and so refers to data residing on ADLS, under a specific path underneath the store `impalademo`.

```
[localhost:21000] > create database db_on_hdfs;
[localhost:21000] > use db_on_hdfs;
[localhost:21000] > create table mostly_on_hdfs (x int) partitioned by (year int);
[localhost:21000] > alter table mostly_on_hdfs add partition (year=2013);
[localhost:21000] > alter table mostly_on_hdfs add partition (year=2014);
[localhost:21000] > alter table mostly_on_hdfs add partition (year=2015)
                  >   location
'adl://impalademo.azuredatalakestore.net/dir1/dir2/dir3/t1';
```

For convenience when working with multiple tables with data files stored in ADLS, you can create a database with a `LOCATION` attribute pointing to an ADLS path. Specify a URL of the form as shown above. Any tables created inside that database automatically create directories underneath the one specified by the database `LOCATION` attribute.

The following session creates a database and two partitioned tables residing entirely on ADLS, one partitioned by a single column and the other partitioned by multiple columns. Because a `LOCATION` attribute with an `adl://` URL is specified for the database, the tables inside that database are automatically created on ADLS underneath the database directory. To see the names of the associated subdirectories, including the partition key values, we use an ADLS client tool to examine how the directory structure is organized on ADLS. For example, Impala partition directories such as `month=1` do not include leading zeroes, which sometimes appear in partition directories created through Hive.

```
[localhost:21000] > create database db_on_adls location
'adl://impalademo.azuredatalakestore.net/dir1/dir2/dir3';
[localhost:21000] > use db_on_adls;

[localhost:21000] > create table partitioned_on_adls (x int) partitioned by (year int);
[localhost:21000] > alter table partitioned_on_adls add partition (year=2013);
[localhost:21000] > alter table partitioned_on_adls add partition (year=2014);
[localhost:21000] > alter table partitioned_on_adls add partition (year=2015);

[localhost:21000] > ! hadoop fs -ls adl://impalademo.azuredatalakestore.net/dir1/dir2/dir3
 --recursive;
2015-03-17 13:56:34          0 dir1/dir2/dir3/
2015-03-17 16:43:28          0 dir1/dir2/dir3/partitioned_on_adls/
2015-03-17 16:43:49          0 dir1/dir2/dir3/partitioned_on_adls/year=2013/
2015-03-17 16:43:53          0 dir1/dir2/dir3/partitioned_on_adls/year=2014/
2015-03-17 16:43:58          0 dir1/dir2/dir3/partitioned_on_adls/year=2015/

[localhost:21000] > create table partitioned_multiple_keys (x int)
                  >   partitioned by (year smallint, month tinyint, day tinyint);
[localhost:21000] > alter table partitioned_multiple_keys
                  >   add partition (year=2015,month=1,day=1);
[localhost:21000] > alter table partitioned_multiple_keys
                  >   add partition (year=2015,month=1,day=31);
[localhost:21000] > alter table partitioned_multiple_keys
                  >   add partition (year=2015,month=2,day=28);

[localhost:21000] > ! hadoop fs -ls adl://impalademo.azuredatalakestore.net/dir1/dir2/dir3
 --recursive;
2015-03-17 13:56:34          0 dir1/dir2/dir3/
2015-03-17 16:47:13          0 dir1/dir2/dir3/partitioned_multiple_keys/
2015-03-17 16:47:44          0
dir1/dir2/dir3/partitioned_multiple_keys/year=2015/month=1/day=1/
```

```
2015-03-17 16:47:50          0
dir1/dir2/dir3/partitioned_multiple_keys/year=2015/month=1/day=31/
2015-03-17 16:47:57          0
dir1/dir2/dir3/partitioned_multiple_keys/year=2015/month=2/day=28/
2015-03-17 16:43:28          0 dir1/dir2/dir3/partitioned_on_adls/
2015-03-17 16:43:49          0 dir1/dir2/dir3/partitioned_on_adls/year=2013/
2015-03-17 16:43:53          0 dir1/dir2/dir3/partitioned_on_adls/year=2014/
2015-03-17 16:43:58          0 dir1/dir2/dir3/partitioned_on_adls/year=2015/
```

The `CREATE DATABASE` and `CREATE TABLE` statements create the associated directory paths if they do not already exist. You can specify multiple levels of directories, and the `CREATE` statement creates all appropriate levels, similar to using `mkdir -p`.

Use the standard ADLS file upload methods to actually put the data files into the right locations. You can also put the directory paths and data files in place before creating the associated Impala databases or tables, and Impala automatically uses the data from the appropriate location after the associated databases and tables are created.

You can switch whether an existing table or partition points to data in HDFS or ADLS. For example, if you have an Impala table or partition pointing to data files in HDFS or ADLS, and you later transfer those data files to the other filesystem, use an `ALTER TABLE` statement to adjust the `LOCATION` attribute of the corresponding table or partition to reflect that change. This location-switching technique is not practical for entire databases that have a custom `LOCATION` attribute.

Internal and External Tables Located on ADLS

Just as with tables located on HDFS storage, you can designate ADLS-based tables as either internal (managed by Impala) or external, by using the syntax `CREATE TABLE` or `CREATE EXTERNAL TABLE` respectively. When you drop an internal table, the files associated with the table are removed, even if they are on ADLS storage. When you drop an external table, the files associated with the table are left alone, and are still available for access by other tools or components. See Overview of Impala Tables on page 196 for details.

If the data on ADLS is intended to be long-lived and accessed by other tools in addition to Impala, create any associated ADLS tables with the `CREATE EXTERNAL TABLE` syntax, so that the files are not deleted from ADLS when the table is dropped.

If the data on ADLS is only needed for querying by Impala and can be safely discarded once the Impala workflow is complete, create the associated ADLS tables using the `CREATE TABLE` syntax, so that dropping the table also deletes the corresponding data files on ADLS.

For example, this session creates a table in ADLS with the same column layout as a table in HDFS, then examines the ADLS table and queries some data from it. The table in ADLS works the same as a table in HDFS as far as the expected file format of the data, table and column statistics, and other table properties. The only indication that it is not an HDFS table is the `adl://` URL in the `LOCATION` property. Many data files can reside in the ADLS directory, and their combined contents form the table data. Because the data in this example is uploaded after the table is created, a `REFRESH` statement prompts Impala to update its cached information about the data files.

```
[localhost:21000] > create table usa_cities_adls like usa_cities location
'adl://impalademo.azuredatalakestore.net/usa_cities';
[localhost:21000] > desc usa_cities_adls;
+-------+----------+---------+
| name  | type     | comment |
+-------+----------+---------+
| id    | smallint |         |
| city  | string   |         |
| state | string   |         |
+-------+----------+---------+

-- Now from a web browser, upload the same data file(s) to ADLS as in the HDFS table,
-- under the relevant store and path. If you already have the data in ADLS, you would
-- point the table LOCATION at an existing path.

[localhost:21000] > refresh usa_cities_adls;
[localhost:21000] > select count(*) from usa_cities_adls;
```

```
+----------+
| count(*) |
+----------+
| 289      |
+----------+
[localhost:21000] > select distinct state from sample_data_adls limit 5;
+----------------------+
| state                |
+----------------------+
| Louisiana            |
| Minnesota            |
| Georgia              |
| Alaska               |
| Ohio                 |
+----------------------+
[localhost:21000] > desc formatted usa_cities_adls;
+------------------------------+----------------------------------------------------+---------+
| name                         | type                                               |
comment |
+------------------------------+----------------------------------------------------+---------+
| # col_name                   | data_type                                          |
comment |
|                              | NULL                                               |
NULL    |
| id                           | smallint                                           |
NULL    |
| city                         | string                                             |
NULL    |
| state                        | string                                             |
NULL    |
|                              | NULL                                               |
NULL    |
| # Detailed Table Information | NULL                                               |
NULL    |
| Database:                    | adls_testing                                       |
NULL    |
| Owner:                       | jrussell                                           |
NULL    |
| CreateTime:                  | Mon Mar 16 11:36:25 PDT 2017                       |
NULL    |
| LastAccessTime:              | UNKNOWN                                            |
NULL    |
| Protect Mode:                | None                                               |
NULL    |
| Retention:                   | 0                                                  |
NULL    |
| Location:                    | adl://impalademo.azuredatalakestore.net/usa_cities |
NULL    |
| Table Type:                  | MANAGED_TABLE                                      |
NULL    |
...
+------------------------------+----------------------------------------------------+---------+
```

In this case, we have already uploaded a Parquet file with a million rows of data to the `sample_data` directory underneath the `impalademo` store on ADLS. This session creates a table with matching column settings pointing to the corresponding location in ADLS, then queries the table. Because the data is already in place on ADLS when the table is created, no REFRESH statement is required.

```
[localhost:21000] > create table sample_data_adls
                  > (id int, id bigint, val int, zerofill string,
                  > name string, assertion boolean, city string, state string)
                  > stored as parquet location
'adl://impalademo.azuredatalakestore.net/sample_data';
[localhost:21000] > select count(*) from sample_data_adls;
+----------+
| count(*) |
+----------+
| 1000000  |
+----------+
[localhost:21000] > select count(*) howmany, assertion from sample_data_adls group by
```

```
assertion;
+---------+-----------+
| howmany | assertion |
+---------+-----------+
| 667149  | true      |
| 332851  | false     |
+---------+-----------+
```

Running and Tuning Impala Queries for Data Stored on ADLS

Once the appropriate `LOCATION` attributes are set up at the table or partition level, you query data stored in ADLS exactly the same as data stored on HDFS or in HBase:

- Queries against ADLS data support all the same file formats as for HDFS data.
- Tables can be unpartitioned or partitioned. For partitioned tables, either manually construct paths in ADLS corresponding to the HDFS directories representing partition key values, or use `ALTER TABLE ... ADD PARTITION` to set up the appropriate paths in ADLS.
- HDFS, Kudu, and HBase tables can be joined to ADLS tables, or ADLS tables can be joined with each other.
- Authorization using the Sentry framework to control access to databases, tables, or columns works the same whether the data is in HDFS or in ADLS.
- The `catalogd` daemon caches metadata for both HDFS and ADLS tables. Use `REFRESH` and `INVALIDATE METADATA` for ADLS tables in the same situations where you would issue those statements for HDFS tables.
- Queries against ADLS tables are subject to the same kinds of admission control and resource management as HDFS tables.
- Metadata about ADLS tables is stored in the same metastore database as for HDFS tables.
- You can set up views referring to ADLS tables, the same as for HDFS tables.
- The `COMPUTE STATS`, `SHOW TABLE STATS`, and `SHOW COLUMN STATS` statements work for ADLS tables also.

Understanding and Tuning Impala Query Performance for ADLS Data

Although Impala queries for data stored in ADLS might be less performant than queries against the equivalent data stored in HDFS, you can still do some tuning. Here are techniques you can use to interpret explain plans and profiles for queries against ADLS data, and tips to achieve the best performance possible for such queries.

All else being equal, performance is expected to be lower for queries running against data on ADLS rather than HDFS. The actual mechanics of the `SELECT` statement are somewhat different when the data is in ADLS. Although the work is still distributed across the datanodes of the cluster, Impala might parallelize the work for a distributed query differently for data on HDFS and ADLS. ADLS does not have the same block notion as HDFS, so Impala uses heuristics to determine how to split up large ADLS files for processing in parallel. Because all hosts can access any ADLS data file with equal efficiency, the distribution of work might be different than for HDFS data, where the data blocks are physically read using short-circuit local reads by hosts that contain the appropriate block replicas. Although the I/O to read the ADLS data might be spread evenly across the hosts of the cluster, the fact that all data is initially retrieved across the network means that the overall query performance is likely to be lower for ADLS data than for HDFS data.

Because data files written to ADLS do not have a default block size the way HDFS data files do, any Impala `INSERT` or `CREATE TABLE AS SELECT` statements use the `PARQUET_FILE_SIZE` query option setting to define the size of Parquet data files. (Using a large block size is more important for Parquet tables than for tables that use other file formats.)

When optimizing aspects of for complex queries such as the join order, Impala treats tables on HDFS and ADLS the same way. Therefore, follow all the same tuning recommendations for ADLS tables as for HDFS ones, such as using the `COMPUTE STATS` statement to help Impala construct accurate estimates of row counts and cardinality. See Tuning Impala for Performance on page 565 for details.

In query profile reports, the numbers for `BytesReadLocal`, `BytesReadShortCircuit`, `BytesReadDataNodeCached`, and `BytesReadRemoteUnexpected` are blank because those metrics come from HDFS. If you do see any indications that a query against an ADLS table performed “remote read” operations, do not be alarmed. That is expected because, by definition, all the I/O for ADLS tables involves remote reads.

Restrictions on Impala Support for ADLS

Impala requires that the default filesystem for the cluster be HDFS. You cannot use ADLS as the only filesystem in the cluster.

Although ADLS is often used to store JSON-formatted data, the current Impala support for ADLS does not include directly querying JSON data. For Impala queries, use data files in one of the file formats listed in How Impala Works with Hadoop File Formats on page 634. If you have data in JSON format, you can prepare a flattened version of that data for querying by Impala as part of your ETL cycle.

You cannot use the `ALTER TABLE ... SET CACHED` statement for tables or partitions that are located in ADLS.

Best Practices for Using Impala with ADLS

The following guidelines represent best practices derived from testing and real-world experience with Impala on ADLS:

- Any reference to an ADLS location must be fully qualified. (This rule applies when ADLS is not designated as the default filesystem.)
- Set any appropriate configuration settings for `impalad`.

Using Impala Logging

The Impala logs record information about:

- Any errors Impala encountered. If Impala experienced a serious error during startup, you must diagnose and troubleshoot that problem before you can do anything further with Impala.
- How Impala is configured.
- Jobs Impala has completed.



Note:

Formerly, the logs contained the query profile for each query, showing low-level details of how the work is distributed among nodes and how intermediate and final results are transmitted across the network. To save space, those query profiles are now stored in zlib-compressed files in `/var/log/impala/profiles`. You can access them through the Impala web user interface. For example, at `http://impalad-node-hostname:25000/queries`, each query is followed by a `Profile` link leading to a page showing extensive analytical data for the query execution.

The auditing feature introduced in Impala 1.1.1 produces a separate set of audit log files when enabled. See Auditing Impala Operations on page 78 for details.

In CDH 5.12 / Impala 2.9 and higher, you can control how many audit event log files are kept on each host through the `--max_audit_event_log_files` startup option for the `impalad` daemon, similar to the `--max_log_files` option for regular log files.

The lineage feature introduced in Impala 2.2.0 produces a separate lineage log file when enabled. See Viewing Lineage Information for Impala Data on page 80 for details.

Locations and Names of Impala Log Files

- By default, the log files are under the directory `/var/log/impala`. To change log file locations, edit the log file properties in the Impala service in Cloudera Manager (**Impala service** > **Configuration**).
- The significant files for the `impalad` process are `impalad.INFO`, `impalad.WARNING`, and `impalad.ERROR`. You might also see a file `impalad.FATAL`, although this is only present in rare conditions.
- The significant files for the `statestored` process are `statestored.INFO`, `statestored.WARNING`, and `statestored.ERROR`. You might also see a file `statestored.FATAL`, although this is only present in rare conditions.
- The significant files for the `catalogd` process are `catalogd.INFO`, `catalogd.WARNING`, and `catalogd.ERROR`. You might also see a file `catalogd.FATAL`, although this is only present in rare conditions.
- Examine the `.INFO` files to see configuration settings for the processes.
- Examine the `.WARNING` files to see all kinds of problem information, including such things as suboptimal settings and also serious runtime errors.
- Examine the `.ERROR` and/or `.FATAL` files to see only the most serious errors, if the processes crash, or queries fail to complete. These messages are also in the `.WARNING` file.
- A new set of log files is produced each time the associated daemon is restarted. These log files have long names including a timestamp. The `.INFO`, `.WARNING`, and `.ERROR` files are physically represented as symbolic links to the latest applicable log files.
- The init script for the `impala-server` service also produces a consolidated log file `/var/log/impalad/impala-server.log`, with all the same information as the corresponding `.INFO`, `.WARNING`, and `.ERROR` files.
- The init script for the `impala-state-store` service also produces a consolidated log file `/var/log/impalad/impala-state-store.log`, with all the same information as the corresponding `.INFO`, `.WARNING`, and `.ERROR` files.

Impala stores information using the `glog_v` logging system. You will see some messages referring to C++ file names. Logging is affected by:

- The `GLOG_v` environment variable specifies which types of messages are logged. See Setting Logging Levels on page 711 for details.
- The `--logbuflevel` startup flag for the `impalad` daemon specifies how often the log information is written to disk. The default is 0, meaning that the log is immediately flushed to disk when Impala outputs an important messages such as a warning or an error, but less important messages such as informational ones are buffered in memory rather than being flushed to disk immediately.
- Cloudera Manager has an Impala configuration setting that sets the `-logbuflevel` startup option.

Managing Impala Logs through Cloudera Manager or Manually

Cloudera recommends installing Impala through the Cloudera Manager administration interface. To assist with troubleshooting, Cloudera Manager collects front-end and back-end logs together into a single view, and let you do a search across log data for all the managed nodes rather than examining the logs on each node separately. If you installed Impala using Cloudera Manager, refer to the topics on Monitoring Services or Logs.

If you are using Impala in an environment not managed by Cloudera Manager, review Impala log files on each host, when you have traced an issue back to a specific system.

Rotating Impala Logs

Impala periodically rotates logs. It switches the physical files representing the current log files and removes older log files that are no longer needed.

In Impala 2.2 and higher, the `--max_log_files` configuration option specifies how many log files to keep at each severity level (`INFO`, `WARNING`, `ERROR`, and `FATAL`). You can specify an appropriate setting for each Impala-related daemon (`impalad`, `statestored`, and `catalogd`).

- A value of 0 preserves all log files, in which case you would set up set up manual log rotation using your Linux tool or technique of choice.
- A value of 1 preserves only the very latest log file.
- The default value is 10.

For some log levels, Impala logs are first temporarily buffered in memory and only written to disk periodically. The `--logbufsecs` setting controls the maximum time that log messages are buffered for. For example, with the default value of 5 seconds, there may be up to a 5 second delay before a logged message shows up in the log file.

It is not recommended that you set `--logbufsecs` to 0 as the setting makes the Impala daemon to spin in the thread that tries to delete old log files.

To set up log rotation on a system managed by Cloudera Manager:

1. In the Impala Configuration tab, type `max_log_files`.
2. Set the appropriate value for the **Maximum Log Files** field for each Impala configuration category, Impala, Catalog Server, and StateStore.
3. Restart the Impala service.

In earlier Cloudera Manager releases, specify the `-max_log_files=`*`maximum`* option in the **Command Line Argument Advanced Configuration Snippet (Safety Valve)** field for each Impala configuration category.

