cloudera®
Ask Bigger Questions

Cloudera Impala User Guide

Cloudera, Inc.
1001 Page Mill Road Bldg 2
Palo Alto, CA 94304
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information

Version: 1.4.x
Date: September 8, 2015

Table of Contents

Introducing Cloudera Impala

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

Cloudera 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.

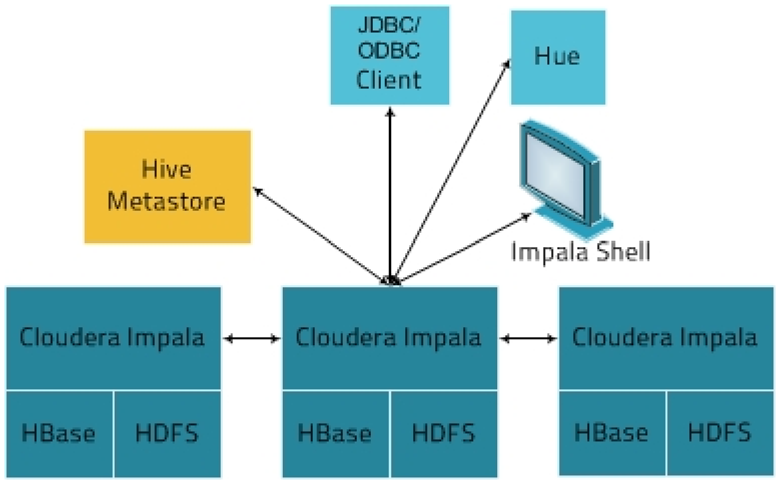
Impala Benefits

Impala provides:

- Familiar SQL interface that data scientists and analysts already know
- Ability to interactively query data on 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 and read with Impala, or to write with Impala and read with Hive
- Single system for big data processing and analytics, so customers can avoid costly modeling and ETL just for analytics

How Cloudera 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.
- Cloudera 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 and HBase storage, including:
  - HDFS file formats: Text file, SequenceFile, RCFile, Avro file, and Parquet.
  - Compression codecs: Snappy, GZIP, Deflate, BZIP.
- Common Hive interfaces including:
  - JDBC driver.
  - ODBC driver.
  - Hue Beeswax and the new Cloudera Impala Query UI.
- Impala command-line interface.
- Kerberos authentication.

Impala Concepts and Architecture

The following sections provide background information to help you become productive using Cloudera 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 a daemon process that runs on each node of the cluster, physically represented by the `impalad` process. It reads and writes to data files; accepts queries transmitted from the `impala-shell` command, Hue, JDBC, or ODBC; parallelizes the queries and distributes work to other nodes in the Impala cluster; and transmits intermediate query results back to the central coordinator node.

You can submit a query to the Impala daemon running on any node, and that node serves as the **coordinator node** for that query. The other nodes transmit partial results back to the coordinator, which constructs the final result set for a query. When running experiments with functionality through the `impala-shell` command, you might always connect to the same Impala daemon for convenience. For clusters running production workloads, you might load-balance between the nodes by submitting each query to a different Impala daemon in round-robin style, using the JDBC or ODBC interfaces.

The Impala daemons are in constant communication with the **statestore**, to confirm which nodes are healthy and can accept new work.

They also receive broadcast messages from the `catalogd` daemon (introduced in Impala 1.2) whenever any Impala node 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 nodes prior to Impala 1.2.

Related information: Modifying Impala Startup Options, Starting Impala, Setting the Idle Query and Idle Session Timeouts for impalad on page 45, Appendix A - Ports Used by Impala on page 279, Using Impala through a Proxy for High Availability on page 45

The Impala Statestore

The Impala component known as the **statestore** checks on the health of **Impala daemons** on all the nodes 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 node in the cluster. If an Impala node goes offline due to hardware failure, network error, software issue, or other reason, the statestore informs all the other nodes so that future queries can avoid making requests to the unreachable node.

Because the statestore's purpose is to help when things go wrong, it is not critical to the normal operation of an Impala cluster. If the statestore is not running or becomes unreachable, the other nodes continue running and distributing work among themselves as usual; the cluster just becomes less robust if other nodes fail while the statestore is offline. When the statestore comes back online, it re-establishes communication with the other nodes and resumes its monitoring function.

Related information: Modifying Impala Startup Options, Starting Impala, Increasing the Statestore Timeout on page 44, Appendix A - Ports Used by Impala on page 279

The Impala Catalog Service

The Impala component known as the **catalog service** relays the metadata changes from Impala SQL statements to all the nodes in a cluster. It is physically represented by a daemon process named `catalogd`; you only need such a process on one node in the cluster. Because the requests are passed through the statestore daemon, it makes sense to run the `statestored` and `catalogd` services on the same node.

This new component in Impala 1.2 reduces the need for the `REFRESH` and `INVALIDATE METADATA` statements. Formerly, if you issued `CREATE DATABASE`, `DROP DATABASE`, `CREATE TABLE`, `ALTER TABLE`, or `DROP TABLE` statements on one Impala node, you needed to issue `INVALIDATE METADATA` on any other node before running a query there, so that it would pick up the changes to schema objects. Likewise, if you issued `INSERT` statements on one node, you needed to issue `REFRESH` *`table_name`* on any other node before running a query there, so that it would recognize the newly added data files. The catalog service removes the need to issue `REFRESH` and `INVALIDATE METADATA` statements when the metadata changes are performed by statement issued through Impala; when you create a table, load data, and so on through Hive, you still need to issue `REFRESH` or `INVALIDATE METADATA` on an Impala node before executing a query there.

This feature, new in Impala 1.2, touches a number of aspects of Impala:

- See Impala Installation, Upgrading Impala and Starting Impala, for usage information for the `catalogd` daemon.
- The `REFRESH` and `INVALIDATE METADATA` statements are no longer 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 node rather than on all nodes. See REFRESH Statement on page 116 and INVALIDATE METADATA Statement on page 111 for the latest usage information for those statements.
- See The Impala Catalog Service on page 14 for background information on the `catalogd` service.