Reviewing Impala Logs

By default, the Impala log is stored at `/var/log/impalad/`. The most comprehensive log, showing informational, warning, and error messages, is in the file name `impalad.INFO`. View log file contents by using the web interface or by examining the contents of the log file. (When you examine the logs through the file system, you can troubleshoot

problems by reading the `impalad.WARNING` and/or `impalad.ERROR` files, which contain the subsets of messages indicating potential problems.)

On a machine named `impala.example.com` with default settings, you could view the Impala logs on that machine by using a browser to access `http://impala.example.com:25000/logs`.



Note:

The web interface limits the amount of logging information displayed. To view every log entry, access the log files directly through the file system.

You can view the contents of the `impalad.INFO` log file in the file system. With the default configuration settings, the start of the log file appears as follows:

```
[user@example impalad]$ pwd
/var/log/impalad
[user@example impalad]$ more impalad.INFO
Log file created at: 2013/01/07 08:42:12
Running on machine: impala.example.com
Log line format: [IWEF]mmdd hh:mm:ss.uuuuuu threadid file:line] msg
I0107 08:42:12.292155 14876 daemon.cc:34] impalad version 0.4 RELEASE (build
9d7fadca0461ab40b9e9df8cdb47107ec6b27cff)
Built on Fri, 21 Dec 2012 12:55:19 PST
I0107 08:42:12.292484 14876 daemon.cc:35] Using hostname: impala.example.com
I0107 08:42:12.292706 14876 logging.cc:76] Flags (see also /varz are on debug webserver):
--dump_ir=false
--module_output=
--be_port=22000
--classpath=
--hostname=impala.example.com
```



Note: The preceding example shows only a small part of the log file. Impala log files are often several megabytes in size.

Understanding Impala Log Contents

The logs store information about Impala startup options. This information appears once for each time Impala is started and may include:

- Machine name.
- Impala version number.
- Flags used to start Impala.
- CPU information.
- The number of available disks.

There is information about each job Impala has run. Because each Impala job creates an additional set of data about queries, the amount of job specific data may be very large. Logs may contained detailed information on jobs. These detailed log entries may include:

- The composition of the query.
- The degree of data locality.
- Statistics on data throughput and response times.

Setting Logging Levels

Impala uses the GLOG system, which supports three logging levels. You can adjust the logging levels using the Cloudera Manager Admin Console. You can adjust logging levels without going through the Cloudera Manager Admin Console

by exporting variable settings. To change logging settings manually, use a command similar to the following on each node before starting `impalad`:

```
export GLOG_v=1
```



Note: For performance reasons, Cloudera highly recommends not enabling the most verbose logging level of 3.

For more information on how to configure GLOG, including how to set variable logging levels for different system components, see documentation for the glog project on github.

Understanding What is Logged at Different Logging Levels

As logging levels increase, the categories of information logged are cumulative. For example, GLOG_v=2 records everything GLOG_v=1 records, as well as additional information.

Increasing logging levels imposes performance overhead and increases log size. Cloudera recommends using GLOG_v=1 for most cases: this level has minimal performance impact but still captures useful troubleshooting information.

Additional information logged at each level is as follows:

- GLOG_v=1 - The default level. Logs information about each connection and query that is initiated to an `impalad` instance, including runtime profiles.
- GLOG_v=2 - Everything from the previous level plus information for each RPC initiated. This level also records query execution progress information, including details on each file that is read.
- GLOG_v=3 - Everything from the previous level plus logging of every row that is read. This level is only applicable for the most serious troubleshooting and tuning scenarios, because it can produce exceptionally large and detailed log files, potentially leading to its own set of performance and capacity problems.

Redacting Sensitive Information from Impala Log Files

Log redaction is a security feature that prevents sensitive information from being displayed in locations used by administrators for monitoring and troubleshooting, such as log files, the Cloudera Manager user interface, and the Impala debug web user interface. You configure regular expressions that match sensitive types of information processed by your system, such as credit card numbers or tax IDs, and literals matching these patterns are obfuscated wherever they would normally be recorded in log files or displayed in administration or debugging user interfaces.

In a security context, the log redaction feature is complementary to the Sentry authorization framework. Sentry prevents unauthorized users from being able to directly access table data. Redaction prevents administrators or support personnel from seeing the smaller amounts of sensitive or personally identifying information (PII) that might appear in queries issued by those authorized users.

See the CDH Security Guide for details about how to enable this feature and set up the regular expressions to detect and redact sensitive information within SQL statement text.

Impala Client Access

Application developers have a number of options to interface with Impala. The core development language with Impala is SQL. You can also use Java or other languages to interact with Impala through the standard JDBC and ODBC interfaces used by many business intelligence tools. For specialized kinds of analysis, you can supplement the Impala built-in functions by writing user-defined functions in C++ or Java.

You can connect and submit requests to the Impala through:

- The impala-shell interactive command interpreter
- The Hue web-based user interface
- JDBC
- ODBC

Each `impalad` daemon process, running on separate nodes in a cluster, listens to several ports for incoming requests:

- Requests from `impala-shell` and Hue are routed to the `impalad` daemons through the same port.
- The `impalad` daemons listen on separate ports for JDBC and ODBC requests.

Impala Startup Options for Client Connections

The following options control client connections to Impala.

--fe_service_threads

Specifies the maximum number of concurrent client connections allowed. The default value is 64 with which 64 queries can run simultaneously.

If you have more clients trying to connect to Impala than the value of this setting, the later arriving clients have to wait for the duration specified by `--accepted_client_cnxn_timeout`. You can increase this value to allow more client connections. However, a large value means more threads to be maintained even if most of the connections are idle, and it could negatively impact query latency. Client applications should use the connection pool to avoid need for large number of sessions.

--accepted_client_cnxn_timeout

Controls how Impala treats new connection requests if it has run out of the number of threads configured by `--fe_service_threads`

If `--accepted_client_cnxn_timeout > 0`, new connection requests are rejected if Impala can't get a server thread within the specified (in seconds) timeout.

If `--accepted_client_cnxn_timeout=0`, i.e. no timeout, clients wait indefinitely to open the new session until more threads are available.

The default timeout is 5 minutes.

The timeout applies only to client facing thrift servers, i.e., HS2 and Beeswax servers.

Using the Impala Shell (impala-shell Command)

You can use the Impala shell tool (`impala-shell`) to set up databases and tables, insert data, and issue queries. For ad hoc queries and exploration, you can submit SQL statements in an interactive session. To automate your work, you can specify command-line options to process a single statement or a script file. The `impala-shell` interpreter accepts all the same SQL statements listed in Impala SQL Statements on page 202, plus some shell-only commands that you can use for tuning performance and diagnosing problems.

The `impala-shell` command fits into the familiar Unix toolchain:

- The `-q` option lets you issue a single query from the command line, without starting the interactive interpreter. You could use this option to run `impala-shell` from inside a shell script or with the command invocation syntax from a Python, Perl, or other kind of script.
- The `-f` option lets you process a file containing multiple SQL statements, such as a set of reports or DDL statements to create a group of tables and views.
- The `--var` option lets you pass substitution variables to the statements that are executed by that `impala-shell` session, for example the statements in a script file processed by the `-f` option. You encode the substitution variable on the command line using the notation `--var=variable_name=value`. Within a SQL statement, you substitute the value by using the notation `${var:variable_name}`. This feature is available in CDH 5.7 / Impala 2.5 and higher.
- The `-o` option lets you save query output to a file.
- The `-B` option turns off pretty-printing, so that you can produce comma-separated, tab-separated, or other delimited text files as output. (Use the `--output_delimiter` option to choose the delimiter character; the default is the tab character.)
- In non-interactive mode, query output is printed to `stdout` or to the file specified by the `-o` option, while incidental output is printed to `stderr`, so that you can process just the query output as part of a Unix pipeline.
- In interactive mode, `impala-shell` uses the `readline` facility to recall and edit previous commands.

Cloudera Manager installs `impala-shell` automatically. You might install `impala-shell` manually on other systems not managed by Cloudera Manager, so that you can issue queries from client systems that are not also running the Impala daemon or other Apache Hadoop components.

For information about establishing a connection to a coordinator Impala daemon through the `impala-shell` command, see Connecting to impalad through impala-shell on page 718.

For a list of the `impala-shell` command-line options, see impala-shell Configuration Options on page 714. For reference information about the `impala-shell` interactive commands, see impala-shell Command Reference on page 721.

impala-shell Configuration Options

You can specify the following options when starting the `impala-shell` command to change how shell commands are executed. The table shows the format to use when specifying each option on the command line, or through the `$HOME/.impalarc` configuration file.



Note:

These options are different than the configuration options for the `impalad` daemon itself. For the `impalad` options, see Modifying Impala Startup Options.

Summary of impala-shell Configuration Options

The following table shows the names and allowed arguments for the `impala-shell` configuration options. You can specify options on the command line, or in a configuration file as described in impala-shell Configuration File on page 717.

Command-Line Option	Configuration File Setting	Explanation
-B or --delimited	write_delimited=true	Causes all query results to be printed in plain format as a delimited text file. Useful for producing data files to be used with other Hadoop components. Also useful for avoiding the performance overhead of pretty-printing all output, especially when running benchmark tests using queries returning large result sets. Specify the delimiter character with the `--output_delimiter` option. Store all query results in a file rather than printing to the screen with the `-B` option. Added in Impala 1.0.1.

Command-Line Option	Configuration File Setting	Explanation
-b or --kerberos_host_fqdn	kerberos_host_fqdn= *load-balancer-hostname*	If set, the setting overrides the expected hostname of the Impala daemon's Kerberos service principal. `impala-shell` will check that the server's principal matches this hostname. This may be used when `impalad` is configured to be accessed via a load-balancer, but it is desired for impala-shell to talk to a specific `impalad` directly.
--print_header	print_header=true	
-o *filename* or --output_file *filename*	output_file=*filename*	Stores all query results in the specified file. Typically used to store the results of a single query issued from the command line with the `-q` option. Also works for interactive sessions; you see the messages such as number of rows fetched, but not the actual result set. To suppress these incidental messages when combining the `-q` and `-o` options, redirect `stderr` to `/dev/null`. Added in Impala 1.0.1.
--output_delimiter=*character*	output_delimiter=*character*	Specifies the character to use as a delimiter between fields when query results are printed in plain format by the `-B` option. Defaults to tab (`'\t'`). If an output value contains the delimiter character, that field is quoted, escaped by doubling quotation marks, or both. Added in Impala 1.0.1.
-p or --show_profiles	show_profiles=true	Displays the query execution plan (same output as the `EXPLAIN` statement) and a more detailed low-level breakdown of execution steps, for every query executed by the shell.
-h or --help	N/A	Displays help information.
N/A	history_max=1000	Sets the maximum number of queries to store in the history file.
-i *hostname* or --impalad=*hostname*[:*portnum*]	impalad=*hostname*[:*portnum*]	Connects to the `impalad` daemon on the specified host. The default port of 21000 is assumed unless you provide another value. You can connect to any host in your cluster that is running `impalad`. If you connect to an instance of `impalad` that was started with an alternate port specified by the `--fe_port` flag, provide that alternative port.
-q *query* or --query=*query*	query=*query*	Passes a query or other `impala-shell` command from the command line. The `impala-shell` interpreter immediately exits after processing the statement. It is limited to a single statement, which could be a `SELECT`, `CREATE TABLE`, `SHOW TABLES`, or any other statement recognized in `impala-shell`. Because you cannot pass a `USE` statement and another query, fully qualify the names for any tables outside the `default` database. (Or use the `-f` option to pass a file with a `USE` statement followed by other queries.)
-f *query_file* or --query_file=*query_file*	query_file=*path_to_query_file*	Passes a SQL query from a file. Multiple statements must be semicolon (;) delimited. In CDH 5.5 / Impala 2.3 and higher, you can specify a filename of `-` to represent

Command-Line Option	Configuration File Setting	Explanation
		standard input. This feature makes it convenient to use `impala-shell` as part of a Unix pipeline where SQL statements are generated dynamically by other tools.
-k or --kerberos	use_kerberos=true	Kerberos authentication is used when the shell connects to `impalad`. If Kerberos is not enabled on the instance of `impalad` to which you are connecting, errors are displayed. See Enabling Kerberos Authentication for Impala for the steps to set up and use Kerberos authentication in Impala.
–query_option="*option=value*" -Q "*option=value*"	Header line `[impala.query_options]`, followed on subsequent lines by *option=value*, one option per line.	Sets default query options for an invocation of the `impala-shell` command. To set multiple query options at once, use more than one instance of this command-line option. The query option names are not case-sensitive.
-s *kerberos_service_name* or –kerberos_service_name=*name*	kerberos_service_name=*name*	Instructs `impala-shell` to authenticate to a particular `impalad` service principal. If a *kerberos_service_name* is not specified, `impala` is used by default. If this option is used in conjunction with a connection in which Kerberos is not supported, errors are returned.
-V or --verbose	verbose=true	Enables verbose output.
--quiet	verbose=false	Disables verbose output.
-v or --version	version=true	Displays version information.
-c	ignore_query_failure=true	Continues on query failure.
-d *default_db* or --database=*default_db*	default_db=*default_db*	Specifies the database to be used on startup. Same as running the USE statement after connecting. If not specified, a database named DEFAULT is used.
--ssl	ssl=true	Enables TLS/SSL for `impala-shell`.
--ca_cert=*path_to_certificate*	ca_cert=*path_to_certificate*	The local pathname pointing to the third-party CA certificate, or to a copy of the server certificate for self-signed server certificates. If `--ca_cert` is not set, `impala-shell` enables TLS/SSL, but does not validate the server certificate. This is useful for connecting to a known-good Impala that is only running over TLS/SSL, when a copy of the certificate is not available (such as when debugging customer installations).
-l	use_ldap=true	Enables LDAP authentication.
-u	user=*user_name*	Supplies the username, when LDAP authentication is enabled by the `-l` option. (Specify the short username, not the full LDAP distinguished name.) The shell then prompts interactively for the password.
–ldap_password_cmd=*command*	N/A	Specifies a command to run to retrieve the LDAP password, when LDAP authentication is enabled by the `-l` option. If the command includes space-separated arguments, enclose the command and its arguments in quotation marks.

Command-Line Option	Configuration File Setting	Explanation
--config_file=*path_to_config_file*	N/A	Specifies the path of the file containing `impala-shell` configuration settings. The default is `$HOME/.impalarc`. This setting can only be specified on the command line.
--live_progress	N/A	Prints a progress bar showing roughly the percentage complete for each query. The information is updated interactively as the query progresses. See LIVE_PROGRESS Query Option (CDH 5.5 or higher only) on page 336.
--live_summary	N/A	Prints a detailed report, similar to the SUMMARY command, showing progress details for each phase of query execution. The information is updated interactively as the query progresses. See LIVE_SUMMARY Query Option (CDH 5.5 or higher only) on page 337.
--var=*variable_name=value*	N/A	Defines a substitution variable that can be used within the `impala-shell` session. The variable can be substituted into statements processed by the `-q` or `-f` options, or in an interactive shell session. Within a SQL statement, you substitute the value by using the notation `${var:variable_name}`. This feature is available in CDH 5.7 / Impala 2.5 and higher.
--auth_creds_ok_in_clear	N/A	Allows LDAP authentication to be used with an insecure connection to the shell. WARNING: This will allow authentication credentials to be sent unencrypted, and hence may be vulnerable to an attack.

impala-shell Configuration File

You can define a set of default options for your `impala-shell` environment, stored in the file `$HOME/.impalarc`. This file consists of key-value pairs, one option per line. Everything after a # character on a line is treated as a comment and ignored.

The configuration file must contain a header label `[impala]`, followed by the options specific to `impala-shell`. (This standard convention for configuration files lets you use a single file to hold configuration options for multiple applications.)

To specify a different filename or path for the configuration file, specify the argument `--config_file=path_to_config_file` on the `impala-shell` command line.

The names of the options in the configuration file are similar (although not necessarily identical) to the long-form command-line arguments to the `impala-shell` command. For the names to use, see Summary of impala-shell Configuration Options on page 714.

Any options you specify on the `impala-shell` command line override any corresponding options within the configuration file.

The following example shows a configuration file that you might use during benchmarking tests. It sets verbose mode, so that the output from each SQL query is followed by timing information. `impala-shell` starts inside the database containing the tables with the benchmark data, avoiding the need to issue a USE statement or use fully qualified table names.

In this example, the query output is formatted as delimited text rather than enclosed in ASCII art boxes, and is stored in a file rather than printed to the screen. Those options are appropriate for benchmark situations, so that the overhead of `impala-shell` formatting and printing the result set does not factor into the timing measurements. It also enables

the show_profiles option. That option prints detailed performance information after each query, which might be valuable in understanding the performance of benchmark queries.

```
[impala]
verbose=true
default_db=tpc_benchmarking
write_delimited=true
output_delimiter=,
output_file=/home/tester1/benchmark_results.csv
show_profiles=true
```

The following example shows a configuration file that connects to a specific remote Impala node, runs a single query within a particular database, then exits. Any query options predefined under the [impala.query_options] section in the configuration file take effect during the session.

You would typically use this kind of single-purpose configuration setting with the impala-shell command-line option --config_file=*path_to_config_file*, to easily select between many predefined queries that could be run against different databases, hosts, or even different clusters. To run a sequence of statements instead of a single query, specify the configuration option query_file=*path_to_query_file* instead.

```
[impala]
impalad=impala-test-node1.example.com
default_db=site_stats
# Issue a predefined query and immediately exit.
query=select count(*) from web_traffic where event_date = trunc(now(),'dd')

[impala.query_options]
mem_limit=32g
```

Connecting to impalad through impala-shell

Within an impala-shell session, you can only issue queries while connected to an instance of the impalad daemon. You can specify the connection information:

- Through command-line options when you run the impala-shell command.
- Through a configuration file that is read when you run the impala-shell command.
- During an impala-shell session, by issuing a CONNECT command.

See impala-shell Configuration Options on page 714 for the command-line and configuration file options you can use.

You can connect to any Impala daemon (impalad), and that daemon coordinates the execution of all queries sent to it.

For simplicity during development, you might always connect to the same host, perhaps running impala-shell on the same host as impalad and specifying the hostname as localhost.

In a production environment, you might enable load balancing, in which you connect to specific host/port combination but queries are forwarded to arbitrary hosts. This technique spreads the overhead of acting as the coordinator node among all the Impala daemons in the cluster. See Using Impala through a Proxy for High Availability on page 72 for details.

To connect the Impala shell during shell startup:

1. Locate the hostname that is running an instance of the impalad daemon. If that impalad uses a non-default port (something other than port 21000) for impala-shell connections, find out the port number also.
2. Use the -i option to the impala-shell interpreter to specify the connection information for that instance of impalad:

```
# When you are logged into the same machine running impalad.
# The prompt will reflect the current hostname.
$ impala-shell

# When you are logged into the same machine running impalad.
# The host will reflect the hostname 'localhost'.
```

```
$ impala-shell -i localhost

# When you are logged onto a different host, perhaps a client machine
# outside the Hadoop cluster.
$ impala-shell -i some.other.hostname

# When you are logged onto a different host, and impalad is listening
# on a non-default port. Perhaps a load balancer is forwarding requests
# to a different host/port combination behind the scenes.
$ impala-shell -i some.other.hostname:port_number
```

To connect the Impala shell after shell startup:

1. Start the Impala shell with no connection:

```
impala-shell
```

You should see a prompt like the following:

```
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) year Cloudera, Inc. All rights reserved.

(Shell
      build version: Impala Shell v3.3.x (hash) built on
      date)
[Not connected] >
```

2. Locate the hostname that is running the `impalad` daemon. If that `impalad` uses a non-default port (something other than port 21000) for `impala-shell` connections, find out the port number also.
3. Use the `connect` command to connect to an Impala instance. Enter a command of the form:

```
[Not connected] > connect impalad-host
[impalad-host:21000] >
```



Note: Replace *impalad-host* with the hostname you have configured to run Impala in your environment. The changed prompt indicates a successful connection.

To start `impala-shell` in a specific database:

You can use all the same connection options as in previous examples. For simplicity, these examples assume that you are logged into one of the Impala daemons.

1. Find the name of the database containing the relevant tables, views, and so on that you want to operate on.
2. Use the `-d` option to the `impala-shell` interpreter to connect and immediately switch to the specified database, without the need for a USE statement or fully qualified names:

```
# Subsequent queries with unqualified names operate on
# tables, views, and so on inside the database named 'staging'.
$ impala-shell -i localhost -d staging

# It is common during development, ETL, benchmarking, and so on
# to have different databases containing the same table names
# but with different contents or layouts.
$ impala-shell -i localhost -d parquet_snappy_compression
$ impala-shell -i localhost -d parquet_gzip_compression
```

To run one or several statements in non-interactive mode:

You can use all the same connection options as in previous examples. For simplicity, these examples assume that you are logged into one of the Impala daemons.

1. Construct a statement, or a file containing a sequence of statements, that you want to run in an automated way, without typing or copying and pasting each time.
2. Invoke `impala-shell` with the `-q` option to run a single statement, or the `-f` option to run a sequence of statements from a file. The `impala-shell` command returns immediately, without going into the interactive interpreter.

```
# A utility command that you might run while developing shell scripts
# to manipulate HDFS files.
$ impala-shell -i localhost -d database_of_interest -q 'show tables'

# A sequence of CREATE TABLE, CREATE VIEW, and similar DDL statements
# can go into a file to make the setup process repeatable.
$ impala-shell -i localhost -d database_of_interest -f recreate_tables.sql
```

Running Commands and SQL Statements in impala-shell

The following are a few of the key syntax and usage rules for running commands and SQL statements in `impala-shell`.

- To see the full set of available commands, press **TAB** twice.
- To cycle through and edit previous commands, click the up-arrow and down-arrow keys.
- Use the standard set of keyboard shortcuts in GNU Readline library for editing and cursor movement, such as **Ctrl-A** for the beginning of line and **Ctrl-E** for the end of line.
- Commands and SQL statements must be terminated by a semi-colon.
- Commands and SQL statements can span multiple lines.
- Use `--` to denote a single-line comment and /* */ to denote a multi-line comment.

  A comment is considered part of the statement it precedes, so when you enter a `--` or `/* */` comment, you get a continuation prompt until you finish entering a statement ending with a semicolon. For example:

```
[impala] > -- This is a test comment
                 > SHOW TABLES LIKE 't*';
```

- If a comment contains the `${variable_name}` and it is not for a variable substitution, the $ character must be escaped, e.g. `-- \${hello}`.

For information on available `impala-shell` commands, see impala-shell Command Reference on page 721.

Variable Substitution in impala-shell

In CDH 5.7 / Impala 2.5 and higher, you can define substitution variables to be used within SQL statements processed by `impala-shell`.

1. You specify the variable and its value as below.
   - On the command line, you specify the option `--var=variable_name=value`
   - Within an interactive session or a script file processed by the `-f` option, use the SET `VAR:variable_name=value` command.
2. Use the above variable in SQL statements in the `impala-shell` session using the notation: `${VAR:variable_name}`.



Note: Because this feature is part of `impala-shell` rather than the `impalad` backend, make sure the client system you are connecting from has the most recent `impala-shell`. You can use this feature with a new `impala-shell` connecting to an older `impalad`, but not the reverse.

For example, here are some `impala-shell` commands that define substitution variables and then use them in SQL statements executed through the `-q` and `-f` options. Notice how the `-q` argument strings are single-quoted to prevent

shell expansion of the `${var:value}` notation, and any string literals within the queries are enclosed by double quotation marks.

```
$ impala-shell --var=tname=table1 --var=colname=x --var=coltype=string -q 'CREATE TABLE
 ${var:tname} (${var:colname} ${var:coltype}) STORED AS PARQUET'
Query: CREATE TABLE table1 (x STRING) STORED AS PARQUET
```

The below example shows a substitution variable passed in by the `--var` option, and then referenced by statements issued interactively. Then the variable is reset with the SET command.

```
$ impala-shell --quiet --var=tname=table1

[impala] > SELECT COUNT(*) FROM ${var:tname};

[impala] > SET VAR:tname=table2;
[impala] > SELECT COUNT(*) FROM ${var:tname};
```

impala-shell Command Reference

Use the following commands within `impala-shell` to pass requests to the `impalad` daemon that the shell is connected to. You can enter a command interactively at the prompt, or pass it as the argument to the `-q` option of `impala-shell`. Most of these commands are passed to the Impala daemon as SQL statements; refer to the corresponding SQL language reference sections for full syntax details.

Command	Explanation
`alter`	Changes the underlying structure or settings of an Impala table, or a table shared between Impala and Hive. See ALTER TABLE Statement on page 205 and ALTER VIEW Statement on page 218 for details.
`compute stats`	Gathers important performance-related information for a table, used by Impala to optimize queries. See COMPUTE STATS Statement on page 219 for details.
`connect`	Connects to the specified instance of `impalad`. The default port of 21000 is assumed unless you provide another value. You can connect to any host in your cluster that is running `impalad`. If you connect to an instance of `impalad` that was started with an alternate port specified by the `--fe_port` flag, you must provide that alternate port. See Connecting to impalad through impala-shell on page 718 for examples. The SET statement has no effect until the `impala-shell` interpreter is connected to an Impala server. Once you are connected, any query options you set remain in effect as you issue a subsequent `CONNECT` command to connect to a different Impala host.
`describe`	Shows the columns, column data types, and any column comments for a specified table. `DESCRIBE FORMATTED` shows additional information such as the HDFS data directory, partitions, and internal properties for the table. See DESCRIBE Statement on page 251 for details about the basic `DESCRIBE` output and the `DESCRIBE FORMATTED` variant. You can use `DESC` as shorthand for the `DESCRIBE` command.
`drop`	Removes a schema object, and in some cases its associated data files. See DROP TABLE Statement on page 268, DROP VIEW Statement on page 270, DROP DATABASE Statement on page 262, and DROP FUNCTION Statement on page 263 for details.
`explain`	Provides the execution plan for a query. `EXPLAIN` represents a query as a series of steps. For example, these steps might be map/reduce stages, metastore operations, or file system operations such as move or rename. See EXPLAIN Statement on page 271 and Using the EXPLAIN Plan for Performance Tuning on page 602 for details.
`help`	Help provides a list of all available commands and options.

Command	Explanation
`history`	Maintains an enumerated cross-session command history. This history is stored in the `~/.impalahistory` file.
`insert`	Writes the results of a query to a specified table. This either overwrites table data or appends data to the existing table content. See INSERT Statement on page 277 for details.
`invalidate metadata`	Updates `impalad` metadata for table existence and structure. Use this command after creating, dropping, or altering databases, tables, or partitions in Hive. See INVALIDATE METADATA Statement on page 286 for details.
`profile`	Displays low-level information about the most recent query. Used for performance diagnosis and tuning. The report starts with the same information as produced by the `EXPLAIN` statement and the `SUMMARY` command. See Using the Query Profile for Performance Tuning on page 604 for details.
`quit`	Exits the shell. Remember to include the final semicolon so that the shell recognizes the end of the command.
`refresh`	Refreshes `impalad` metadata for the locations of HDFS blocks corresponding to Impala data files. Use this command after loading new data files into an Impala table through Hive or through HDFS commands. See REFRESH Statement on page 291 for details.
`rerun` or `@`	Executes a previous `impala-shell` command again, from the list of commands displayed by the `history` command. These could be SQL statements, or commands specific to `impala-shell` such as `quit` or `profile`. Specify an integer argument. A positive integer `N` represents the command labelled `N` in the output of the `HISTORY` command. A negative integer `-N` represents the `N`th command from the end of the list, such as `-1` for the most recent command. Commands that are executed again do not produce new entries in the `HISTORY` output list.
`select`	Specifies the data set on which to complete some action. All information returned from `select` can be sent to some output such as the console or a file or can be used to complete some other element of query. See SELECT Statement on page 295 for details.
`set`	Manages query options for an `impala-shell` session. The available options are the ones listed in Query Options for the SET Statement on page 322. These options are used for query tuning and troubleshooting. Issue `SET` with no arguments to see the current query options, either based on the `impalad` defaults, as specified by you at `impalad` startup, or based on earlier `SET` statements in the same session. To modify option values, issue commands with the syntax `set option=value`. To restore an option to its default, use the `unset` command. The `SET` statement has no effect until the `impala-shell` interpreter is connected to an Impala server. Once you are connected, any query options you set remain in effect as you issue a subsequent `CONNECT` command to connect to a different Impala host. In Impala 2.0 and later, `SET` is available as a SQL statement for any kind of application as well as in `impala-shell`. See SET Statement on page 321 for details.
`shell`	Executes the specified command in the operating system shell without exiting `impala-shell`. You can use the `!` character as shorthand for the `shell` command. **Note:** Quote any instances of the `--` or `/*` tokens to avoid them being interpreted as the start of a comment. To embed comments within `source` or `!` commands, use the shell comment character `#` before the comment portion of the line.

Command	Explanation
`show`	Displays metastore data for schema objects created and accessed through Impala, Hive, or both. `show` can be used to gather information about objects such as databases, tables, and functions. See SHOW Statement on page 363 for details.
`source` or `src`	Executes one or more statements residing in a specified file from the local filesystem. Allows you to perform the same kinds of batch operations as with the `-f` option, but interactively within the interpreter. The file can contain SQL statements and other `impala-shell` commands, including additional `SOURCE` commands to perform a flexible sequence of actions. Each command or statement, except the last one in the file, must end with a semicolon. See Running Commands and SQL Statements in impala-shell on page 720 for examples.
`summary`	Summarizes the work performed in various stages of a query. It provides a higher-level view of the information displayed by the `EXPLAIN` command. Added in Impala 1.4.0. See Using the SUMMARY Report for Performance Tuning on page 603 for details about the report format and how to interpret it. The time, memory usage, and so on reported by `SUMMARY` only include the portions of the statement that read data, not when data is written. Therefore, the `PROFILE` command is better for checking the performance and scalability of `INSERT` statements. In CDH 5.5 / Impala 2.3 and higher, you can see a continuously updated report of the summary information while a query is in progress. See LIVE_SUMMARY Query Option (CDH 5.5 or higher only) on page 337 for details.
`unset`	Removes any user-specified value for a query option and returns the option to its default value. See Query Options for the SET Statement on page 322 for the available query options. In CDH 5.7 / Impala 2.5 and higher, it can also remove user-specified substitution variables using the notation `UNSET VAR:`*`variable_name`*.
`use`	Indicates the database against which to execute subsequent commands. Lets you avoid using fully qualified names when referring to tables in databases other than `default`. See USE Statement on page 385 for details. Not effective with the `-q` option, because that option only allows a single statement in the argument.
`version`	Returns Impala version information.

Configuring Impala to Work with ODBC

Third-party products can be designed to integrate with Impala using ODBC. For the best experience, ensure any third-party product you intend to use is supported. Verifying support includes checking that the versions of Impala, ODBC, the operating system, and the third-party product have all been approved for use together. Before configuring your systems to use ODBC, download a connector. You may need to sign in and accept license agreements before accessing the pages required for downloading ODBC connectors.

Downloading the ODBC Driver



Important: As of late 2015, most business intelligence applications are certified with the 2.x ODBC drivers. Although the instructions on this page cover both the 2.x and 1.x drivers, expect to use the 2.x drivers exclusively for most ODBC applications connecting to Impala. CDH 6.0 has been tested with the Impala ODBC driver version 2.5.42, and Cloudera recommends that you use this version when you start using CDH 6.0.

See the database drivers section on the Cloudera downloads web page to download and install the driver.

Configuring the ODBC Port

Versions 2.5 and 2.0 of the Cloudera ODBC Connector, currently certified for some but not all BI applications, use the HiveServer2 protocol, corresponding to Impala port 21050. Impala supports Kerberos authentication with all the supported versions of the driver, and requires ODBC 2.05.13 for Impala or higher for LDAP username/password authentication.

Version 1.x of the Cloudera ODBC Connector uses the original HiveServer1 protocol, corresponding to Impala port 21000.

Example of Setting Up an ODBC Application for Impala

To illustrate the outline of the setup process, here is a transcript of a session to set up all required drivers and a business intelligence application that uses the ODBC driver, under Mac OS X. Each `.dmg` file runs a GUI-based installer, first for the underlying IODBC driver needed for non-Windows systems, then for the Cloudera ODBC Connector, and finally for the BI tool itself.

```
$ ls -1
Cloudera-ODBC-Driver-for-Impala-Install-Guide.pdf
BI_Tool_Installer.dmg
iodbc-sdk-3.52.7-macosx-10.5.dmg
ClouderaImpalaODBC.dmg
$ open iodbc-sdk-3.52.7-macosx-10.dmg
Install the IODBC driver using its installer
$ open ClouderaImpalaODBC.dmg
Install the Cloudera ODBC Connector using its installer
$ installer_dir=$(pwd)
$ cd /opt/cloudera/impalaodbc
$ ls -1
Cloudera ODBC Driver for Impala Install Guide.pdf
Readme.txt
Setup
lib
ErrorMessages
Release Notes.txt
Tools
$ cd Setup
$ ls
odbc.ini    odbcinst.ini
$ cp odbc.ini ~/.odbc.ini
$ vi ~/.odbc.ini
$ cat ~/.odbc.ini
[ODBC]
# Specify any global ODBC configuration here such as ODBC tracing.

[ODBC Data Sources]
Sample Cloudera Impala DSN=Cloudera ODBC Driver for Impala

[Sample Cloudera Impala DSN]

# Description: DSN Description.
# This key is not necessary and is only to give a description of the data source.
Description=Cloudera ODBC Driver for Impala DSN

# Driver: The location where the ODBC driver is installed to.
Driver=/opt/cloudera/impalaodbc/lib/universal/libclouderaimpalaodbc.dylib

# The DriverUnicodeEncoding setting is only used for SimbaDM
# When set to 1, SimbaDM runs in UTF-16 mode.
# When set to 2, SimbaDM runs in UTF-8 mode.
#DriverUnicodeEncoding=2

# Values for HOST, PORT, KrbFQDN, and KrbServiceName should be set here.
# They can also be specified on the connection string.
HOST=hostname.sample.example.com
PORT=21050
Schema=default

# The authentication mechanism.
# 0 - No authentication (NOSASL)
```

```
# 1 - Kerberos authentication (SASL)
# 2 - Username authentication (SASL)
# 3 - Username/password authentication (SASL)
# 4 - Username/password authentication with SSL (SASL)
# 5 - No authentication with SSL (NOSASL)
# 6 - Username/password authentication (NOSASL)
AuthMech=0

# Kerberos related settings.
KrbFQDN=
KrbRealm=
KrbServiceName=

# Username/password authentication with SSL settings.
UID=
PWD
CAIssuedCertNamesMismatch=1
TrustedCerts=/opt/cloudera/impalaodbc/lib/universal/cacerts.pem

# Specify the proxy user ID to use.
#DelegationUID=

# General settings
TSaslTransportBufSize=1000
RowsFetchedPerBlock=10000
SocketTimeout=0
StringColumnLength=32767
UseNativeQuery=0
$ pwd
/opt/cloudera/impalaodbc/Setup
$ cd $installer_dir
$ open BI_Tool_Installer.dmg
Install the BI tool using its installer
$ ls /Applications | grep BI_Tool
BI_Tool.app
$ open -a BI_Tool.app
In the BI tool, connect to a data source using port 21050
```

Notes about JDBC and ODBC Interaction with Impala SQL Features

Most Impala SQL features work equivalently through the `impala-shell` interpreter of the JDBC or ODBC APIs. The following are some exceptions to keep in mind when switching between the interactive shell and applications using the APIs:



Note: If your JDBC or ODBC application connects to Impala through a load balancer such as `haproxy`, be cautious about reusing the connections. If the load balancer has set up connection timeout values, either check the connection frequently so that it never sits idle longer than the load balancer timeout value, or check the connection validity before using it and create a new one if the connection has been closed.

Configuring Impala to Work with JDBC

Impala supports the standard JDBC interface, allowing access from commercial Business Intelligence tools and custom software written in Java or other programming languages. The JDBC driver allows you to access Impala from a Java program that you write, or a Business Intelligence or similar tool that uses JDBC to communicate with various database products.

Setting up a JDBC connection to Impala involves the following steps:

- Verifying the communication port where the Impala daemons in your cluster are listening for incoming JDBC requests.
- Installing the JDBC driver on every system that runs the JDBC-enabled application.
- Specifying a connection string for the JDBC application to access one of the servers running the `impalad` daemon, with the appropriate security settings.

Configuring the JDBC Port

The default port used by JDBC 2.0 and later (as well as ODBC 2.x) is 21050. Impala server accepts JDBC connections through this same port 21050 by default. Make sure this port is available for communication with other hosts on your network, for example, that it is not blocked by firewall software. If your JDBC client software connects to a different port, specify that alternative port number with the `--hs2_port` option when starting `impalad`. See Impala Startup Options for details about Impala startup options. See Ports Used by Impala on page 743 for information about all ports used for communication between Impala and clients or between Impala components.

Choosing the JDBC Driver

In Impala 2.0 and later, you have the choice between the Cloudera JDBC Connector and the Hive 0.13 or higher JDBC driver. Cloudera recommends using the Cloudera JDBC Connector where practical.