By default, the metadata loading and caching on startup happens asynchronously, so Impala can begin accepting requests promptly. To enable the original behavior, where Impala waited until all metadata was loaded before accepting any requests, set the `catalogd` configuration option `--load_catalog_in_background=false`.

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. See New Features in Impala Version 1.2.4 for details.

Related information: Modifying Impala Startup Options, Starting Impala, Appendix A - Ports Used by Impala on page 279

Programming 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 descended from 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.

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

- 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 loading 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`). There is no `INSERT ... VALUES` syntax to insert a single row.
- 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.
- 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 impose length constraints on string data types. For example, you define a database column as `STRING` rather than `CHAR(1)` or `VARCHAR(64)`.
- For query-intensive applications, 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`.
- From the data warehousing world, you will recognize the notion of partitioned tables.
- In Impala 1.2 and higher, UDFs let you perform custom comparisons and transformation logic during `SELECT` and `INSERT...SELECT` statements.

Related information: Impala SQL Language Reference on page 49

Overview of Impala Programming Interfaces

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

- The `impala-shell` interactive command interpreter.
- The Apache 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 faster, 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 16, 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 14 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 265 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.

Cluster Sizing Guidelines for Impala

This document provides a very rough guideline to estimate the size of a cluster needed for a specific customer application. You can use this information when planning how much and what type of hardware to acquire for a new cluster, or when adding Impala workloads to an existing cluster.

Note: Before making purchase or deployment decisions, consult your Cloudera representative to verify the conclusions about hardware requirements based on your data volume and workload.

Always use hosts with identical specifications and capacities for all the nodes in the cluster. Currently, Impala divides the work evenly between cluster nodes, regardless of their exact hardware configuration. Because work can be distributed in different ways for different queries, if some hosts are overloaded compared to others in terms of CPU, memory, I/O, or network, you might experience inconsistent performance and overall slowness

For analytic workloads with star/snowflake schemas, and using consistent hardware for all nodes (64 GB RAM, 12 2 TB hard drives, 2x E5-2630L 12 cores total, 10 GB network), the following table estimates the number of data nodes needed in the cluster based on data size and the number of concurrent queries, for workloads similar to TPC-DS benchmark queries:

Table 1: Cluster size estimation based on the number of concurrent queries and data size with a 20 second average query response time

Data Size	1 query	10 queries	100 queries	1000 queries	2000 queries
250 GB	2	2	5	35	70
500 GB	2	2	10	70	135
1 TB	2	2	15	135	270
15 TB	2	20	200	N/A	N/A
30 TB	4	40	400	N/A	N/A
60 TB	8	80	800	N/A	N/A

Factors Affecting Scalability

A typical analytic workload (TPC-DS style queries) using recommended hardware is usually CPU-bound. Each node can process roughly 1.6 GB/sec. Both CPU-bound and disk-bound workloads can scale almost linearly with cluster size. However, for some workloads, the scalability might be bounded by the network, or even by memory.

If the workload is already network bound (on a 10 GB network), increasing the cluster size won’t reduce the network load; in fact, a larger cluster could increase network traffic because some queries involve “broadcast” operations to all data nodes. Therefore, boosting the cluster size does not improve query throughput in a network-constrained environment.

Let’s look at a memory-bound workload. A workload is memory-bound if Impala cannot run any additional concurrent queries because all memory allocated has already been consumed, but neither CPU, disk, nor network is saturated yet. This can happen because currently Impala uses only a single core per node to process join and aggregation queries. For a node with 128 GB of RAM, if a join node takes 50 GB, the system cannot run more than 2 such queries at the same time.

Therefore, at most 2 cores are used. Throughput can still scale almost linearly even for a memory-bound workload. It's just that the CPU will not be saturated. Per-node throughput will be lower than 1.6 GB/sec. Consider increasing the memory per node.

As long as the workload is not network- or memory-bound, we can use the 1.6 GB/second per node as the throughput estimate.

A More Precise Approach

A more precise sizing estimate would require not only queries per minute (QPM), but also an average data size scanned per query (D). With the proper partitioning strategy, D is usually a fraction of the total data size. The following equation can be used as a rough guide to estimate the number of nodes (N) needed:

```
Eq 1: N > QPM * D / 100 GB
```

Here is an example. Suppose, on average, a query scans 50 GB of data and the average response time is required to be 15 seconds or less when there are 100 concurrent queries. The QPM is 100/15*60 = 400. We can estimate the number of node using our equation above.

```
N > QPM * D / 100GB
N > 400 * 50GB / 100GB
N > 200
```

Because this figure is a rough estimate, the corresponding number of nodes could be between 100 and 500.

Depending on the complexity of the query, the processing rate of query might change. If the query has more joins, aggregation functions, or CPU-intensive functions such as string processing or complex UDFs, the process rate will be lower than 1.6 GB/second per node. On the other hand, if the query only does scan and filtering on numbers, the processing rate can be higher.

Estimating Memory Requirements

Impala can handle joins between multiple large tables. Make sure that statistics are collected for all the joined tables, using the `COMPUTE STATS` statement. However, joining big tables does consume more memory. Follow the steps below to calculate the minimum memory requirement.

Suppose you are running the following join:

```
select a.*, b.col_1, b.col_2, … b.col_n
from a, b
where a.key = b.key
and b.col_1 in (1,2,4...)
and b.col_4 in (....);
```

And suppose table `B` is smaller than table `A` (but still a large table).

The memory requirement for the query is the right-hand table (`B`), after decompression, filtering (`b.col_n in ...`) and after projection (only using certain columns) must be less than the total memory of the entire cluster.

```
Cluster Total Memory Requirement  = Size of the smaller table *
  selectivity factor from the predicate *
  projection factor * compression ratio
```

In this case, assume that table `B` is 100 TB in Parquet format with 200 columns. The predicate on `B` (`b.col_1 in ...and b.col_4 in ...`) will select only 10% of the rows from `B` and for projection, we are only projecting

5 columns out of 200 columns. Usually, Snappy compression gives us 3 times compression, so we estimate a 3x compression factor.

```
Cluster Total Memory Requirement  = Size of the smaller table *
  selectivity factor from the predicate *
  projection factor * compression ratio
  = 100TB * 10% * 5/200 * 3
  = 0.75TB
  = 750GB
```

So, if you have a 10-node cluster, each node has 128 GB of RAM and you give 80% to Impala, then you have 1 TB of usable memory for Impala, which is more than 750GB. Therefore, your cluster can handle join queries of this magnitude.