If you are already using JDBC applications with an earlier Impala release, you must update your JDBC driver to one of these choices, because the Hive 0.12 driver that was formerly the only choice is not compatible with Impala 2.0 and later.

Both the Cloudera JDBC 2.5 Connector and the Hive JDBC driver provide a substantial speed increase for JDBC applications with Impala 2.0 and higher, for queries that return large result sets.



Important: CDH 6.0 has been tested using the Impala JDBC driver version 2.5.45 and 2.6.2. Cloudera recommends that you use one of these two versions when you start using Impala in CDH 6.0.

Enabling Impala JDBC Support on Client Systems

Using the Cloudera JDBC Connector (recommended)

You download and install the Cloudera JDBC 2.5 connector on any Linux, Windows, or Mac system where you intend to run JDBC-enabled applications. From the Cloudera Connectors download page, you choose the appropriate protocol (JDBC or ODBC) and target product (Impala or Hive). The ease of downloading and installing on a wide variety of systems makes this connector a convenient choice for organizations with heterogeneous environments.

Using the Hive JDBC Driver

You install the Hive JDBC driver (`hive-jdbc` package) through the Linux package manager, on hosts within the CDH cluster. The driver consists of several JAR files. The same driver can be used by Impala and Hive.

To get the JAR files, install the Hive JDBC driver on each host in the cluster that will run JDBC applications. Follow the instructions for Installing Cloudera JDBC and ODBC Drivers on Clients in CDH.



Note: The latest JDBC driver, corresponding to Hive 0.13, provides substantial performance improvements for Impala queries that return large result sets. Impala 2.0 and later are compatible with the Hive 0.13 driver. If you already have an older JDBC driver installed, and are running Impala 2.0 or higher, consider upgrading to the latest Hive JDBC driver for best performance with JDBC applications.

If you are using JDBC-enabled applications on hosts outside the CDH cluster, you cannot use the CDH install procedure on the non-CDH hosts. Install the JDBC driver on at least one CDH host using the preceding procedure. Then download the JAR files to each client machine that will use JDBC with Impala:

```
commons-logging-X.X.X.jar
  hadoop-common.jar
  hive-common-X.XX.X-cdhX.X.X.jar
  hive-jdbc-X.XX.X-cdhX.X.X.jar
  hive-metastore-X.XX.X-cdhX.X.X.jar
  hive-service-X.XX.X-cdhX.X.X.jar
  httpclient-X.X.X.jar
  httpcore-X.X.X.jar
```

```
libfb303-X.X.X.jar
libthrift-X.X.X.jar
log4j-X.X.XX.jar
slf4j-api-X.X.X.jar
slf4j-logXjXX-X.X.X.jar
```

To enable JDBC support for Impala on the system where you run the JDBC application:

1. Download the JAR files listed above to each client machine.

   Note: For Maven users, see this sample github page for an example of the dependencies you could add to a `pom` file instead of downloading the individual JARs.

2. Store the JAR files in a location of your choosing, ideally a directory already referenced in your `CLASSPATH` setting. For example:
   - On Linux, you might use a location such as `/opt/jars/`.
   - On Windows, you might use a subdirectory underneath `C:\Program Files`.
3. To successfully load the Impala JDBC driver, client programs must be able to locate the associated JAR files. This often means setting the `CLASSPATH` for the client process to include the JARs. Consult the documentation for your JDBC client for more details on how to install new JDBC drivers, but some examples of how to set `CLASSPATH` variables include:
   - On Linux, if you extracted the JARs to `/opt/jars/`, you might issue the following command to prepend the JAR files path to an existing classpath:

```
export CLASSPATH=/opt/jars/*.jar:$CLASSPATH
```

   - On Windows, use the **System Properties** control panel item to modify the **Environment Variables** for your system. Modify the environment variables to include the path to which you extracted the files.

     Note: If the existing `CLASSPATH` on your client machine refers to some older version of the Hive JARs, ensure that the new JARs are the first ones listed. Either put the new JAR files earlier in the listings, or delete the other references to Hive JAR files.

Establishing JDBC Connections

The JDBC driver class depends on which driver you select.

Note: If your JDBC or ODBC application connects to Impala through a load balancer such as `haproxy`, be cautious about reusing the connections. If the load balancer has set up connection timeout values, either check the connection frequently so that it never sits idle longer than the load balancer timeout value, or check the connection validity before using it and create a new one if the connection has been closed.

Using the Cloudera JDBC Connector (recommended)

Depending on the level of the JDBC API your application is targeting, you can use the following fully-qualified class names (FQCNs):

- `com.cloudera.impala.jdbc41.Driver`
- `com.cloudera.impala.jdbc41.DataSource`
- `com.cloudera.impala.jdbc4.Driver`

- `com.cloudera.impala.jdbc4.DataSource`
- `com.cloudera.impala.jdbc3.Driver`
- `com.cloudera.impala.jdbc3.DataSource`

The connection string has the following format:

```
jdbc:impala://Host:Port[/Schema];Property1=Value;Property2=Value;...
```

The `port` value is typically 21050 for Impala.

For full details about the classes and the connection string (especially the property values available for the connection string), download the appropriate driver documentation for your platform from the Impala JDBC Connector download page.

Using the Hive JDBC Driver

For example, with the Hive JDBC driver, the class name is `org.apache.hive.jdbc.HiveDriver`. Once you have configured Impala to work with JDBC, you can establish connections between the two. To do so for a cluster that does not use Kerberos authentication, use a connection string of the form `jdbc:hive2://host:port/;auth=noSasl`. For example, you might use:

```
jdbc:hive2://myhost.example.com:21050/;auth=noSasl
```

To connect to an instance of Impala that requires Kerberos authentication, use a connection string of the form `jdbc:hive2://host:port/;principal=principal_name`. The principal must be the same user principal you used when starting Impala. For example, you might use:

```
jdbc:hive2://myhost.example.com:21050/;principal=impala/myhost.example.com@H2.EXAMPLE.COM
```

To connect to an instance of Impala that requires LDAP authentication, use a connection string of the form `jdbc:hive2://host:port/db_name;user=ldap_userid;password=ldap_password`. For example, you might use:

```
jdbc:hive2://myhost.example.com:21050/test_db;user=fred;password=xyz123
```



Note:

Prior to CDH 5.7 / Impala 2.5, the Hive JDBC driver did not support connections that use both Kerberos authentication and SSL encryption. If your cluster is running an older release that has this restriction, to use both of these security features with Impala through a JDBC application, use the Cloudera JDBC Connector as the JDBC driver.

Notes about JDBC and ODBC Interaction with Impala SQL Features

Most Impala SQL features work equivalently through the `impala-shell` interpreter of the JDBC or ODBC APIs. The following are some exceptions to keep in mind when switching between the interactive shell and applications using the APIs:

- **Complex type considerations:**
  - Queries involving the complex types (`ARRAY`, `STRUCT`, and `MAP`) require notation that might not be available in all levels of JDBC and ODBC drivers. If you have trouble querying such a table due to the driver level or inability to edit the queries used by the application, you can create a view that exposes a "flattened" version of the complex columns and point the application at the view. See Complex Types (CDH 5.5 or higher only) on page 139 for details.

- The complex types available in CDH 5.5 / Impala 2.3 and higher are supported by the JDBC `getColumns()` API. Both `MAP` and `ARRAY` are reported as the JDBC SQL Type `ARRAY`, because this is the closest matching Java SQL type. This behavior is consistent with Hive. `STRUCT` types are reported as the JDBC SQL Type `STRUCT`.

  To be consistent with Hive's behavior, the `TYPE_NAME` field is populated with the primitive type name for scalar types, and with the full `toSql()` for complex types. The resulting type names are somewhat inconsistent, because nested types are printed differently than top-level types. For example, the following list shows how `toSQL()` for Impala types are translated to `TYPE_NAME` values:

```
DECIMAL(10,10)          becomes  DECIMAL
CHAR(10)                becomes  CHAR
VARCHAR(10)             becomes  VARCHAR
ARRAY<DECIMAL(10,10)>   becomes  ARRAY<DECIMAL(10,10)>
ARRAY<CHAR(10)>         becomes  ARRAY<CHAR(10)>
ARRAY<VARCHAR(10)>      becomes  ARRAY<VARCHAR(10)>
```

Kudu Considerations for DML Statements

Currently, Impala `INSERT`, `UPDATE`, or other DML statements issued through the JDBC interface against a Kudu table do not return JDBC error codes for conditions such as duplicate primary key columns. Therefore, for applications that issue a high volume of DML statements, prefer to use the Kudu Java API directly rather than a JDBC application.

Troubleshooting Impala

Troubleshooting for Impala requires being able to diagnose and debug problems with performance, network connectivity, out-of-memory conditions, disk space usage, and crash or hang conditions in any of the Impala-related daemons.

Troubleshooting Impala SQL Syntax Issues

In general, if queries issued against Impala fail, you can try running these same queries against Hive.

- If a query fails against both Impala and Hive, it is likely that there is a problem with your query or other elements of your CDH environment:
  - Review the Language Reference to ensure your query is valid.
  - Check Impala Reserved Words on page 745 to see if any database, table, column, or other object names in your query conflict with Impala reserved words. Quote those names with backticks (``) if so.
  - Check Impala Built-In Functions on page 391 to confirm whether Impala supports all the built-in functions being used by your query, and whether argument and return types are the same as you expect.
  - Review the contents of the Impala logs for any information that may be useful in identifying the source of the problem.
- If a query fails against Impala but not Hive, it is likely that there is a problem with your Impala installation.

Troubleshooting Crashes Caused by Memory Resource Limit

Under very high concurrency, Impala could encounter a serious error due to usage of various operating system resources. Errors similar to the following may be caused by operating system resource exhaustion:

```
F0629 08:20:02.956413 29088 llvm-codegen.cc:111] LLVM hit fatal error: Unable to allocate
 section memory!
terminate called after throwing an instance of
'boost::exception_detail::clone_impl<boost::exception_detail::error_info_injector<boost::thread_resource_error>
 >'
```

The KRPC implementation in CDH 6.1 greatly reduces thread counts and the chances of hitting a resource limit in CDH 6.1 and higher.

If you still get an error similar to the above in Impala 3.0 and higher, try increasing the `max_map_count` OS virtual memory parameter. `max_map_count` defines the maximum number of memory map areas that a process can use. Configure each host running an `impalad` daemon with the command to increase `max_map_count` to 8 GB.

```
echo 8000000 > /proc/sys/vm/max_map_count
```

To make the above settings durable, refer to your OS documentation. For example, on RHEL 6.x:

1. Add the following line to `/etc/sysctl.conf`:

```
vm.max_map_count=8000000
```

2. Run the following command:

```
sysctl -p
```

Troubleshooting I/O Capacity Problems

Impala queries are typically I/O-intensive. If there is an I/O problem with storage devices, or with HDFS itself, Impala queries could show slow response times with no obvious cause on the Impala side. Slow I/O on even a single Impala daemon could result in an overall slowdown, because queries involving clauses such as `ORDER BY`, `GROUP BY`, or `JOIN` do not start returning results until all executor Impala daemons have finished their work.

To test whether the Linux I/O system itself is performing as expected, run Linux commands like the following on each host Impala daemon is running:

```
$ sudo sysctl -w vm.drop_caches=3 vm.drop_caches=0
vm.drop_caches = 3
vm.drop_caches = 0
$ sudo dd if=/dev/sda bs=1M of=/dev/null count=1k
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 5.60373 s, 192 MB/s
$ sudo dd if=/dev/sdb bs=1M of=/dev/null count=1k
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 5.51145 s, 195 MB/s
$ sudo dd if=/dev/sdc bs=1M of=/dev/null count=1k
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 5.58096 s, 192 MB/s
$ sudo dd if=/dev/sdd bs=1M of=/dev/null count=1k
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 5.43924 s, 197 MB/s
```

On modern hardware, a throughput rate of less than 100 MB/s typically indicates a performance issue with the storage device. Correct the hardware problem before continuing with Impala tuning or benchmarking.

Impala Troubleshooting Quick Reference

The following table lists common problems and potential solutions.

Symptom	Explanation	Recommendation
Impala takes a long time to start.	Impala instances with large numbers of tables, partitions, or data files take longer to start because the metadata for these objects is broadcast to all `impalad` nodes and cached.	Adjust timeout and synchronicity settings.
Joins fail to complete.	There may be insufficient memory. During a join, data from the second, third, and so on sets to be joined is loaded into memory. If Impala chooses an inefficient join order or join mechanism, the query could exceed the total memory available.	Start by gathering statistics with the `COMPUTE STATS` statement for each table involved in the join. Consider specifying the `[SHUFFLE]` hint so that data from the joined tables is split up between nodes rather than broadcast to each node. If tuning at the SQL level is not sufficient, add more memory to your system or join smaller data sets.
Queries return incorrect results.	Impala metadata may be outdated after changes are performed in Hive.	Where possible, use the appropriate Impala statement (`INSERT`, `LOAD DATA`, `CREATE TABLE`, `ALTER TABLE`, `COMPUTE STATS`, and so on) rather than switching back and forth between Impala and Hive. Impala automatically broadcasts the results of DDL and DML operations to all Impala nodes in the cluster, but does not

Symptom	Explanation	Recommendation
		automatically recognize when such changes are made through Hive. After inserting data, adding a partition, or other operation in Hive, refresh the metadata for the table as described in REFRESH Statement on page 291.
Queries are slow to return results.	Some `impalad` instances may not have started. Using a browser, connect to the host running the Impala state store. Connect using an address of the form `http://hostname:port/metrics`. **Note:** Replace *hostname* and *port* with the hostname and port of your Impala state store host machine and web server port. The default port is 25010. The number of `impalad` instances listed should match the expected number of `impalad` instances installed in the cluster. There should also be one `impalad` instance installed on each DataNode.	Ensure Impala is installed on all DataNodes. Start any `impalad` instances that are not running.
Queries are slow to return results.	Impala may not be configured to use native checksumming. Native checksumming uses machine-specific instructions to compute checksums over HDFS data very quickly. Review Impala logs. If you find instances of "`INFO util.NativeCodeLoader: Loaded the native-hadoop`" messages, native checksumming is not enabled.	Ensure Impala is configured to use native checksumming as described in Post-Installation Configuration for Impala on page 36.
Queries are slow to return results.	Impala may not be configured to use data locality tracking.	Test Impala for data locality tracking and make configuration changes as necessary. Information on this process can be found in Post-Installation Configuration for Impala on page 36.
Attempts to complete Impala tasks such as executing INSERT-SELECT actions fail. The Impala logs include notes that files could not be opened due to permission denied.	This can be the result of permissions issues. For example, you could use the Hive shell as the hive user to create a table. After creating this table, you could attempt to complete some action, such as an INSERT-SELECT on the table. Because the table was created using one user and the INSERT-SELECT is attempted by another, this action may fail due to permissions issues.	In general, ensure the Impala user has sufficient permissions. In the preceding example, ensure the Impala user has sufficient permissions to the table that the Hive user created.
Impala fails to start up, with the `impalad` logs referring	A large number of databases, tables, partitions, and so on can require metadata synchronization, particularly on startup, that takes longer than the default timeout for the statestore service.	Configure the statestore timeout value and possibly other settings related to the frequency of statestore updates and metadata loading. See

Symptom	Explanation	Recommendation
to errors connecting to the statestore service and attempts to re-register.		Increasing the Statestore Timeout on page 70 and Scalability Considerations for the Impala Statestore on page 606.

Impala Web User Interface for Debugging

Each of the Impala daemons (`impalad`, `statestored`, and `catalogd`) includes a built-in web server that displays diagnostic and status information.

impalad Web UI

The `impalad` Web UI includes information about configuration settings, running and completed queries, and associated performance and resource usage for queries. In particular, the **Details** link for each query displays alternative views of the query including a graphical representation of the plan, and the output of the `EXPLAIN`, `SUMMARY`, and `PROFILE` statements from `impala-shell`. Each host that runs the `impalad` daemon has its own instance of the Web UI, with details about those queries for which that host served as the coordinator. The `impalad` Web UI is primarily used for diagnosing query problems that can be traced to a particular node.

statestored Web UI

The `statestored` Web UI includes information about memory usage, configuration settings, and ongoing health checks performed by `statestored`. Because there is only a single instance of the `statestored` within any Impala cluster, you access the Web UI only on the particular host that serves as the Impala StateStore.

catalogd Web UI

The `catalogd` Web UI includes information about the databases, tables, and other objects managed by Impala, in addition to the resource usage and configuration settings of the `catalogd`. Because there is only a single instance of the `catalogd` within any Impala cluster, you access the Web UI only on the particular host that serves as the Impala Catalog Server.

Debug Web UI for impalad

To debug and troubleshoot an `impalad` using a web-based interface, open the URL `http://impala-server-hostname:25000/` in a browser. (For secure clusters, use the prefix `https://` instead of `http://`.)

Because each Impala node produces its own set of debug information, you should choose a specific node that you want to investigate an issue on.

Turning off the Web UI for impalad

To disable Web UI for an `impalad`:

1. Stop the `impalad`.
2. Restart the `impalad` with the `--enable_webserver=false` startup flag.

Main Page

The main `impalad` Web UI page at **/** lists the following information about the `impalad`:

- The version of the `impalad` daemon

  The **Version** section also contains other information, such as when Impala was built and what build flags were used.

- Process start time

- Hardware information
- OS information
- Process information
- CGroup information

Admission Controller Page

The **Admission Controller** `impalad` debug Web UI is at **/admission** page under the main `impalad` Web UI.

Use the **/admission** page to troubleshoot queued queries and the admission control.

The **admission** page provides the following information about each resource pool to which queries have been submitted at least once:

- Time since the `statestored` received the last update
- A warning if this `impalad` is considered disconnected from the `statestored` and thus the information on this page could be stale.
- Pool configuration
- Queued queries submitted to this coordinator, in the order of submission
- Running queries submitted to this coordinator
- Pool stats
  - Average of time in queue: An exponential moving average which represents the average time in queue over the last 10 to 12 queries. If a query is admitted immediately, the wait time of 0 is used in calculating this average wait time.
- Histogram of the distribution of peak memory used by queries admitted to the pool

  Use the histogram to figure out settings for the minimum and maximum query `MEM_LIMIT` ranges for this pool.

  The histogram displays data for all queries admitted to the pool, including the queries that finished, got canceled, or hit an error.

Click on the pool name to only display information relevant to that pool. You can then refresh the debug page to see only the information for that specific pool.

Click **Reset informational stats for all pools** to reset the stats that keep track of historical data, such as **Totals** stats, **Time in queue (exponential moving average)**, and the histogram.

The above information is also available as a JSON object from the following HTTP endpoint:

```
http://impala-server-hostname:port/admission?json
```

See Admission Control and Query Queuing on page 549 for the description of the properties in admission control.

Known Backends Page

The **Known backends** page of the `impalad` debug Web UI is at **/backends** under the main `impalad` Web UI.

This page lists the following info for each of the `impalad` nodes in the cluster. Because each `impalad` daemon knows about every other `impalad` daemon through the StateStore, this information should be the same regardless of which node you select.

- Address of the node: Host name and port
- KRPC address: The KRPC address of the `impalad`. Use this address when you issue the `SHUTDOWN` command for this `impalad`.
- Whether acting as a coordinator
- Whether acting as an executor
- Quiescing status: Specify whether the graceful shutdown process has been initiated on this `impalad`.
- Memory limit for admission: The amount of memory that can be admitted to this backend by the admission controller.

- Memory reserved: The amount of memory reserved by queries that are active, either currently executing or finished but not yet closed, on this backend.

  The memory reserved for a query that is currently executing is its memory limit, if set. Otherwise, if the query has no limit or if the query finished executing, the current consumption is used.

- Memory of the queries admitted to this coordinator: The memory submitted to this particular host by the queries admitted by this coordinator.

Catalog Page

The **Catalog** page of the `impalad` debug Web UI is at **/catalog** under the main `impalad` Web UI.

This page displays a list of databases and associated tables recognized by this instance of `impalad`. You can use this page to locate which database a table is in, check the exact spelling of a database or table name, look for identical table names in multiple databases. The primary debugging use case would be to check if an `impalad` instance has knowledge of a particular table that someone expects to be in a particular database.

Hadoop Configuration

The **Hadoop Configuration** page of the `impalad` debug Web UI is at **/hadoop-varz** under the main `impalad` Web UI.

This page displays the Hadoop common configurations that Impala is running with.

JMX

The **JMX** page of the `impalad` debug Web UI is at **/jmx** under the main `impalad` Web UI.

This page displays monitoring information about various JVM subsystems, such as memory pools, thread management, runtime. etc.

Java Log Level

The **Change log level** page of the `impalad` debug Web UI is at **/log_level** under the main `impalad` Web UI.

This page displays the current Java and backend log levels, and it allows you to change the log levels dynamically without having to restart the `impalad`.

Logs Page

The **INFO logs** page of the `impalad` debug Web UI is at **/logs** under the main `impalad` Web UI.

This page shows the last portion of the `impalad.INFO` log file, including the info, warning, and error logs for the `impalad`. You can see the details of the most recent operations, whether the operations succeeded or encountered errors. This page provides one central place for the log files and saves you from looking around the filesystem for the log files, which could be in different locations on clusters that use cluster management software.

Memz Page

The **Memory Usage** page of the `impalad` debug Web UI is at **/memz** under the main `impalad` Web UI.

This page displays the summary and detailed information about memory usage by the `impalad`.

Metrics Page

The **Metrics** page of the `impalad` debug Web UI is at **/metrics** under the main `impalad` Web UI.

This page displays the current set of metrics, counters and flags representing various aspects of `impalad` internal operations.

Queries Page

The **Queries** page of the `impalad` debug Web UI is at **/queries** under the main `impalad` Web UI.

This page lists:

- Currently running queries

- Queries that have completed their execution, but have not been closed yet
- Completed queries whose details still reside in memory

The queries are listed in reverse chronological order, with the most recent at the top. You can control the amount of memory devoted to completed queries by specifying the `----query_log_size` startup option for `impalad`.

This page provides:

- How many SQL statements are failing (`State` value of `EXCEPTION`)
- How large the result sets are (`# rows fetched`)
- How long each statement took (`Start Time` and `End Time`)

Click the **Details** link for a query to display the detailed performance characteristics of that query, such as the profile output.

On the query detail page, in the **Profile** tab, you have options to export the query profile output to the Thrif, text, or Json format.

The **Queries** page also includes the **Query Locations** section that lists the number of running queries with fragments on this host.

RPC Services Page

The **RPC durations** page of the `impalad` debug Web UI is at **/rpcz** under the main `impalad` Web UI.

This page displays information, such as the duration, about the RPC communications of this `impalad` with other Impala daemons.

Sessions Page

The **Sessions** page of the `impalad` debug Web UI is at **/session** under the main `impalad` Web UI.

This page displays information about the sessions currently connected to this `impalad` instance. For example, sessions could include connections from the `impala-shell` command, JDBC or ODBC applications, or the Impala Query UI in the Hue web interface.

Threadz Page

The **Threads** page of the `impalad` debug Web UI is at **/threadz** under the main `impalad` Web UI.

This page displays information about the threads used by this instance of `impalad`, and it shows which categories they are grouped into. Making use of this information requires substantial knowledge about Impala internals.

Varz Page

The **Varz** page of the `impalad` debug Web UI is at **/varz** under the main `impalad` Web UI.

This page shows the configuration settings in effect when this instance of `impalad` communicates with other Hadoop components such as HDFS and YARN. These settings are collected from a set of configuration files.

The bottom of this page also lists all the command-line settings in effect for this instance of `impalad`. See Modifying Impala Startup Options for information about modifying these values.

Prometheus Metrics Page

At **/metrics_prometheus** under the main `impalad` Web UI, the metrics are generated in Prometheus exposition format that Prometheus can consume for event monitoring and alerting. The **/metrics_prometheus** is not shown in the Web UI list of pages.

Debug Web UI for statestored

To debug and troubleshoot the `statestored` daemon using a web-based interface, open the URL `http://impala-server-hostname:25010/` in a browser. (For secure clusters, use the prefix `https://` instead of `http://`.)

Turning off the Web UI for statestored

To disable Web UI for the `statestored`:

1. Stop the `statestored`.
2. Restart the `statestored` with the `--enable_webserver=false` startup flag.

Main Page

The main `statestored` Web UI page at **/** lists the following information about the `statestored`:

- The version of the `statestored` daemon
- Process start time
- Hardware information
- OS information
- Process information
- CGroup information

Logs Page

The **INFO logs** page of the debug Web UI is at **/logs** under the main `statestored` Web UI.

This page shows the last portion of the `statestored.INFO` log file, including the info, warning, and error logs for the `statestored`. You can refer here to see the details of the most recent operations, whether the operations succeeded or encountered errors. This page provides one central place for the log files and saves you from looking around the filesystem for the log files, which could be in different locations on clusters that use cluster management software.

Memz Page

The **Memory Usage** page of the debug Web UI is at **/memz** under the main `statestored` Web UI.

This page displays summary and detailed information about memory usage by the `statestored`. You can see the memory limit in effect for the node, and how much of that memory Impala is currently using.

Metrics Page

The **Metrics** page of the debug Web UI is at **/metrics** under the main `statestored` Web UI.

This page displays the current set of metrics: counters and flags representing various aspects of `statestored` internal operation.

RPC Services Page

The **RPC durations** page of the `statestored` debug Web UI is at **/rpcz** under the main `statestored` Web UI.

This page displays information, such as the durations, about the RPC communications of this `statestored` with other Impala daemons.

Subscribers Page

The **Subscribers** page of the debug Web UI is at **/subscribers** under the main `statestored` Web UI.

This page displays information about the other Impala daemons that have registered with the `statestored` to receive and send updates.

Threadz Page

The **Threads** page of the debug Web UI is at **/threadz** under the main `statestored` Web UI.

This page displays information about the threads used by this instance of `statestored`, and shows which categories they are grouped into. Making use of this information requires substantial knowledge about Impala internals.

Topics Page

The **Topics** page of the debug Web UI is at **/topics** under the main `statestored` Web UI.

This page displays information about the topics to which the other Impala daemons have registered to receive updates.

Varz Page

The **Varz** page of the debug Web UI is at **/varz** under the main `statestored` Web UI.

This page shows the configuration settings in effect when this instance of `statestored` communicates with other Hadoop components such as HDFS and YARN. These settings are collected from a set of configuration files.

The bottom of this page also lists all the command-line settings in effect for this instance of `statestored`. See Modifying Impala Startup Options for information about modifying these values.

Prometheus Metrics Page

At **/metrics_prometheus** under the main `statestored` Web UI, the metrics are generated in Prometheus exposition format that Prometheus can consumes for event monitoring and alerting. The **/metrics_prometheus** is not shown in the Web UI list of pages.

Debug Web UI for catalogd

The main page of the debug Web UI is at `http://impala-server-hostname:25020/` (non-secure cluster) or `https://impala-server-hostname:25020/` (secure cluster).

Turning off the Web UI for catalogd

To disable Web UI for the `catalogd`:

1. Stop the `catalogd`.
2. Restart the `catalogd` with the `--enable_webserver=false` startup flag.

Main Page

The main `catalogd` Web UI page at **/** lists the following information about the `catalogd`:

- The version of the `catalogd` daemon
- Process start time
- Hardware information
- OS information
- Process information
- CGroup information

Catalog Page

The **Catalog** page of the debug Web UI is at **/catalog** under the main `catalogd` Web UI.

This page displays a list of databases and associated tables recognized by this instance of `catalogd`. You can use this page to locate which database a table is in, check the exact spelling of a database or table name, look for identical table names in multiple databases. The catalog information is represented as the underlying Thrift data structures.

JMX

The **JMX** page of the `catalogd` debug Web UI is at **/jmx** under the main `catalogd` Web UI.

This page displays monitoring information about various JVM subsystems, such as memory pools, thread management, runtime. etc.

Java Log Level

The **Change log level** page of the `catalogd` debug Web UI is at **/log_level** under the main `catalogd` Web UI.

The page displays the current Java and backend log levels and allows you to change the log levels dynamically without having to restart the `catalogd`

Logs Page

The **INFO logs** page of the debug Web UI is at **/logs** under the main `catalogd` Web UI.

This page shows the last portion of the `catalogd.INFO` log file, including the info, warning, and error logs for the `catalogd` daemon. You can refer here to see the details of the most recent operations, whether the operations succeeded or encountered errors. This page provides one central place for the log files and saves you from looking around the filesystem for the log files, which could be in different locations on clusters that use cluster management software.

Memz Page

The **Memory Usage** page of the debug Web UI is at **/memz** under the main `catalogd` Web UI.

This page displays summary and detailed information about memory usage by the `catalogd`. You can see the memory limit in effect for the node, and how much of that memory Impala is currently using.

Metrics Page

The **Metrics** page of the debug Web UI is at **/metrics** under the main `catalogd` Web UI.

This page displays the current set of metrics: counters and flags representing various aspects of `catalogd` internal operation.

RPC Services Page

The **RPC durations** page of the `catalogd` debug Web UI is at **/rpcz** under the main `catalogd` Web UI.

This page displays information, such as the durations, about the RPC communications of this `catalogd` with other Impala daemons.

Threadz Page

The **Threads** page of the debug Web UI is at **/threadz** under the main `catalogd` Web UI.

This page displays information about the threads used by this instance of `catalogd`, and shows which categories they are grouped into. Making use of this information requires substantial knowledge about Impala internals.

Varz Page

The **Varz** page of the debug Web UI is at **/varz** under the main `catalogd` Web UI.

This page shows the configuration settings in effect when this instance of `catalogd` communicates with other Hadoop components such as HDFS and YARN. These settings are collected from a set of configuration files.

The bottom of this page also lists all the command-line settings in effect for this instance of `catalogd`. See Modifying Impala Startup Options for information about modifying these values.

Prometheus Metrics Page

At **/metrics_prometheus** under the main `impalad` Web UI, the metrics are generated in Prometheus exposition format that Prometheus can consume for event monitoring and alerting. The **/metrics_prometheus** is not shown in the Web UI list of pages.

Breakpad Minidumps for Impala (CDH 5.8 or higher only)

The breakpad project is an open-source framework for crash reporting. In CDH 5.8 / Impala 2.6 and higher, Impala can use `breakpad` to record stack information and register values when any of the Impala-related daemons crash due to an error such as `SIGSEGV` or unhandled exceptions. The dump files are much smaller than traditional core dump files. The dump mechanism itself uses very little memory, which improves reliability if the crash occurs while the system is low on memory.



Important: Because of the internal mechanisms involving Impala memory allocation and Linux signalling for out-of-memory (OOM) errors, if an Impala-related daemon experiences a crash due to an OOM condition, it does *not* generate a minidump for that error.

Enabling or Disabling Minidump Generation

By default, a minidump file is generated when an Impala-related daemon crashes.

To turn off generation of the minidump files, use one of the following options:

- Set the `--enable_minidumps` configuration setting to `false`. Restart the corresponding services or daemons.
- Set the `--minidump_path` configuration setting to an empty string. Restart the corresponding services or daemons.

In CDH 5.9 / Impala 2.7 and higher, you can send a `SIGUSR1` signal to any Impala-related daemon to write a Breakpad minidump. For advanced troubleshooting, you can now produce a minidump without triggering a crash.

Specifying the Location for Minidump Files

By default, all minidump files are written to the following location on the host where a crash occurs:

- Clusters managed by Cloudera Manager: `/var/log/impala-minidumps/`*`daemon_name`*
- Clusters not managed by Cloudera Manager: *`impala_log_dir`*`/`*`daemon_name`*`/minidumps/`*`daemon_name`*

The minidump files for `impalad`, `catalogd`, and `statestored` are each written to a separate directory.

To specify a different location, set the **minidump_path** configuration setting of one or more Impala-related daemons, and restart the corresponding services or daemons.

If you specify a relative path for this setting, the value is interpreted relative to the default **minidump_path** directory.

Controlling the Number of Minidump Files

Like any files used for logging or troubleshooting, consider limiting the number of minidump files, or removing unneeded ones, depending on the amount of free storage space on the hosts in the cluster.

Because the minidump files are only used for problem resolution, you can remove any such files that are not needed to debug current issues.

To control how many minidump files Impala keeps around at any one time, set the **max_minidumps** configuration setting for of one or more Impala-related daemon, and restart the corresponding services or daemons. The default for this setting is 9. A zero or negative value is interpreted as “unlimited”.

Detecting Crash Events

You can see in the Impala log files or in the Cloudera Manager charts for Impala when crash events occur that generate minidump files. Because each restart begins a new log file, the “crashed” message is always at or near the bottom of the log file. (There might be another later message if core dumps are also enabled.)

Using the Minidump Files for Problem Resolution

Typically, you provide minidump files to Cloudera Support as part of problem resolution, in the same way that you might provide a core dump. The **Send Diagnostic Data** under the **Support** menu in Cloudera Manager guides you through the process of selecting a time period and volume of diagnostic data, then collects the data from all hosts and transmits the relevant information for you.

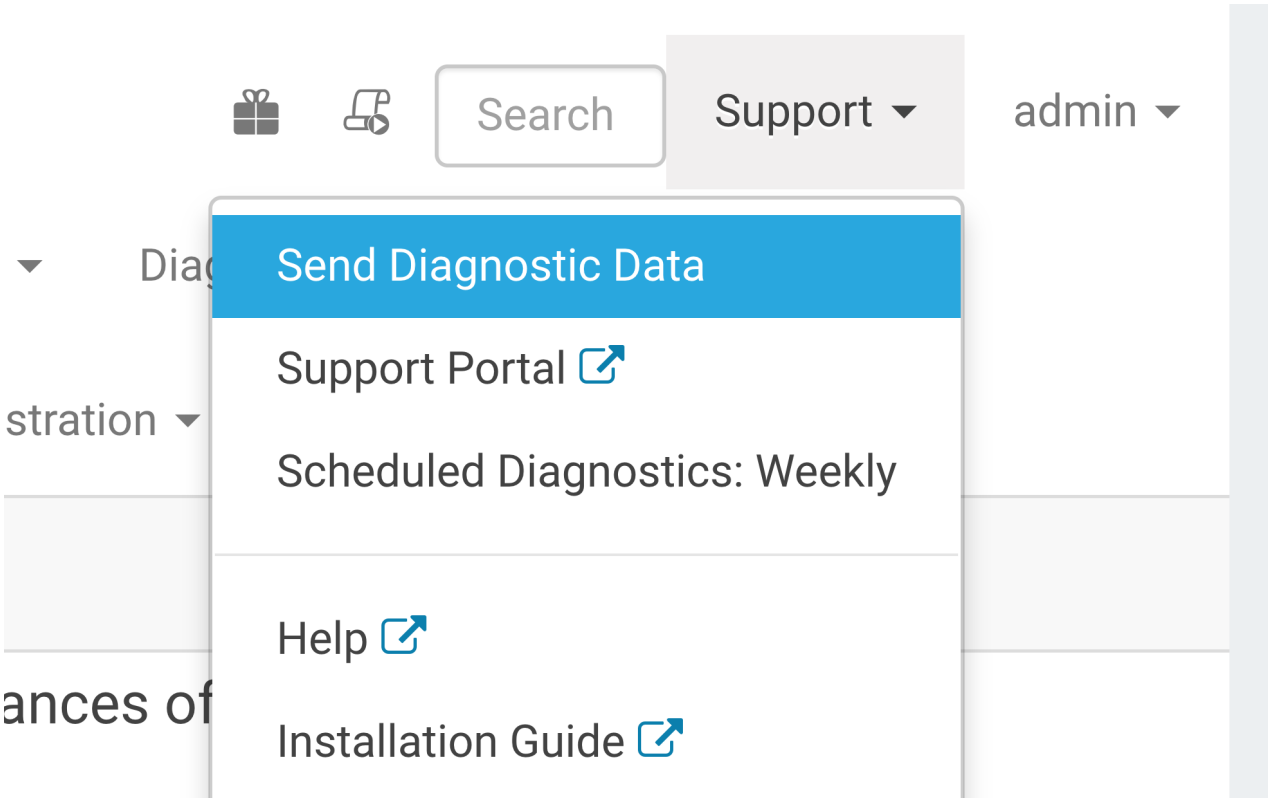


Figure 6: Send Diagnostic Data choice under Support menu

You might get additional instructions from Cloudera Support about collecting minidumps to better isolate a specific problem. Because the information in the minidump files is limited to stack traces and register contents, the possibility of including sensitive information is much lower than with core dump files. If any sensitive information is included in the minidump, Cloudera Support preserves the confidentiality of that information.

Demonstration of Breakpad Feature

The following example uses the command `kill -11` to simulate a SIGSEGV crash for an `impalad` process on a single DataNode, then examines the relevant log files and minidump file.