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 180 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 don't over-partition.

Partitioning is an important aspect of performance tuning for Impala. Follow the procedures in Partitioning on page 233 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. 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 84 for the syntax of the `COMPUTE STATS` statement, and How Impala Uses Statistics for Query Optimization on page 212 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 203.

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.

Impala Tutorial

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:

- If you already have a CDH environment set up and just need to add Impala to it, follow the installation process described in Impala Installation. Make sure to also install Hive and its associated metastore database 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 experiments and then discard, set up the Cloudera QuickStart VM, which includes CDH and Impala on CentOS 6.3 (64-bit). 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.

Set Up Some Basic .csv Tables

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 user name for `cloudera` 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
cloudera
$ hdfs dfs -ls /user
Found 3 items
drwxr-xr-x   - cloudera cloudera            0 2013-04-22 18:54 /user/cloudera
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/cloudera/sample_data/tab1 /user/cloudera/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/cloudera/sample_data/tab1
$ hdfs dfs -ls /user/cloudera/sample_data/tab1
Found 1 items
-rw-r--r--   1 cloudera cloudera        192 2013-04-02 20:08
/user/cloudera/sample_data/tab1/tab1.csv


$ hdfs dfs -put tab2.csv /user/cloudera/sample_data/tab2
$ hdfs dfs -ls /user/cloudera/sample_data/tab2
Found 1 items
-rw-r--r--   1 cloudera cloudera        158 2013-04-02 20:09
/user/cloudera/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/cloudera/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/cloudera/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 you followed the installation instructions closely, in Impala Installation.
- 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.

Note:

In early beta Impala releases, the examples in this tutorial relied on the Hive `CREATE TABLE` command. The `CREATE TABLE` statement is available in Impala for 0.7 and higher, so now the tutorial uses the native Impala `CREATE TABLE`.

The following examples set up 2 tables, referencing the paths and sample data supplied with the Impala Demo VM. 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:

```
$ hdfs dfs -ls /user/hive/tpcds/customer
Found 1 items
-rw-r--r--   1 cloudera 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||Javier.
Lewis@VFAxlnZEvOx.org|2452508|
2|AAAAAAAACAAAAAAA|819667|1461|31655|2452318|2452288|Dr.|Amy|Moses|Y|9|4|1966|TOGO||Amy.Moses@Ov
k9KjHH.com|2452318|
3|AAAAAAAADAAAAAAA|1473522|6247|48572|2449130|2449100|Miss|Latisha|Hamilton|N|18|9|1979|NIUE||La
tisha.Hamilton@V.com|2452313|
4|AAAAAAAAEAAAAAAA|1703214|3986|39558|2450030|2450000|Dr.|Michael|White|N|7|6|1983|MEXICO||Micha
el.White@i.org|2452361|
5|AAAAAAAAFAAAAAAA|953372|4470|36368|2449438|2449408|Sir|Robert|Moran|N|8|5|1956|FIJI||Robert.Mo
ran@Hh.edu|2452469|
...
```

Here is the SQL script we might save 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
```

Note:

Currently, the `impala-shell` interpreter requires that any command entered interactively be a single line, so if you experiment with these commands yourself, either save to a `.sql` file and use the `-f` option to run the script, or wrap each command onto one line before pasting into the shell.

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 9071849993695174436_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 259 and Using the SequenceFile File Format with Impala Tables on page 261 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 255 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 163 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
```

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 Installation, Upgrading Impala, and Configuring Impala for details.

For additional security tasks typically performed by administrators, see Impala Security Configuration.

For a detailed example of configuring a cluster to share resources between Impala queries and MapReduce jobs, see Setting up a Multi-tenant Cluster for Impala and MapReduce

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. Enable this feature if your cluster is underutilized at some times and overutilized at others. Overutilization is indicated by performance bottlenecks and queries being cancelled due to out-of-memory conditions, when those same queries are successful and perform well during times with less concurrent load. Admission control works as a safeguard to avoid out-of-memory conditions during heavy concurrent usage.

Important: Cloudera strongly recommends you upgrade to CDH 5.0.0 or later to use admission control. In CDH 4, admission control will only work if you *don't* have Hue deployed; unclosed Hue queries will accumulate and exceed the queue size limit. On CDH 4, to use admission control, you must explicitly enable it by specifying `--disable_admission_control=false` in the `impalad` command-line options safety valve field.

Important:

Use the `COMPUTE STATS` statement for large tables involved in join queries, and follow other steps from Tuning Impala for Performance on page 203 to tune your queries. Although `COMPUTE STATS` is an important statement to help optimize query performance, it is especially important when admission control is enabled:

- 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.
- The admission control feature also relies on the statistics produced by the `COMPUTE STATS` statement to generate accurate estimates of memory usage for complex queries. If the estimates are inaccurate due to missing statistics, Impala might hold back queries unnecessarily even though there is sufficient memory to run them, or might allow queries to run that end up exceeding the memory limit and being cancelled.

Overview of Impala Admission Control

On a busy CDH cluster, you might find there is an optimal number of Impala queries that run concurrently. Because Impala queries are typically I/O-intensive, you might not find any throughput benefit in running more concurrent queries when the I/O capacity is fully utilized. Because Impala by default cancels queries that exceed the specified memory limit, running multiple large-scale queries at once can result in having to re-run some queries that are cancelled.

The admission control feature lets you set a cluster-wide 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.

For details on the internal workings of admission control, see How Impala Schedules and Enforces Limits on Concurrent Queries on page 34.

How Impala Admission Control Relates to YARN

The admission control feature is similar in some ways to the YARN resource management framework, and they can be used separately or together. This section describes some similarities and differences, to help you decide when to use one, the other, or both together.

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 is not aware of other Hadoop workloads such as MapReduce jobs, you might use YARN with static service pools on heterogeneous CDH 5 clusters where resources are shared between Impala and other Hadoop components. Devote a percentage of cluster resources to Impala, 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 remainder of work within the cluster.