First, as root on a worker node, we kill the `impalad` process with a SIGSEGV error. The original process ID was 23114. (Cloudera Manager restarts the process with a new pid, as shown by the second `ps` command.)

```
# ps ax | grep impalad
23114 ?        Sl     0:18
/opt/cloudera/parcels/<parcel_version>/lib/impala/sbin-retail/impalad
--flagfile=/var/run/cloudera-scm-agent/process/114-impala-IMPALAD/impala-conf/impalad_flags
31259 pts/0    S+     0:00 grep impalad
#
# kill -11 23114
#
# ps ax | grep impalad
31374 ?        Rl     0:04
/opt/cloudera/parcels/<parcel_version>/lib/impala/sbin-retail/impalad
--flagfile=/var/run/cloudera-scm-agent/process/114-impala-IMPALAD/impala-conf/impalad_flags
31475 pts/0    S+     0:00 grep impalad
```

We locate the log directory underneath /var/log. There is a .INFO, .WARNING, and .ERROR log file for the 23114 process ID. The minidump message is written to the .INFO file and the .ERROR file, but not the .WARNING file. In this case, a large core file was also produced.

```
# cd /var/log/impalad
# ls -la | grep 23114
-rw-------   1 impala impala 3539079168 Jun 23 15:20 core.23114
-rw-r--r--   1 impala impala      99057 Jun 23 15:20 hs_err_pid23114.log
-rw-r--r--   1 impala impala        351 Jun 23 15:20
impalad.worker_node_123.impala.log.ERROR.20160623-140343.23114
-rw-r--r--   1 impala impala      29101 Jun 23 15:20
impalad.worker_node_123.impala.log.INFO.20160623-140343.23114
-rw-r--r--   1 impala impala        228 Jun 23 14:03
impalad.worker_node_123.impala.log.WARNING.20160623-140343.23114
```

The .INFO log includes the location of the minidump file, followed by a report of a core dump. With the breakpad minidump feature enabled, now we might disable core dumps or keep fewer of them around.

```
# cat impalad.worker_node_123.impala.log.INFO.20160623-140343.23114
...
Wrote minidump to
/var/log/impala-minidumps/impalad/0980da2d-a905-01e1-25ff883a-04ee027a.dmp
#
# A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00000030c0e0b68a, pid=23114, tid=139869541455968
#
# JRE version: Java(TM) SE Runtime Environment (7.0_67-b01) (build 1.7.0_67-b01)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (24.65-b04 mixed mode linux-amd64 compressed
 oops)
# Problematic frame:
# C  [libpthread.so.0+0xb68a]  pthread_cond_wait+0xca
#
# Core dump written. Default location: /var/log/impalad/core or core.23114
#
# An error report file with more information is saved as:
# /var/log/impalad/hs_err_pid23114.log
#
# If you would like to submit a bug report, please visit:
#   http://bugreport.sun.com/bugreport/crash.jsp
# The crash happened outside the Java Virtual Machine in native code.
# See problematic frame for where to report the bug.
...

# cat impalad.worker_node_123.impala.log.ERROR.20160623-140343.23114

Log file created at: 2016/06/23 14:03:43
Running on machine:.worker_node_123
Log line format: [IWEF]mmdd hh:mm:ss.uuuuuu threadid file:line] msg
E0623 14:03:43.911002 23114 logging.cc:118] stderr will be logged to this file.
Wrote minidump to
/var/log/impala-minidumps/impalad/0980da2d-a905-01e1-25ff883a-04ee027a.dmp
```

The resulting minidump file is much smaller than the corresponding core file, making it much easier to supply diagnostic information to Cloudera Support. The transmission process for the minidump files is automated through Cloudera Manager.

```
# pwd
/var/log/impalad
# cd ../impala-minidumps/impalad
# ls
0980da2d-a905-01e1-25ff883a-04ee027a.dmp
# du -kh *
2.4M  0980da2d-a905-01e1-25ff883a-04ee027a.dmp
```

Ports Used by Impala

Impala uses the TCP ports listed in the following table. Before deploying Impala, ensure these ports are open on each system.

Scope / Role	Setting in Cloudera Manager	Default Port	Comment
Impala Daemon	Impala Daemon Frontend Port	21000	Used to transmit commands and receive results by `impala-shell` and version 1.2 of the Cloudera ODBC driver.
Impala Daemon	Impala Daemon Frontend Port	21050	Used to transmit commands and receive results by applications, such as Business Intelligence tools, using JDBC, the Beeswax query editor in Hue, and version 2.0 or higher of the Cloudera ODBC driver.
Impala Daemon	Impala Daemon Backend Port	22000	Internal use only. Impala daemons use this port to communicate with each other.
Impala Daemon	StateStoreSubscriber Service Port	23000	Internal use only. Impala daemons listen on this port for updates from the statestore daemon.
Catalog Daemon	StateStoreSubscriber Service Port	23020	Internal use only. The catalog daemon listens on this port for updates from the statestore daemon.
Impala Daemon	Impala Daemon HTTP Server Port	25000	Impala web interface for administrators to monitor and troubleshoot.
Impala StateStore Daemon	StateStore HTTP Server Port	25010	StateStore web interface for administrators to monitor and troubleshoot.
Impala Catalog Daemon	Catalog HTTP Server Port	25020	Catalog service web interface for administrators to monitor and troubleshoot. New in Impala 1.2 and higher.
Impala StateStore Daemon	StateStore Service Port	24000	Internal use only. The statestore daemon listens on this port for registration/unregistration requests.
Impala Catalog Daemon	Catalog Service Port	26000	Internal use only. The catalog service uses this port to communicate with the Impala daemons. New in Impala 1.2 and higher.

Scope / Role	Setting in Cloudera Manager	Default Port	Comment
Impala Daemon	KRPC Port	27000	Internal use only. Impala daemons use this port for KRPC based communication with each other.

Impala Reserved Words

This topic lists the reserved words in Impala.

A reserved word is one that cannot be used directly as an identifier. If you need to use it as an identifier, you must quote it with backticks. For example:

- `CREATE TABLE select (x INT)`: fails
- `CREATE TABLE \`select\` (x INT)`: succeeds

Because different database systems have different sets of reserved words, and the reserved words change from release to release, carefully consider database, table, and column names to ensure maximum compatibility between products and versions.

Also consider whether your object names are the same as any Hive keywords, and rename or quote any that conflict as you might switch between Impala and Hive when doing analytics and ETL. Consult the list of Hive keywords.

To future-proof your code, you should avoid additional words in case they become reserved words if Impala adds features in later releases. This kind of planning can also help to avoid name conflicts in case you port SQL from other systems that have different sets of reserved words. The Future Keyword column in the table below indicates those additional words that you should avoid for table, column, or other object names, even though they are not currently reserved by Impala.

The following is a summary of the process for deciding whether a particular SQL 2016 word is to be reserved in Impala.

- By default, Impala targets to have the same list of reserved words as SQL 2016.
- At the same time, to be compatible with earlier versions of Impala and to avoid breaking existing tables/workloads, Impala built-in function names are removed from the reserved words list, e.g. COUNT, AVG, as Impala generally does not need to reserve the names of built-in functions for parsing to work.
- For those remaining SQL 2016 reserved words, if a word is likely to be in-use by users of older Impala versions and if there is a low chance of Impala needing to reserve that word in the future, then the word is not reserved.
- Otherwise, the word is reserved in Impala.

List of Current Reserved Words

Keyword	Reserved in SQL:2016	Reserved in Impala 2.12 and lower	Reserved in CDH 6.0 / Impala 3.0 and higher	Future Keyword
abs	X			
acos	X			
add		X	X	
aggregate		X	X	
all	X	X	X	
allocate	X		X	
alter	X	X	X	
analytic		X	X	
and	X	X	X	
anti		X	X	

any	X		X	
api_version			X	
are	X		X	
array	X	X	X	
array_agg	X		X	
array_max_cardinality	X		X	
as	X	X	X	
asc		X	X	
asensitive	X		X	
asin	X			
asymmetric	X		X	
at	X		X	
atan	X			
atomic	X		X	
authorization	X		X	
avg	X			
avro		X	X	
backup				X
begin	X			X
begin_frame	X		X	
begin_partition	X		X	
between	X	X	X	
bigint	X	X	X	
binary	X	X	X	
blob	X		X	
block_size			X	
boolean	X	X	X	
both	X		X	
break				X
browse				X
bulk				X
by	X	X	X	
cache				
cached		X	X	
call	X			
called	X		X	

cardinality	X		X	
cascade		X	X	
cascaded	X		X	
case	X	X	X	
cast	X	X	X	
ceil	X			
ceiling	X			
change		X	X	
char	X	X	X	
char_length	X			
character	X		X	
character_length	X			
check	X			X
checkpoint				X
class		X	X	
classifier	X			
clob	X		X	
close	X			X
close_fn			X	
clustered				X
coalesce	X			X
collate	X		X	
collect	X		X	
column	X	X	X	
columns		X	X	
comment		X	X	
commit	X		X	
compression		X	X	
compute		X	X	
condition	X		X	
conf				
connect	X		X	
constraint	X		X	
contains	X		X	
continue				X
convert	X		X	

copy	X		X	
corr	X		X	
corresponding	X		X	
cos	X			
cosh	X			
count	X			
covar_pop	X		X	
covar_samp	X		X	
create	X	X	X	
cross	X	X	X	
cube	X		X	
cume_dist	X			
current	X	X	X	
current_catalog	X			
current_date	X		X	
current_default_transform_group	X		X	
current_path	X		X	
current_role	X		X	
current_row	X		X	
current_schema	X		X	
current_time	X		X	
current_timestamp	X			X
current_transform_group_for_type	X		X	
current_user	X			X
cursor	X		X	
cycle	X		X	
data		X	X	
database		X	X	
databases		X	X	
date	X	X	X	
datetime		X	X	
day	X			
dayofweek				
dbcc				X
deallocate	X		X	
dec	X		X	

decfloat	X		X	
decimal	X	X	X	
declare	X		X	
default	X	X	X	
define	X		X	
delete	X	X	X	
delimited		X	X	
dense_rank	X			
deny				X
deref	X		X	
desc		X	X	
describe	X	X	X	
deterministic	X		X	
disconnect	X		X	
disk				X
distinct	X	X	X	
distributed				X
div		X	X	
double	X	X	X	
drop	X	X	X	
dump				X
dynamic	X		X	
each	X		X	
element	X		X	
else	X	X	X	
empty	X		X	
encoding		X	X	
end	X	X	X	
end-exec	X		X	
end_frame	X		X	
end_partition	X		X	
equals	X		X	
errlvl				X
escape	X		X	
escaped		X	X	
every	X		X	

except	X		X	
exchange				
exec	X		X	
execute	X		X	
exists	X	X	X	
exit				X
exp	X			
explain		X	X	
extended		X	X	
external	X	X	X	
extract	X			
false	X	X	X	
fetch	X		X	
fields		X	X	
file				X
filefactor				X
fileformat		X	X	
files		X	X	
filter	X		X	
finalize_fn			X	
first		X	X	
first_value	X			
float	X	X	X	
floor	X			
following		X	X	
for	X	X	X	
foreign	X		X	
format		X	X	
formatted		X	X	
frame_row	X		X	
free	X		X	
freetext				X
from	X	X	X	
full	X	X	X	
function	X	X	X	
functions		X	X	

fusion	X		X	
get	X		X	
global	X		X	
goto				X
grant	X	X	X	
group	X	X	X	
grouping	X		X	
groups	X		X	
hash		X	X	
having	X	X	X	
hold	X		X	
holdlock				X
hour	X			
identity	X			X
if		X	X	
ignore		X	X	
ilike		X	X	
import				
in	X	X	X	
incremental		X	X	
index				X
indicator	X		X	
init_fn			X	
initial	X		X	
inner	X	X	X	
inout	X		X	
inpath		X	X	
insensitive	X		X	
insert	X	X	X	
int	X	X	X	
integer	X	X	X	
intermediate		X	X	
intersect	X		X	
intersection	X		X	
interval	X	X	X	
into	X	X	X	

invalidate		X	X	
iregexp		X	X	
is	X	X	X	
join	X	X	X	
json_array	X		X	
json_arrayagg	X		X	
json_exists	X		X	
json_object	X		X	
json_objectagg	X		X	
json_query	X		X	
json_table	X		X	
json_table_primitive	X		X	
json_value	X		X	
key				X
kill				X
kudu		X	X	
lag	X			
language	X			
large	X		X	
last		X	X	
last_value	X			
lateral	X		X	
lead	X			
leading	X		X	
left	X	X	X	
less				
like	X	X	X	
like_regex	X		X	
limit		X	X	
lineno				X
lines		X	X	
listagg	X		X	
ln	X			
load		X	X	
local	X		X	
localtime	X			

localtimestamp	X		X	
location		X	X	
log	X			
log10	X		X	
lower	X			
macro				
map		X	X	
match	X		X	
match_number	X		X	
match_recognize	X		X	
matches	X		X	
max	X			
member	X			
merge	X		X	
merge_fn			X	
metadata		X	X	
method	X		X	
min	X			
minute	X			
mod	X			
modifies	X		X	
module	X			
month	X			
more				
multiset	X		X	
national	X		X	
natural	X		X	
nchar	X		X	
nclob	X		X	
new	X			
no	X		X	
nocheck				X
nonclustered				X
none	X		X	
normalize	X		X	
not	X	X	X	

nth_value	X		X	
ntile	X			
null	X	X	X	
nullif	X			X
nulls		X	X	
numeric	X		X	
occurrences_regex	X		X	
octet_length	X		X	
of	X		X	
off				X
offset	X	X	X	
offsets				X
old	X			
omit	X		X	
on	X	X	X	
one	X		X	
only	X		X	
open	X			X
option				X
or	X	X	X	
order	X	X	X	
out	X		X	
outer	X	X	X	
over	X	X	X	
overlaps	X		X	
overlay	X		X	
overwrite		X	X	
parameter	X			
parquet		X	X	
parquetfile		X	X	
partialscan				
partition	X	X	X	
partitioned		X	X	
partitions		X	X	
pattern	X		X	
per	X		X	

percent	X		X	
percent_rank	X			
percentile_cont	X		X	
percentile_disc	X		X	
period	X			
pivot				X
plan				X
portion	X		X	
position	X		X	
position_regex	X		X	
power	X			
precedes	X		X	
preceding		X	X	
precision	X			X
prepare	X		X	
prepare_fn			X	
preserve				
primary	X	X	X	
print				X
proc				X
procedure	X		X	
produced		X	X	
ptf	X		X	
public				X
purge		X	X	
raiseerror				X
range	X	X	X	
rank	X			
rcfile		X	X	
read				X
reads	X		X	
readtext				X
real	X	X	X	
reconfigure				X
recover		X	X	
recursive	X		X	

reduce				
ref	X		X	
references	X		X	
referencing	X		X	
refresh		X	X	
regexp		X	X	
regr_avgx	X		X	
regr_avgy	X		X	
regr_count	X		X	
regr_intercept	X		X	
regr_r2	X		X	
regr_slope	X		X	
regr_sxx	X		X	
regr_sxy	X		X	
regr_syy	X		X	
release	X		X	
rename		X	X	
repeatable		X	X	
replace		X	X	
replication		X	X	
restore				X
restrict		X	X	
result	X			
return	X			X
returns	X	X	X	
revert				X
revoke	X	X	X	
right	X	X	X	
rlike		X	X	
role		X	X	
roles		X	X	
rollback	X		X	
rollup	X		X	
row	X	X	X	
row_number	X			
rowcount				X

rows	X	X	X	
rule				X
running	X		X	
save				X
savepoint	X		X	
schema		X	X	
schemas		X	X	
scope	X		X	
scroll	X		X	
search	X		X	
second	X			
securityaudit				X
seek	X		X	
select	X	X	X	
semi		X	X	
sensitive	X		X	
sequencefile		X	X	
serdeproperties		X	X	
serialize_fn			X	
session_user	X			X
set	X	X	X	
setuser				X
show	X	X	X	
shutdown				X
similar	X		X	
sin	X			
sinh	X			
skip	X		X	
smallint	X	X	X	
some	X		X	
sort		X	X	
specific	X		X	
specifictype	X		X	
sql	X			
sqlexception	X		X	
sqlstate	X		X	

sqlwarning	X		X	
sqrt	X			
start	X			
static	X		X	
statistics				X
stats		X	X	
stddev_pop	X			
stddev_samp	X			
stored		X	X	
straight_join			X	
string		X	X	
struct		X	X	
submultiset	X		X	
subset	X		X	
substring	X			
substring_regex	X		X	
succeeds	X		X	
sum	X			
symbol		X	X	
symmetric	X		X	
system	X			
system_time	X		X	
system_user	X		X	
table	X	X	X	
tables		X	X	
tablesample	X	X	X	
tan	X			
tanh	X			
tblproperties		X	X	
terminated		X	X	
textfile		X	X	
textsize				X
then	X	X	X	
time	X			
timestamp	X	X	X	
timezone_hour	X		X	

timezone_minute	X		X	
tinyint		X	X	
to	X	X	X	
top				X
trailing	X		X	
tran				X
transform				
translate	X			
translate_regex	X		X	
translation	X		X	
treat	X		X	
trigger	X		X	
trim	X			
trim_array	X		X	
true	X	X	X	
truncate	X	X	X	
try_convert				X
uescape	X		X	
unbounded		X	X	
uncached		X	X	
union	X	X	X	
unique	X		X	
uniquejoin				
unknown	X	X	X	
unnest	X		X	
unpivot				X
update	X	X	X	
update_fn			X	
updatetext				X
upper	X			
upsert		X	X	
use		X	X	
user	X			X
using	X	X	X	
utc_tmestamp				
value	X			

value_of	X		X	
values	X	X	X	
var_pop	X			
var_samp	X			
varbinary	X		X	
varchar	X	X	X	
varying	X		X	
versioning	X		X	
view		X	X	
views				
waitfor				X
when	X	X	X	
whenever	X		X	
where	X	X	X	
while				X
width_bucket	X		X	
window	X		X	
with	X	X	X	
within	X		X	
without	X		X	
writetext				X
year	X			

Impala Frequently Asked Questions

Here are the categories of frequently asked questions for Apache Impala, the interactive SQL engine for CDH.

Transition to Apache Governance

Does "Apache Impala (incubating)" mean Impala is not production-ready?

No. The “(incubating)” label was only applied to the Apache Impala project while it was transitioning to governance by the Apache Software Foundation. Impala graduated to a top-level Apache project on November 15, 2017.

Impala has always been Apache-licensed. The software itself is the same production-ready and battle-tested analytic database that has been supported by Cloudera since Impala 1.0 in 2013.

Why does the Impala version string in CDH 5.10 say Impala 2.7, while the docs refer to Impala 2.8?

The version of Impala that is included in CDH 5.10 is Impala 2.7 plus almost all the patches that went into Impala 2.8. CDH 5.10 was released very shortly after Apache Impala 2.8, and the version string was not updated in the CDH packaging. To accurately relate the CDH 5.10 feature set to the corresponding level of Apache Impala, the documentation refers to Impala 2.8 as the minimum Impala version number for features such as the `MT_DOP` query option and full integration of Impala SQL syntax with Apache Kudu.