You could also try out the combination of YARN, Impala, and Llama, where YARN manages all cluster resources and Impala queries request resources from YARN by using the Llama component as an intermediary. YARN is a more centralized, general-purpose service, with somewhat higher latency than admission control due to the requirement to pass requests back and forth through the YARN and Llama components.

The Impala admission control feature uses the same mechanism as the YARN resource manager to map users to pools and authenticate them. Although the YARN resource manager is only available with CDH 5 and higher, internally Impala includes the necessary infrastructure to work consistently on both CDH 4 and CDH 5. You do not need to run the actual YARN and Llama components for admission control to operate.

In Cloudera Manager, the controls for Impala resource management change slightly depending on whether the Llama role is enabled, which brings Impala under the control of YARN. When you use Impala without the Llama role, you can specify three properties (memory limit, query queue size, and queue timeout) for the admission control feature. When the Llama role is enabled, you can specify query queue size and queue timeout, but the memory limit is enforced by YARN and not settable through the **Dynamic Resource Pools** page.

How Impala Schedules and Enforces Limits on Concurrent Queries

The admission control system is decentralized, embedded in each `impalad` daemon and communicating through the statestore mechanism. Although the limits you set for memory usage and number of concurrent queries apply cluster-wide, each `impalad` 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. There could be times when the query queue contained more queries than the specified limit, or when the estimated of memory usage for a query is not exact and the overall memory usage exceeds the specified limit. 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 the memory limit, to leave some headroom for queries to use more memory than expected, without being cancelled as a result.

At any time, the set of queued queries could include queries submitted through multiple different `impalad` nodes. All the queries submitted through a particular node will be executed in order, so a `CREATE TABLE` followed by an `INSERT` on the same table would succeed. Queries submitted through different nodes 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 nodes, set up all table structures ahead of time so that the statements controlled by the queueing 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 `impalad` node.

The limit on the number of concurrent queries is a "soft" one, To achieve high throughput, Impala makes quick decisions at the node level about which queued queries to dispatch. Therefore, Impala might slightly exceed the limit from time to time.

To avoid a large backlog of queued requests, you can also set an upper limit on the size of the queue for queries that are delayed. When the number of queued queries exceeds this limit, further queries are cancelled rather than being queued. You can also configure a timeout period, 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, HiveServer 2)

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.

Admission control has the following limitations or special behavior when used with JDBC or ODBC applications:

- If you want to submit queries to different resource pools through the `REQUEST_POOL` query option, as described in REQUEST_POOL on page 201, that option is only settable for a session through the `impala-shell` interpreter or cluster-wide through an `impalad` startup option.
- The `MEM_LIMIT` query option, sometimes useful to work around problems caused by inaccurate memory estimates for complicated queries, is only settable through the `impala-shell` interpreter and cannot be used directly through JDBC or ODBC applications.
- Admission control does not use the other resource-related query options, `RESERVATION_REQUEST_TIMEOUT` or `V_CPU_CORES`. Those query options only apply to the YARN resource management framework.

Configuring Admission Control

The configuration options for admission control range from the simple (a single resource pool with a single set of options) to the complex (multiple resource pools with different options, each pool handling queries for a different set of users and groups). You can configure the settings through the Cloudera Manager user interface, or on a system without Cloudera Manager by editing configuration files or through startup options to the `impalad` daemon.

Configuring Admission Control with Cloudera Manager

In the Cloudera Manager Admin Console, you can configure pools to manage queued Impala queries, and the options for the limit on number of concurrent queries and how to handle queries that exceed the limit. For details, see the Cloudera Manager documentation for managing resources.

See Examples of Admission Control Configurations on page 38 for a sample setup for admission control under Cloudera Manager.

Configuring Admission Control Manually

If you do not use Cloudera Manager, you use a combination of startup options for the `impalad` daemon, and optionally editing or manually constructing the configuration files `fair-scheduler.xml` and `llama-site.xml`.

> **Note:** Because Cloudera Manager 5 includes a GUI for these settings but Cloudera Manager 4 does not, if you are using Cloudera Manager 4, include the appropriate configuration options in the `impalad` command-line options safety valve field.

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.

The `impalad` configuration options related to the admission control feature are:

`--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: `0`

`--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`

`--default_pool_mem_limit`

Purpose: Maximum amount of memory 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 upper- 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 84 for guidelines about how and when to use this statement.

Type: string

Default: `""` (empty string, meaning unlimited)

`--disable_admission_control`

Purpose: Turns off the admission control feature entirely, regardless of other configuration option settings.

Type: Boolean

Default: `false`

`--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 Llama configuration file (`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 small subset of the settings that can go in this file, as described below. For details about all the Llama configuration settings, see the documentation on Github.

`--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`

For an advanced configuration with multiple resource pools using different settings, set up the `fair-scheduler.xml` and `llama-site.xml` configuration files manually. 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
```

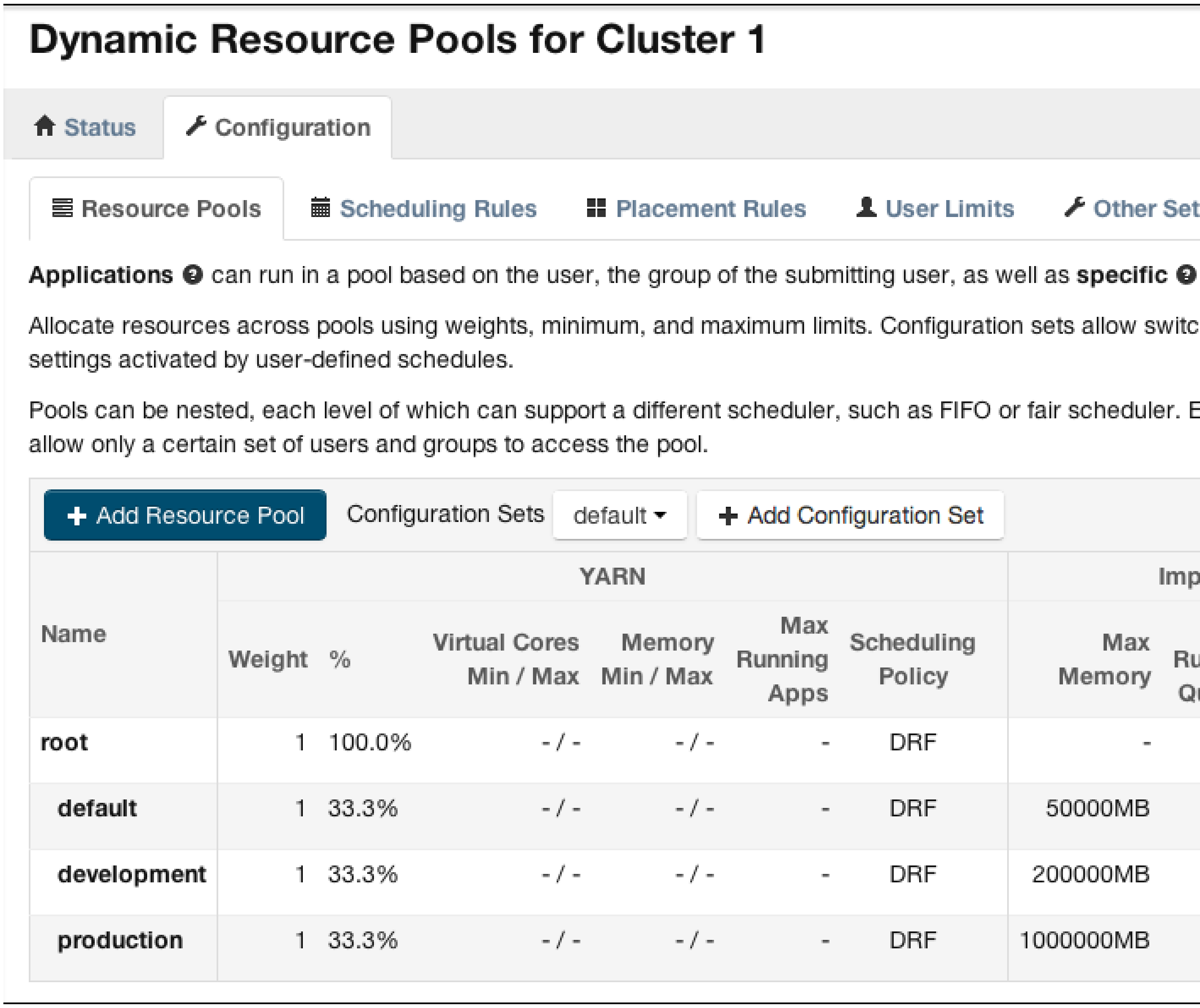
For details about all the Llama configuration settings, see the documentation on Github.

See Examples of Admission Control Configurations on page 38 for sample configuration files for admission control using multiple resource pools, without Cloudera Manager.

Examples of Admission Control Configurations

For full instructions about configuring dynamic resource pools through Cloudera Manager, see Dynamic Resource Pools in the Cloudera Manager documentation. The following examples demonstrate some important points related to the Impala admission control feature.

The following figure shows a sample of the **Dynamic Resource Pools** page in Cloudera Manager, accessed through the **Clusters** > ***ClusterName*** > **Other** > **Dynamic Resource Pools** > **Configuration** menu choice. Numbers from all the resource pools are combined into the topmost `root` pool. The `default` pool is for users who are not assigned any other pool by the user-to-pool mapping settings. The `development` and `production` pools show how you can set different limits for different classes of users, for total memory, number of concurrent queries, and number of queries that can be queued.


Dynamic Resource Pools for Cluster 1

Applications can run in a pool based on the user, the group of the submitting user, as well as **specific**

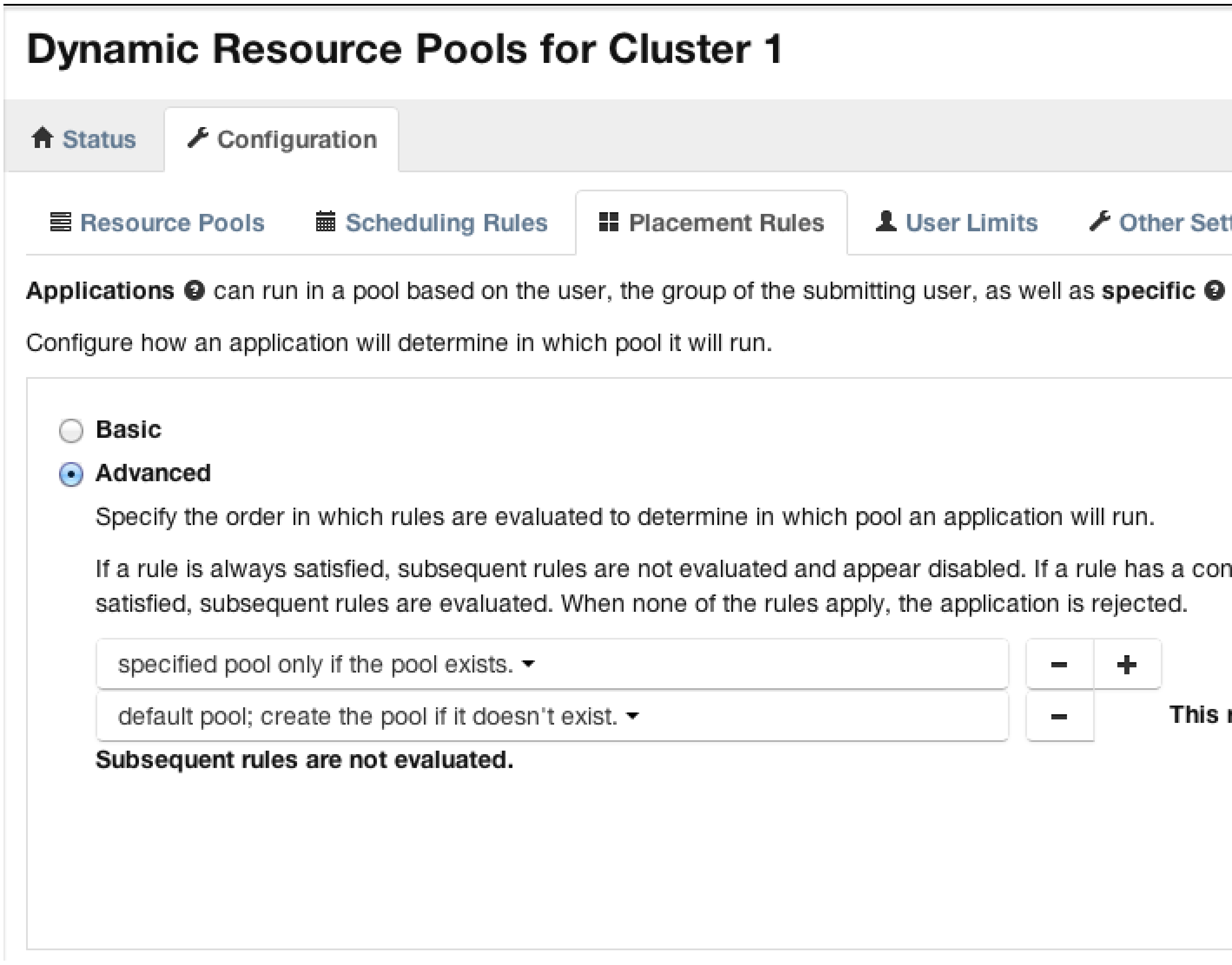
Allocate resources across pools using weights, minimum, and maximum limits. Configuration sets allow switc settings activated by user-defined schedules.

Pools can be nested, each level of which can support a different scheduler, such as FIFO or fair scheduler. E allow only a certain set of users and groups to access the pool.

Name	YARN						Imp	
	Weight	%	Virtual Cores Min / Max	Memory Min / Max	Max Running Apps	Scheduling Policy	Max Memory	Ru Qu
root	1	100.0%	- / -	- / -	-	DRF	-	
default	1	33.3%	- / -	- / -	-	DRF	50000MB	
development	1	33.3%	- / -	- / -	-	DRF	200000MB	
production	1	33.3%	- / -	- / -	-	DRF	1000000MB	

Figure 1: Sample Settings for Cloudera Manager Dynamic Resource Pools Page

The following figure shows a sample of the **Placement Rules** page in Cloudera Manager, accessed through the **Clusters** > ***ClusterName*** > **Other** > **Dynamic Resource Pools** > **Configuration** > **Placement Rules** menu choice. The settings demonstrate a reasonable configuration of a pool named `default` to service all requests where the specified resource pool does not exist, is not explicitly set, or the user or group is not authorized for the specified pool.



Dynamic Resource Pools for Cluster 1

Status | Configuration

Resource Pools | Scheduling Rules | Placement Rules | User Limits | Other Sett

Applications can run in a pool based on the user, the group of the submitting user, as well as **specific**

Configure how an application will determine in which pool it will run.

Basic

[✓ Advanced]

Specify the order in which rules are evaluated to determine in which pool an application will run.

If a rule is always satisfied, subsequent rules are not evaluated and appear disabled. If a rule has a con
satisfied, subsequent rules are evaluated. When none of the rules apply, the application is rejected.

specified pool only if the pool exists.

default pool; create the pool if it doesn't exist. **This r**

Subsequent rules are not evaluated.

Figure 2: Sample Settings for Cloudera Manager Placement Rules Page

For clusters not managed by Cloudera Manager, here are sample `fair-scheduler.xml` and `llama-site.xml` files that define resource pools equivalent to the ones in the preceding Cloudera Manager dialog. These sample files are stripped down: in a real deployment they might contain other settings for use with various aspects of the YARN and Llama components. The settings shown here are the significant ones for the Impala admission control feature.

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>llama.am.throttling.maximum.placed.reservations.root.development</name>
    <value>50</value>
  </property>
  <property>
    <name>llama.am.throttling.maximum.queued.reservations.root.development</name>
    <value>100</value>
  </property>
  <property>
    <name>llama.am.throttling.maximum.placed.reservations.root.production</name>
    <value>100</value>
  </property>
  <property>
    <name>llama.am.throttling.maximum.queued.reservations.root.production</name>
    <value>200</value>
  </property>
</configuration>
```