The full list of relevant commits for the Impala included with CDH 5.10.0, and the upstream Apache Impala project, are:

- CDH 5.10: https://github.com/cloudera/Impala/commits/cdh5-2.7.0_5.10.0
- Apache Impala 2.8: https://github.com/apache/incubator-impala/commits/branch-2.8.0
-

Because the Cloudera policy is to keep version numbering consistent across CDH maintenance releases, the Impala version string in the CDH packaging remains at 2.7 for CDH 5.10.1 and any future 5.10 maintenance releases.

Trying Impala

How do I try Impala out?

To look at the core features and functionality on Impala, the easiest way to try out Impala is to download the Cloudera QuickStart VM and start the Impala service through Cloudera Manager, then use `impala-shell` in a terminal window or the Impala Query UI in the Hue web interface.

To do performance testing and try out the management features for Impala on a cluster, you need to move beyond the QuickStart VM with its virtualized single-node environment. Ideally, download the Cloudera Manager software to set up the cluster, then install the Impala software through Cloudera Manager.

Does Cloudera offer a VM for demonstrating Impala?

Cloudera offers a demonstration VM called the QuickStart VM, available in VMWare, VirtualBox, and KVM formats. For more information, see the Cloudera QuickStart VM. After booting the QuickStart VM, many services are turned off by default; in the Cloudera Manager UI that appears automatically, turn on Impala and any other components that you want to try out.

Where can I find Impala documentation?

The core Impala developer and administrator information remains in the associated Impala documentation portion. Information about Impala release notes, installation, configuration, startup, and security is embedded in the corresponding CDH guides.

- Impala Upgrade Considerations on page 38
- Configuring Impala
- Security for Impala
- CDH Version and Packaging Information

Where can I get more information about Impala?

More product information is available here:

- O'Reilly introductory e-book: Cloudera Impala: Bringing the SQL and Hadoop Worlds Together
- O'Reilly getting started guide for developers: Getting Started with Impala: Interactive SQL for Apache Hadoop
- Blog: Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real
- Webinar: Introduction to Impala
- Product website page: Cloudera Enterprise RTQ

To see the latest release announcements for Impala, see the Cloudera Announcements forum.

How can I ask questions and provide feedback about Impala?

- Join the Impala discussion forum and the Impala mailing list to ask questions and provide feedback.
- Use the Impala Jira project to log bug reports and requests for features.

Where can I get sample data to try?

You can get scripts that produce data files and set up an environment for TPC-DS style benchmark tests from this Github repository. In addition to being useful for experimenting with performance, the tables are suited to experimenting with many aspects of SQL on Impala: they contain a good mixture of data types, data distributions, partitioning, and relational data suitable for join queries.

Impala System Requirements

What are the software and hardware requirements for running Impala?

For information on Impala requirements, see Impala Requirements on page 23. Note that there is often a minimum required level of Cloudera Manager for any given Impala version.

How much memory is required?

Although Impala is not an in-memory database, when dealing with large tables and large result sets, you should expect to dedicate a substantial portion of physical memory for the `impalad` daemon. Recommended physical memory for an Impala node is 128 GB or higher. If practical, devote approximately 80% of physical memory to Impala.

The amount of memory required for an Impala operation depends on several factors:

- The file format of the table. Different file formats represent the same data in more or fewer data files. The compression and encoding for each file format might require a different amount of temporary memory to decompress the data for analysis.
- Whether the operation is a `SELECT` or an `INSERT`. For example, Parquet tables require relatively little memory to query, because Impala reads and decompresses data in 8 MB chunks. Inserting into a Parquet table is a more memory-intensive operation because the data for each data file (potentially hundreds of megabytes, depending on the value of the `PARQUET_FILE_SIZE` query option) is stored in memory until encoded, compressed, and written to disk.

- Whether the table is partitioned or not, and whether a query against a partitioned table can take advantage of partition pruning.
- Whether the final result set is sorted by the `ORDER BY` clause. Each Impala node scans and filters a portion of the total data, and applies the `LIMIT` to its own portion of the result set. In CDH 5.1 / Impala 1.4 and higher, if the sort operation requires more memory than is available on any particular host, Impala uses a temporary disk work area to perform the sort. The intermediate result sets are all sent back to the coordinator node, which does the final sorting and then applies the `LIMIT` clause to the final result set.

  For example, if you execute the query:

```
select * from giant_table order by some_column limit 1000;
```

  and your cluster has 50 nodes, then each of those 50 nodes will transmit a maximum of 1000 rows back to the coordinator node. The coordinator node needs enough memory to sort (`LIMIT` * *cluster_size*) rows, although in the end the final result set is at most `LIMIT` rows, 1000 in this case.

  Likewise, if you execute the query:

```
select * from giant_table where test_val > 100 order by some_column;
```

  then each node filters out a set of rows matching the `WHERE` conditions, sorts the results (with no size limit), and sends the sorted intermediate rows back to the coordinator node. The coordinator node might need substantial memory to sort the final result set, and so might use a temporary disk work area for that final phase of the query.
- Whether the query contains any join clauses, `GROUP BY` clauses, analytic functions, or `DISTINCT` operators. These operations all require some in-memory work areas that vary depending on the volume and distribution of data. In Impala 2.0 and later, these kinds of operations utilize temporary disk work areas if memory usage grows too large to handle. See SQL Operations that Spill to Disk on page 607 for details.
- The size of the result set. When intermediate results are being passed around between nodes, the amount of data depends on the number of columns returned by the query. For example, it is more memory-efficient to query only the columns that are actually needed in the result set rather than always issuing `SELECT *`.
- The mechanism by which work is divided for a join query. You use the `COMPUTE STATS` statement, and query hints in the most difficult cases, to help Impala pick the most efficient execution plan. See Performance Considerations for Join Queries on page 568 for details.

See Hardware Requirements on page 24 for more details and recommendations about Impala hardware prerequisites.

What processor type and speed does Cloudera recommend?

Impala makes use of SSE 4.1 instructions.

What EC2 instances are recommended for Impala?

For large storage capacity and large I/O bandwidth, consider the `hs1.8xlarge` and `cc2.8xlarge` instance types. Impala I/O patterns typically do not benefit enough from SSD storage to make up for the lower overall size. For performance and security considerations for deploying CDH and its components on AWS, see Cloudera Enterprise Reference Architecture for AWS Deployments.

Supported and Unsupported Functionality In Impala

What are the main features of Impala?

- A large set of SQL statements, including SELECT and INSERT, with joins, Subqueries in Impala SELECT Statements on page 312, and Impala Analytic Functions on page 506. Highly compatible with HiveQL, and also including some vendor extensions. For more information, see Impala SQL Language Reference on page 101.

- Distributed, high-performance queries. See Tuning Impala for Performance on page 565 for information about Impala performance optimizations and tuning techniques for queries.
- Using Cloudera Manager, you can deploy and manage your Impala services. Cloudera Manager is the best way to get started with Impala on your cluster.
- Using Hue for queries.
- Appending and inserting data into tables through the INSERT statement. See How Impala Works with Hadoop File Formats on page 634 for the details about which operations are supported for which file formats.
- ODBC: Impala is certified to run against MicroStrategy and Tableau, with restrictions. For more information, see Configuring Impala to Work with ODBC on page 723.
- Querying data stored in HDFS and HBase in a single query. See Using Impala to Query HBase Tables on page 684 for details.
- In Impala 2.2.0 and higher, querying data stored in the Amazon Simple Storage Service (S3). See Using Impala with the Amazon S3 Filesystem on page 692 for details.
- Concurrent client requests. Each Impala daemon can handle multiple concurrent client requests. The effects on performance depend on your particular hardware and workload.
- Kerberos authentication. For more information, see Impala Security on page 82.
- Partitions. With Impala SQL, you can create partitioned tables with the CREATE TABLE statement, and add and drop partitions with the ALTER TABLE statement. Impala also takes advantage of the partitioning present in Hive tables. See Partitioning for Impala Tables on page 625 for details.

What features from relational databases or Hive are not available in Impala?

- Querying streaming data.
- Deleting individual rows. You delete data in bulk by overwriting an entire table or partition, or by dropping a table.
- Indexing (not currently). LZO-compressed text files can be indexed outside of Impala, as described in Using LZO-Compressed Text Files on page 640.
- Full text search on text fields. The Cloudera Search product is appropriate for this use case.
- Custom Hive Serializer/Deserializer classes (SerDes). Impala supports a set of common native file formats that have built-in SerDes in CDH. See How Impala Works with Hadoop File Formats on page 634 for details.
- Checkpointing within a query. That is, Impala does not save intermediate results to disk during long-running queries. Currently, Impala cancels a running query if any host on which that query is executing fails. When one or more hosts are down, Impala reroutes future queries to only use the available hosts, and Impala detects when the hosts come back up and begins using them again. Because a query can be submitted through any Impala node, there is no single point of failure. In the future, we will consider adding additional work allocation features to Impala, so that a running query would complete even in the presence of host failures.
- Hive indexes.
- Non-Hadoop data stores, such as relational databases.

For the detailed list of features that are different between Impala and HiveQL, see SQL Differences Between Impala and Hive on page 541.

Does Impala support generic JDBC?

Impala supports the HiveServer2 JDBC driver.

Is Avro supported?

Yes, Avro is supported. Impala has always been able to query Avro tables. You can use the Impala LOAD DATA statement to load existing Avro data files into a table. Starting with CDH 5.1 / Impala 1.4, you can create Avro tables with Impala. Currently, you still use the INSERT statement in Hive to copy data from another table into an Avro table. See Using the Avro File Format with Impala Tables on page 659 for details.

How do I?

How do I prevent users from seeing the text of SQL queries?

For instructions on making the Impala log files unreadable by unprivileged users, see Securing Impala Data and Log Files on page 83.

For instructions on password-protecting the web interface to the Impala log files and other internal server information, see Securing the Impala Web User Interface on page 84.

In CDH 5.4 / Impala 2.2 and higher, you can use the log redaction feature to obfuscate sensitive information in Impala log files. See http://www.cloudera.com/documentation/enterprise/latest/topics/sg_redaction.html for details.

How do I know how many Impala nodes are in my cluster?

The Impala statestore keeps track of how many `impalad` nodes are currently available. You can see this information through the statestore web interface. For example, at the URL `http://statestore_host:25010/metrics` you might see lines like the following:

```
statestore.live-backends:3
statestore.live-backends.list:[host1:22000, host1:26000, host2:22000]
```

The number of `impalad` nodes is the number of list items referring to port 22000, in this case two. (Typically, this number is one less than the number reported by the `statestore.live-backends` line.) If an `impalad` node became unavailable or came back after an outage, the information reported on this page would change appropriately.

Impala Performance

Are results returned as they become available, or all at once when a query completes?

Impala streams results whenever they are available, when possible. Certain SQL operations (aggregation or `ORDER BY`) require all of the input to be ready before Impala can return results.

Why does my query run slowly?

There are many possible reasons why a given query could be slow. Use the following checklist to diagnose performance issues with existing queries, and to avoid such issues when writing new queries, setting up new nodes, creating new tables, or loading data.

- Immediately after the query finishes, issue a `SUMMARY` command in `impala-shell`. You can check which phases of execution took the longest, and compare estimated values for memory usage and number of rows with the actual values.
- Immediately after the query finishes, issue a `PROFILE` command in `impala-shell`. The numbers in the `BytesRead`, `BytesReadLocal`, and `BytesReadShortCircuit` should be identical for a specific node. For example:

```
- BytesRead: 180.33 MB
- BytesReadLocal: 180.33 MB
- BytesReadShortCircuit: 180.33 MB
```

  If `BytesReadLocal` is lower than `BytesRead`, something in your cluster is misconfigured, such as the `impalad` daemon not running on all the data nodes. If `BytesReadShortCircuit` is lower than `BytesRead`, short-circuit reads are not enabled properly on that node; see Post-Installation Configuration for Impala on page 36 for instructions.
- If the table was just created, or this is the first query that accessed the table after an `INVALIDATE METADATA` statement or after the `impalad` daemon was restarted, there might be a one-time delay while the metadata for the table is loaded and cached. Check whether the slowdown disappears when the query is run again. When doing

performance comparisons, consider issuing a DESCRIBE *table_name* statement for each table first, to make sure any timings only measure the actual query time and not the one-time wait to load the table metadata.

- Is the table data in uncompressed text format? Check by issuing a DESCRIBE FORMATTED *table_name* statement. A text table is indicated by the line:

```
InputFormat: org.apache.hadoop.mapred.TextInputFormat
```

Although uncompressed text is the default format for a CREATE TABLE statement with no STORED AS clauses, it is also the bulkiest format for disk storage and consequently usually the slowest format for queries. For data where query performance is crucial, particularly for tables that are frequently queried, consider starting with or converting to a compact binary file format such as Parquet, Avro, RCFile, or SequenceFile. For details, see How Impala Works with Hadoop File Formats on page 634.

- If your table has many columns, but the query refers to only a few columns, consider using the Parquet file format. Its data files are organized with a column-oriented layout that lets queries minimize the amount of I/O needed to retrieve, filter, and aggregate the values for specific columns. See Using the Parquet File Format with Impala Tables on page 643 for details.
- If your query involves any joins, are the tables in the query ordered so that the tables or subqueries are ordered with the one returning the largest number of rows on the left, followed by the smallest (most selective), the second smallest, and so on? That ordering allows Impala to optimize the way work is distributed among the nodes and how intermediate results are routed from one node to another. For example, all other things being equal, the following join order results in an efficient query:

```
select some_col from
    huge_table join big_table join small_table join medium_table
  where
    huge_table.id = big_table.id
    and big_table.id = medium_table.id
    and medium_table.id = small_table.id;
```

See Performance Considerations for Join Queries on page 568 for performance tips for join queries.

- Also for join queries, do you have table statistics for the table, and column statistics for the columns used in the join clauses? Column statistics let Impala better choose how to distribute the work for the various pieces of a join query. See Table and Column Statistics on page 575 for details about gathering statistics.
- Does your table consist of many small data files? Impala works most efficiently with data files in the multi-megabyte range; Parquet, a format optimized for data warehouse-style queries, uses large files (originally 1 GB, now 256 MB in Impala 2.0 and higher) with a block size matching the file size. Use the DESCRIBE FORMATTED *table_name* statement in impala-shell to see where the data for a table is located, and use the hadoop fs -ls or hdfs dfs -ls Unix commands to see the files and their sizes. If you have thousands of small data files, that is a signal that you should consolidate into a smaller number of large files. Use an INSERT ... SELECT statement to copy the data to a new table, reorganizing into new data files as part of the process. Prefer to construct large data files and import them in bulk through the LOAD DATA or CREATE EXTERNAL TABLE statements, rather than issuing many INSERT ... VALUES statements; each INSERT ... VALUES statement creates a separate tiny data file. If you have thousands of files all in the same directory, but each one is megabytes in size, consider using a partitioned table so that each partition contains a smaller number of files. See the following point for more on partitioning.
- If your data is easy to group according to time or geographic region, have you partitioned your table based on the corresponding columns such as YEAR, MONTH, and/or DAY? Partitioning a table based on certain columns allows queries that filter based on those same columns to avoid reading the data files for irrelevant years, postal codes, and so on. (Do not partition down to too fine a level; try to structure the partitions so that there is still sufficient data in each one to take advantage of the multi-megabyte HDFS block size.) See Partitioning for Impala Tables on page 625 for details.

Why does my SELECT statement fail?

When a SELECT statement fails, the cause usually falls into one of the following categories:

- A timeout because of a performance, capacity, or network issue affecting one particular node.
- Excessive memory use for a join query, resulting in automatic cancellation of the query.

- A low-level issue affecting how native code is generated on each node to handle particular `WHERE` clauses in the query. For example, a machine instruction could be generated that is not supported by the processor of a certain node. If the error message in the log suggests the cause was an illegal instruction, consider turning off native code generation temporarily, and trying the query again.
- Malformed input data, such as a text data file with an enormously long line, or with a delimiter that does not match the character specified in the `FIELDS TERMINATED BY` clause of the `CREATE TABLE` statement.

Why does my INSERT statement fail?

When an `INSERT` statement fails, it is usually the result of exceeding some limit within a Hadoop component, typically HDFS.

- An `INSERT` into a partitioned table can be a strenuous operation due to the possibility of opening many files and associated threads simultaneously in HDFS. Impala 1.1.1 includes some improvements to distribute the work more efficiently, so that the values for each partition are written by a single node, rather than as a separate data file from each node.
- Certain expressions in the `SELECT` part of the `INSERT` statement can complicate the execution planning and result in an inefficient `INSERT` operation. Try to make the column data types of the source and destination tables match up, for example by doing `ALTER TABLE ... REPLACE COLUMNS` on the source table if necessary. Try to avoid `CASE` expressions in the `SELECT` portion, because they make the result values harder to predict than transferring a column unchanged or passing the column through a built-in function.
- Be prepared to raise some limits in the HDFS configuration settings, either temporarily during the `INSERT` or permanently if you frequently run such `INSERT` statements as part of your ETL pipeline.
- The resource usage of an `INSERT` statement can vary depending on the file format of the destination table. Inserting into a Parquet table is memory-intensive, because the data for each partition is buffered in memory until it reaches 1 gigabyte, at which point the data file is written to disk. Impala can distribute the work for an `INSERT` more efficiently when statistics are available for the source table that is queried during the `INSERT` statement. See Table and Column Statistics on page 575 for details about gathering statistics.

Does Impala performance improve as it is deployed to more hosts in a cluster in much the same way that Hadoop performance does?

Yes. Impala scales with the number of hosts. It is important to install Impala on all the DataNodes in the cluster, because otherwise some of the nodes must do remote reads to retrieve data not available for local reads. Data locality is an important architectural aspect for Impala performance. See this Impala performance blog post for background. Note that this blog post refers to benchmarks with Impala 1.1.1; Impala has added even more performance features in the 1.2.x series.

Is the HDFS block size reduced to achieve faster query results?

No. Impala does not make any changes to the HDFS or HBase data sets.

The default Parquet block size is relatively large (256 MB in Impala 2.0 and later; 1 GB in earlier releases). You can control the block size when creating Parquet files using the PARQUET_FILE_SIZE query option.

Does Impala use caching?

Impala does not cache table data. It does cache some table and file metadata. Although queries might run faster on subsequent iterations because the data set was cached in the OS buffer cache, Impala does not explicitly control this.

Impala takes advantage of the HDFS caching feature in CDH. You can designate which tables or partitions are cached through the `CACHED` and `UNCACHED` clauses of the `CREATE TABLE` and `ALTER TABLE` statements. Impala can also take advantage of data that is pinned in the HDFS cache through the `hdfscacheadmin` command. See Using HDFS Caching with Impala (CDH 5.3 or higher only) on page 593 for details.

Impala Use Cases

What are good use cases for Impala as opposed to Hive or MapReduce?