Guidelines for Using Admission Control

To see how admission control works for particular queries, examine the profile output for the query. This 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.

Where practical, use Cloudera Manager 5 to configure the admission control parameters. The Cloudera Manager GUI is much simpler than editing the configuration files directly. In Cloudera Manager 4, the admission control settings are not available directly, but you can use the `impalad` safety valve field to configure appropriate startup options.

Remember that the limits imposed by admission control are "soft" limits. Although the limits you specify for number of concurrent queries and amount of memory apply cluster-wide, the decentralized nature of this mechanism means that each Impala 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 as a fallback, the overall Impala memory limit and the Linux cgroups mechanism serve as hard limits to prevent overallocation of memory, by cancelling queries if necessary.

If you have trouble getting a query to run because its estimated memory usage is too high, you can override the estimate by setting the `MEM_LIMIT` query option in `impala-shell`, then issuing the query through the shell in the same session. The `MEM_LIMIT` value is treated as the estimated amount of memory, overriding the estimate that Impala would generate based on table and column statistics. This value is used only for making admission control decisions, and is not pre-allocated by the query.

In `impala-shell`, you can also specify which resource pool to direct queries to by setting the `REQUEST_POOL` query option. (This option was named `YARN_POOL` during the CDH 5 beta period.)

The statements affected by the admission control feature are primarily queries, but also include statements that write data such as `INSERT` and `CREATE TABLE AS SELECT`. Most write operations in Impala are not resource-intensive, but inserting into a Parquet table can require substantial memory due to buffering a substantial amount of data before writing out each Parquet data block. See Loading Data into Parquet Tables on page 247 for instructions about inserting data efficiently into Parquet tables.

Although admission control does not scrutinize memory usage for other kinds of DDL statements, if a query is queued due to a limit on concurrent queries or memory usage, subsequent statements in the same session are also queued so that they are processed in the correct order:

```
-- This query could be queued to avoid out-of-memory at times of heavy load.
select * from huge_table join enormous_table using (id);
-- If so, this subsequent statement in the same session is also queued
-- until the previous statement completes.
drop table huge_table;
```

If you set up different resource pools for different users and groups, consider reusing any classifications and hierarchy you developed for use with Sentry security. See Enabling Sentry Authorization for Impala for details. For details about all the Fair Scheduler configuration settings, see the Apache wiki, in particular the tags such as <queue> and <`aclSubmitApps`> to map users and groups to particular resource pools (queues).

Using YARN Resource Management with Impala (CDH 5 Only)

You can limit the CPU and memory resources used by Impala, to manage and prioritize workloads on clusters that run jobs from many Hadoop components. (Currently, there is no limit or throttling on the I/O for Impala queries.) In CDH 5, Impala can use the underlying Apache Hadoop YARN resource management framework, which allocates the required resources for each Impala query. Impala estimates the resources required by the query on each node of the cluster, and requests the resources from YARN.

Requests from Impala to YARN go through an intermediary service called Llama (Long-Lived Application Master). When the resource requests are granted, Impala starts the query and places all relevant execution threads into the CGroup containers and sets up the memory limit on each node. If sufficient resources are not available, the Impala query waits until other jobs complete and the resources are freed. During query processing, as the need for additional resources arises, Llama can “expand” already-requested resources, to avoid over-allocating at the start of the query.

After a query is finished, Llama caches the resources (for example, leaving memory allocated) in case they are needed for subsequent Impala queries. This caching mechanism avoids the latency involved in making a whole new set of resource requests for each query. If the resources are needed by YARN for other types of jobs, Llama returns them.

While the delays to wait for resources might make individual queries seem less responsive on a heavily loaded cluster, the resource management feature makes the overall performance of the cluster smoother and more predictable, without sudden spikes in utilization due to memory paging, CPUs pegged at 100%, and so on.

The Llama Daemon

Llama is a system that mediates resource management between Cloudera Impala and Hadoop YARN. Llama enables Impala to reserve, use, and release resource allocations in a Hadoop cluster. Llama is only required if resource management is enabled in Impala.

By default, YARN allocates resources bit-by-bit as needed by MapReduce jobs. Impala needs all resources available at the same time, so that intermediate results can be exchanged between cluster nodes, and queries do not stall partway through waiting for new resources to be allocated. Llama is the intermediary process that ensures all requested resources are available before each Impala query actually begins.

For Llama installation instructions, see Llama installation.

For management through Cloudera Manager, see Adding the Llama Role.

Controlling Resource Estimation Behavior

By default, Impala consults the table statistics and column statistics for each table in a query, and uses those figures to construct estimates of needed resources for each query. See COMPUTE STATS Statement on page 84 for the statement to collect table and column statistics for a table.

To avoid problems with inaccurate or missing statistics, which can lead to inaccurate estimates of resource consumption, Impala allows you to set default estimates for CPU and memory resource consumption. As a query grows to require more resources, Impala will request more resources from Llama (this is called "expanding" a query reservation). When the query is complete, those resources are returned to YARN as normal. To enable this feature, use the command-line option `-rm_always_use_defaults` when starting the `impalad` daemon, and optionally `-rm_default_memory=`*`size`* and `-rm_default_cpu_cores`. Cloudera recommends always running with `-rm_always_use_defaults` enabled when using resource management, because if the query needs more resources than the default values, the resource requests are expanded dynamically as the query runs. See impalad Startup Options for Resource Management on page 43 for details about each option.

Checking Resource Estimates and Actual Usage

To make resource usage easier to verify, the output of the `EXPLAIN` SQL statement now includes information about estimated memory usage, whether table and column statistics are available for each table, and the number of virtual cores that a query will use. You can get this information through the `EXPLAIN` statement without actually running the query. The extra information requires setting the query option `EXPLAIN_LEVEL=verbose`; see EXPLAIN Statement on page 103 for details. The same extended information is shown at the start of the output from the `PROFILE` statement in `impala-shell`. The detailed profile information is only available after running the query. You can take appropriate actions (gathering statistics, adjusting query options) if you find that queries fail or run with suboptimal performance when resource management is enabled.

How Resource Limits Are Enforced

- CPU limits are enforced by the Linux CGroups mechanism. YARN grants resources in the form of containers that correspond to CGroups on the respective machines.
- Memory is enforced by Impala's query memory limits. Once a reservation request has been granted, Impala sets the query memory limit according to the granted amount of memory before executing the query.

Enabling Resource Management for Impala

To enable resource management for Impala, first you set up the YARN and Llama services for your CDH cluster. Then you add startup options and customize resource management settings for the Impala services.

Required CDH Setup for Resource Management with Impala

YARN is the general-purpose service that manages resources for many Hadoop components within a CDH cluster. Llama is a specialized service that acts as an intermediary between Impala and YARN, translating Impala resource requests to YARN and coordinating with Impala so that queries only begin executing when all needed resources have been granted by YARN.

For information about setting up the YARN and Llama services, see the instructions for YARN and Llama in the *CDH 5 Installation Guide*.

Using Impala with a Llama High Availability Configuration

Impala can take advantage of the Llama high availability feature, with additional Llama servers that step in if the primary one becomes unavailable. (Only one Llama server at a time services all resource requests.) Before using this feature from Impala, read the background information about Llama HA, its main features, and how to set it up.

Setting up the Impala side in a Llama HA configuration involves setting the `impalad` configuration options `-llama_addresses` (mandatory) and optionally `-llama_max_request_attempts`, `-llama_registration_timeout_secs`, and `-llama_registration_wait_secs`. See impalad Startup Options for Resource Management on page 43 for usage instructions for those options.