Impala is well-suited to executing SQL queries for interactive exploratory analytics on large data sets. Hive and MapReduce are appropriate for very long running, batch-oriented tasks such as ETL.

Is MapReduce required for Impala? Will Impala continue to work as expected if MapReduce is stopped?

Impala does not use MapReduce at all.

Can Impala be used for complex event processing?

For example, in an industrial environment, many agents may generate large amounts of data. Can Impala be used to analyze this data, checking for notable changes in the environment?

Complex Event Processing (CEP) is usually performed by dedicated stream-processing systems. Impala is not a stream-processing system, as it most closely resembles a relational database.

Is Impala intended to handle real time queries in low-latency applications or is it for ad hoc queries for the purpose of data exploration?

Ad-hoc queries are the primary use case for Impala. We anticipate it being used in many other situations where low-latency is required. Whether Impala is appropriate for any particular use-case depends on the workload, data size and query volume. See Impala Benefits on page 16 for the primary benefits you can expect when using Impala.

Questions about Impala And Hive

How does Impala compare to Hive and Pig?

Impala is different from Hive and Pig because it uses its own daemons that are spread across the cluster for queries. Because Impala does not rely on MapReduce, it avoids the startup overhead of MapReduce jobs, allowing Impala to return results in real time.

Can I do transforms or add new functionality?

Impala adds support for UDFs in Impala 1.2. You can write your own functions in C++, or reuse existing Java-based Hive UDFs. The UDF support includes scalar functions and user-defined aggregate functions (UDAs). User-defined table functions (UDTFs) are not currently supported.

Impala does not currently support an extensible serialization-deserialization framework (SerDes), and so adding extra functionality to Impala is not as straightforward as for Hive or Pig.

Can any Impala query also be executed in Hive?

Yes. There are some minor differences in how some queries are handled, but Impala queries can also be completed in Hive. Impala SQL is a subset of HiveQL, with some functional limitations such as transforms. For details of the Impala SQL dialect, see Impala SQL Statements on page 202. For the Impala built-in functions, see Impala Built-In Functions on page 391. For the detailed list of differences between Impala and HiveQL, see SQL Differences Between Impala and Hive on page 541.

Can I use Impala to query data already loaded into Hive and HBase?

There are no additional steps to allow Impala to query tables managed by Hive, whether they are stored in HDFS or HBase. Make sure that Impala is configured to access the Hive metastore correctly and you should be ready to go. Keep in mind that `impalad`, by default, runs as the `impala` user, so you might need to adjust some file permissions depending on how strict your permissions are currently.

See Using Impala to Query HBase Tables on page 684 for details about querying data in HBase.

Is Hive an Impala requirement?

The Hive metastore service is a requirement. Impala shares the same metastore database as Hive, allowing Impala and Hive to access the same tables transparently.

Hive itself is optional, and does not need to be installed on the same nodes as Impala. Currently, Impala supports a wider variety of read (query) operations than write (insert) operations; you use Hive to insert data into tables that use certain file formats. See How Impala Works with Hadoop File Formats on page 634 for details.

Impala Availability

Is Impala production ready?

Impala has finished its beta release cycle, and the 1.0, 1.1, and 1.2 GA releases are production ready. The 1.1.x series includes additional security features for authorization, an important requirement for production use in many organizations. The 1.2.x series includes important performance features, particularly for large join queries. Some Cloudera customers are already using Impala for large workloads.

The Impala 1.3.0 and higher releases are bundled with corresponding levels of CDH. The number of new features grows with each release. See New Features in CDH 6.0.0 for a full list.

How do I configure Hadoop high availability (HA) for Impala?

You can set up a proxy server to relay requests back and forth to the Impala servers, for load balancing and high availability. See Using Impala through a Proxy for High Availability on page 72 for details.

You can enable HDFS HA for the Hive metastore. See the CDH5 High Availability Guide for details.

What happens if there is an error in Impala?

There is not a single point of failure in Impala. All Impala daemons are fully able to handle incoming queries. If a machine fails however, all queries with fragments running on that machine will fail. Because queries are expected to return quickly, you can just rerun the query if there is a failure. See Impala Concepts and Architecture on page 18 for details about the Impala architecture.

The longer answer: Impala must be able to connect to the Hive metastore. Impala aggressively caches metadata so the metastore host should have minimal load. Impala relies on the HDFS NameNode, and you can configure HA for HDFS. Impala also has centralized services, known as the statestore and catalog services, that run on one host only. Impala continues to execute queries if the statestore host is down, but it will not get state updates. For example, if a host is added to the cluster while the statestore host is down, the existing instances of `impalad` running on the other hosts will not find out about this new host. Once the statestore process is restarted, all the information it serves is automatically reconstructed from all running Impala daemons.

What is the maximum number of rows in a table?

There is no defined maximum. Some customers have used Impala to query a table with over a trillion rows.

Can Impala and MapReduce jobs run on the same cluster without resource contention?

Yes. See Controlling Impala Resource Usage on page 588 for how to control Impala resource usage using the Linux cgroup mechanism, and Resource Management on page 549 for how to use Impala with the YARN resource management framework. Impala is designed to run on the DataNode hosts. Any contention depends mostly on the cluster setup and workload.

For a detailed information about configuring a cluster to share resources between Impala queries and MapReduce jobs, see https://www.cloudera.com/documentation/enterprise/latest/topics/admin_howto_multitenancy.html and Configuring Resource Pools and Admission Control on page 554.

Impala Internals

On which hosts does Impala run?

Cloudera strongly recommends running the `impalad` daemon on each DataNode for good performance. Although this topology is not a hard requirement, if there are data blocks with no Impala daemons running on any of the hosts containing replicas of those blocks, queries involving that data could be very inefficient. In that case, the data must be transmitted from one host to another for processing by "remote reads", a condition Impala normally tries to avoid. See Impala Concepts and Architecture on page 18 for details about the Impala architecture. Impala schedules query fragments on all hosts holding data relevant to the query, if possible.

In cases where some hosts in the cluster have much greater CPU and memory capacity than others, or where some hosts have extra CPU capacity because some CPU-intensive phases are single-threaded, some users have run multiple `impalad` daemons on a single host to take advantage of the extra CPU capacity. This configuration is only practical for specific workloads that rely heavily on aggregation, and the physical hosts must have sufficient memory to accommodate the requirements for multiple `impalad` instances.

How are joins performed in Impala?

By default, Impala automatically determines the most efficient order in which to join tables using a cost-based method, based on their overall size and number of rows. (This is a new feature in Impala 1.2.2 and higher.) The `COMPUTE STATS` statement gathers information about each table that is crucial for efficient join performance. Impala chooses between two techniques for join queries, known as "broadcast joins" and "partitioned joins". See Joins in Impala SELECT Statements on page 296 for syntax details and Performance Considerations for Join Queries on page 568 for performance considerations.

How does Impala process join queries for large tables?

Impala utilizes multiple strategies to allow joins between tables and result sets of various sizes. When joining a large table with a small one, the data from the small table is transmitted to each node for intermediate processing. When joining two large tables, the data from one of the tables is divided into pieces, and each node processes only selected pieces. See Joins in Impala SELECT Statements on page 296 for details about join processing, Performance Considerations for Join Queries on page 568 for performance considerations, and Optimizer Hints in Impala on page 387 for how to fine-tune the join strategy.

What is Impala's aggregation strategy?

Impala currently only supports in-memory hash aggregation. In Impala 2.0 and higher, if the memory requirements for a join or aggregation operation exceed the memory limit for a particular host, Impala uses a temporary work area on disk to help the query complete successfully.

How is Impala metadata managed?

Impala uses two pieces of metadata: the catalog information from the Hive metastore and the file metadata from the NameNode. Currently, this metadata is lazily populated and cached when an `impalad` needs it to plan a query.

The REFRESH statement updates the metadata for a particular table after loading new data through Hive. The INVALIDATE METADATA Statement on page 286 statement refreshes all metadata, so that Impala recognizes new tables or other DDL and DML changes performed through Hive.

In Impala 1.2 and higher, a dedicated `catalogd` daemon broadcasts metadata changes due to Impala DDL or DML statements to all nodes, reducing or eliminating the need to use the REFRESH and `INVALIDATE METADATA` statements.

What load do concurrent queries produce on the NameNode?

The load Impala generates is very similar to MapReduce. Impala contacts the NameNode during the planning phase to get the file metadata (this is only run on the host the query was sent to). Every `impalad` will read files as part of normal processing of the query.

How does Impala achieve its performance improvements?

These are the main factors in the performance of Impala versus that of other Hadoop components and related technologies.

Impala avoids MapReduce. While MapReduce is a great general parallel processing model with many benefits, it is not designed to execute SQL. Impala avoids the inefficiencies of MapReduce in these ways:

- Impala does not materialize intermediate results to disk. SQL queries often map to multiple MapReduce jobs with all intermediate data sets written to disk.
- Impala avoids MapReduce start-up time. For interactive queries, the MapReduce start-up time becomes very noticeable. Impala runs as a service and essentially has no start-up time.
- Impala can more naturally disperse query plans instead of having to fit them into a pipeline of map and reduce jobs. This enables Impala to parallelize multiple stages of a query and avoid overheads such as sort and shuffle when unnecessary.

Impala uses a more efficient execution engine by taking advantage of modern hardware and technologies:

- Impala generates runtime code. Impala uses LLVM to generate assembly code for the query that is being run. Individual queries do not have to pay the overhead of running on a system that needs to be able to execute arbitrary queries.
- Impala uses available hardware instructions when possible. Impala uses the supplemental SSE3 (SSSE3) instructions which can offer tremendous speedups in some cases. (Impala 2.0 and 2.1 required the SSE4.1 instruction set; Impala 2.2 and higher relax the restriction again so only SSSE3 is required.)
- Impala uses better I/O scheduling. Impala is aware of the disk location of blocks and is able to schedule the order to process blocks to keep all disks busy.
- Impala is designed for performance. A lot of time has been spent in designing Impala with sound performance-oriented fundamentals, such as tight inner loops, inlined function calls, minimal branching, better use of cache, and minimal memory usage.

What happens when the data set exceeds available memory?

Currently, if the memory required to process intermediate results on a node exceeds the amount available to Impala on that node, the query is cancelled. You can adjust the memory available to Impala on each node, and you can fine-tune the join strategy to reduce the memory required for the biggest queries. We do plan on supporting external joins and sorting in the future.

Keep in mind though that the memory usage is not directly based on the input data set size. For aggregations, the memory usage is the number of rows *after* grouping. For joins, the memory usage is the combined size of the tables *excluding* the biggest table, and Impala can use join strategies that divide up large joined tables among the various nodes rather than transmitting the entire table to each node.

What are the most memory-intensive operations?

If a query fails with an error indicating “memory limit exceeded”, you might suspect a memory leak. The problem could actually be a query that is structured in a way that causes Impala to allocate more memory than you expect, exceeded the memory allocated for Impala on a particular node. Some examples of query or table structures that are especially memory-intensive are:

- `INSERT` statements using dynamic partitioning, into a table with many different partitions. (Particularly for tables using Parquet format, where the data for each partition is held in memory until it reaches the full block size in size before it is written to disk.) Consider breaking up such operations into several different `INSERT` statements, for example to load data one year at a time rather than for all years at once.
- `GROUP BY` on a unique or high-cardinality column. Impala allocates some handler structures for each different value in a `GROUP BY` query. Having millions of different `GROUP BY` values could exceed the memory limit.
- Queries involving very wide tables, with thousands of columns, particularly with many `STRING` columns. Because Impala allows a `STRING` value to be up to 32 KB, the intermediate results during such queries could require substantial memory allocation.

When does Impala hold on to or return memory?

Impala allocates memory using `tcmalloc`, a memory allocator that is optimized for high concurrency. Once Impala allocates memory, it keeps that memory reserved to use for future queries. Thus, it is normal for Impala to show high memory usage when idle. If Impala detects that it is about to exceed its memory limit (defined by the `-mem_limit` startup option or the `MEM_LIMIT` query option), it deallocates memory not needed by the current queries.

When issuing queries through the JDBC or ODBC interfaces, make sure to call the appropriate close method afterwards. Otherwise, some memory associated with the query is not freed.

SQL

Is there an UPDATE statement?

In CDH 5.10 / Impala 2.8 and higher, Impala has the statements `UPDATE`, `DELETE`, and `UPSERT`. These statements apply to Kudu tables only.

For non-Kudu tables, you can use the following techniques to achieve the same goals as the familiar `UPDATE` statement, in a way that preserves efficient file layouts for subsequent queries:

- Replace the entire contents of a table or partition with updated data that you have already staged in a different location, either using `INSERT OVERWRITE`, `LOAD DATA`, or manual HDFS file operations followed by a `REFRESH` statement for the table. Optionally, you can use built-in functions and expressions in the `INSERT` statement to transform the copied data in the same way you would normally do in an `UPDATE` statement, for example to turn a mixed-case string into all uppercase or all lowercase.
- To update a single row, use an HBase table, and issue an `INSERT ... VALUES` statement using the same key as the original row. Because HBase handles duplicate keys by only returning the latest row with a particular key value, the newly inserted row effectively hides the previous one.

Can Impala do user-defined functions (UDFs)?

Impala 1.2 and higher does support UDFs and UDAs. You can either write native Impala UDFs and UDAs in C++, or reuse UDFs (but not UDAs) originally written in Java for use with Hive. See User-Defined Functions (UDFs) on page 525 for details.

Why do I have to use REFRESH and INVALIDATE METADATA, what do they do?

In Impala 1.2 and higher, there is much less need to use the `REFRESH` and `INVALIDATE METADATA` statements:

- The new `impala-catalog` service, represented by the `catalogd` daemon, broadcasts the results of Impala DDL statements to all Impala nodes. Thus, if you do a `CREATE TABLE` statement in Impala while connected to one node, you do not need to do `INVALIDATE METADATA` before issuing queries through a different node.
- The catalog service only recognizes changes made through Impala, so you must still issue a `REFRESH` statement if you load data through Hive or by manipulating files in HDFS, and you must issue an `INVALIDATE METADATA` statement if you create a table, alter a table, add or drop partitions, or do other DDL statements in Hive.
- Because the catalog service broadcasts the results of `REFRESH` and `INVALIDATE METADATA` statements to all nodes, in the cases where you do still need to issue those statements, you can do that on a single node rather than on every node, and the changes will be automatically recognized across the cluster, making it more convenient to load balance by issuing queries through arbitrary Impala nodes rather than always using the same coordinator node.

Why is space not freed up when I issue DROP TABLE?

Impala deletes data files when you issue a `DROP TABLE` on an internal table, but not an external one. By default, the `CREATE TABLE` statement creates internal tables, where the files are managed by Impala. An external table is created with a `CREATE EXTERNAL TABLE` statement, where the files reside in a location outside the control of Impala. Issue a `DESCRIBE FORMATTED` statement to check whether a table is internal or external. The keyword `MANAGED_TABLE`

indicates an internal table, from which Impala can delete the data files. The keyword EXTERNAL_TABLE indicates an external table, where Impala will leave the data files untouched when you drop the table.

Even when you drop an internal table and the files are removed from their original location, you might not get the hard drive space back immediately. By default, files that are deleted in HDFS go into a special trashcan directory, from which they are purged after a period of time (by default, 6 hours). For background information on the trashcan mechanism, see HDFS Architecture. For information on purging files from the trashcan, see File System Shell.

When Impala deletes files and they are moved to the HDFS trashcan, they go into an HDFS directory owned by the `impala` user. If the `impala` user does not have an HDFS home directory where a trashcan can be created, the files are not deleted or moved, as a safety measure. If you issue a `DROP TABLE` statement and find that the table data files are left in their original location, create an HDFS directory `/user/impala`, owned and writeable by the `impala` user. For example, you might find that `/user/impala` is owned by the `hdfs` user, in which case you would switch to the `hdfs` user and issue a command such as:

```
hdfs dfs -chown -R impala /user/impala
```

Is there a DUAL table?

You might be used to running queries against a single-row table named DUAL to try out expressions, built-in functions, and UDFs. Impala does not have a DUAL table. To achieve the same result, you can issue a SELECT statement without any table name:

```
select 2+2;
select substr('hello',2,1);
select pow(10,6);
```

Partitioned Tables

How do I load a big CSV file into a partitioned table?

To load a data file into a partitioned table, when the data file includes fields like year, month, and so on that correspond to the partition key columns, use a two-stage process. First, use the `LOAD DATA` or `CREATE EXTERNAL TABLE` statement to bring the data into an unpartitioned text table. Then use an `INSERT ... SELECT` statement to copy the data from the unpartitioned table to a partitioned one. Include a `PARTITION` clause in the `INSERT` statement to specify the partition key columns. The `INSERT` operation splits up the data into separate data files for each partition. For examples, see Partitioning for Impala Tables on page 625. For details about loading data into partitioned Parquet tables, a popular choice for high-volume data, see Loading Data into Parquet Tables on page 644.

Can I do INSERT ... SELECT * into a partitioned table?

When you use the `INSERT ... SELECT *` syntax to copy data into a partitioned table, the columns corresponding to the partition key columns must appear last in the columns returned by the `SELECT *`. You can create the table with the partition key columns defined last. Or, you can use the `CREATE VIEW` statement to create a view that reorders the columns: put the partition key columns last, then do the `INSERT ... SELECT *` from the view.

HBase

What kinds of Impala queries or data are best suited for HBase?

HBase tables are ideal for queries where normally you would use a key-value store. That is, where you retrieve a single row or a few rows, by testing a special unique key column using the = or `IN` operators.

HBase tables are not suitable for queries that produce large result sets with thousands of rows. HBase tables are also not suitable for queries that perform full table scans because the WHERE clause does not request specific values from the unique key column.

Use HBase tables for data that is inserted one row or a few rows at a time, such as by the INSERT ... VALUES syntax. Loading data piecemeal like this into an HDFS-backed table produces many tiny files, which is a very inefficient layout for HDFS data files.

If the lack of an UPDATE statement in Impala is a problem for you, you can simulate single-row updates by doing an INSERT ... VALUES statement using an existing value for the key column. The old row value is hidden; only the new row value is seen by queries.

HBase tables are often wide (containing many columns) and sparse (with most column values NULL). For example, you might record hundreds of different data points for each user of an online service, such as whether the user had registered for an online game or enabled particular account features. With Impala and HBase, you could look up all the information for a specific customer efficiently in a single query. For any given customer, most of these columns might be NULL, because a typical customer might not make use of most features of an online service.

Appendix: Apache License, Version 2.0

SPDX short identifier: Apache-2.0

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims

licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

1. You must give any other recipients of the Work or Derivative Works a copy of this License; and
2. You must cause any modified files to carry prominent notices stating that You changed the files; and
3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

  http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```