The `impalad` daemon on the coordinator node registers with the Llama server for each query, receiving a handle that is used for subsequent resource requests. If a Llama server becomes unavailable, all running Impala queries are cancelled. Subsequent queries register with the next specified Llama server. This registration only happens when a query or similar request causes an `impalad` to request resources through Llama. Therefore, when a Llama server becomes unavailable, that fact might not be reported immediately in the Impala status information such as the `metrics` page in the debug web UI.

impalad Startup Options for Resource Management

The following startup options for `impalad` enable resource management and customize its parameters for your cluster configuration:

- `-enable_rm`: Whether to enable resource management or not, either `true` or `false`. The default is `false`. None of the other resource management options have any effect unless `-enable_rm` is turned on.
- `-llama_host`: Hostname or IP address of the Llama service that Impala should connect to. The default is `127.0.0.1`.
- `-llama_port`: Port of the Llama service that Impala should connect to. The default is 15000.
- `-llama_callback_port`: Port that Impala should start its Llama callback service on. Llama reports when resources are granted or preempted through that service.
- `-cgroup_hierarchy_path`: Path where YARN and Llama will create CGroups for granted resources. Impala assumes that the CGroup for an allocated container is created in the path '*cgroup_hierarchy_path* + *container_id*'.
- `-rm_always_use_defaults`: If this Boolean option is enabled, Impala ignores computed estimates and always obtains the default memory and CPU allocation from Llama at the start of the query. These default estimates are approximately 2 CPUs and 4 GB of memory, possibly varying slightly depending on cluster size, workload, and so on. Cloudera recommends enabling `-rm_always_use_defaults` whenever resource management is used, and relying on these default values (that is, leaving out the two following options).
- `-rm_default_memory=*size*`: Optionally sets the default estimate for memory usage for each query. You can use suffixes such as MB and GB, MEM_LIMIT query option. Only has an effect when `-rm_always_use_defaults` is also enabled.
- `-rm_default_cpu_cores`: Optionally sets the default estimate for number of virtual CPU cores for each query. Only has an effect when `-rm_always_use_defaults` is also enabled.

The following options fine-tune the interaction of Impala with Llama when Llama high availability (HA) is enabled. The `-llama_addresses` option is only applicable in a Llama HA environment. `-llama_max_request_attempts`, `-llama_registration_timeout_secs`, and `-llama_registration_wait_secs` work whether or not Llama HA is enabled, but are most useful in combination when Llama is set up for high availability.

- `-llama_addresses`: Comma-separated list of *`hostname`*`:`*`port`* items, specifying all the members of the Llama availability group. Defaults to `"127.0.0.1:15000"`.

- `-llama_max_request_attempts`: Maximum number of times a request to reserve, expand, or release resources is retried until the request is cancelled. Attempts are only counted after Impala is registered with Llama. That is, a request survives at most`llama_max_request_attempts-1` re-registrations. Defaults to 5.
- `-llama_registration_timeout_secs`: Maximum number of seconds that Impala will attempt to register or re-register with Llama. If registration is unsuccessful, Impala cancels the action with an error, which could result in an `impalad` startup failure or a cancelled query. A setting of -1 means try indefinitely. Defaults to 30.
- `-llama_registration_wait_secs`: Number of seconds to wait between attempts during Llama registration. Defaults to 3.

impala-shell Query Options for Resource Management

Before issuing SQL statements through the `impala-shell` interpreter, you can use the `SET` command to configure the following parameters related to resource management:

- EXPLAIN_LEVEL on page 194
- MEM_LIMIT on page 200
- RESERVATION_REQUEST_TIMEOUT (CDH 5 Only) on page 202
- V_CPU_CORES (CDH 5 Only) on page 202

Limitations of Resource Management for Impala

Currently, Impala in CDH 5 has the following limitations for resource management of Impala queries:

- Table statistics are required, and column statistics are highly valuable, for Impala to produce accurate estimates of how much memory to request from YARN. See Table Statistics on page 212 and Column Statistics on page 213 for instructions on gathering both kinds of statistics, and EXPLAIN Statement on page 103 for the extended `EXPLAIN` output where you can check that statistics are available for a specific table and set of columns.
- If the Impala estimate of required memory is lower than is actually required for a query, Impala dynamically expands the amount of requested memory. Queries might still be cancelled if the reservation expansion fails, for example if there are insufficient remaining resources for that pool, or the expansion request takes long enough that it exceeds the query timeout interval, or because of YARN preemption. You can see the actual memory usage after a failed query by issuing a `PROFILE` command in `impala-shell`. Specify a larger memory figure with the `MEM_LIMIT` query option and re-try the query.
- The `MEM_LIMIT` query option, and the other resource-related query options, are not currently settable through the ODBC or JDBC interfaces.

Setting Timeout Periods for Daemons, Queries, and Sessions

Depending on how busy your CDH cluster is, you might increase or decrease various timeout values.

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
```

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. Specify the following startup options for the `impalad` daemon:

- The `--idle_query_timeout` option 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 the query is cancelled, the client program cannot retrieve any further results.
- The `--idle_session_timeout` option specifies the time in seconds after which an idle session is expired. 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 means that sessions never expire.

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

Using Impala through a Proxy for High Availability

For most clusters that have multiple users and production availability requirements, you might set up a proxy server to relay requests to and from Impala. This configuration has the following advantages:

- Applications connect to a single well-known host and port, rather than keeping track of the hosts where the `impalad` daemon is running.
- If any host running the `impalad` daemon becomes unavailable, application connection requests will still succeed because you always connect to the proxy server.
- 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 issue queries using round-robin scheduling, so that each connection uses a different coordinator node. This load-balancing technique lets the Impala nodes share this additional work, rather than concentrating it on a single machine.

The following setup steps are a general outline that apply to any load-balancing proxy software.

1. Download the load-balancing proxy software. It should only need to be installed and configured on a single host.
2. Configure the software (typically by editing a configuration file). Set up a port that the load balancer will listen on to relay Impala requests back and forth.
3. Specify the host and port settings for each Impala node. These are the hosts that the load balancer will choose from when relaying each Impala query. See Appendix A - Ports Used by Impala on page 279 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.

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, to prevent man-in-the-middle attacks. To clarify that the load-balancing proxy server is legitimate, perform these extra Kerberos setup steps:

1. This section assumes you are starting with a Kerberos-enabled cluster. See Enabling Kerberos Authentication for Impala for instructions for setting up Impala with Kerberos. See the *CDH Security Guide* for general steps to set up Kerberos: CDH 4 instructions or CDH 5 instructions.
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. If not, go back over the initial Kerberos configuration steps. to the keytab on each host running the `impalad` daemon.

3. Copy the keytab file from the proxy host to all other hosts in the cluster that run the `impalad` daemon. (For optimal performance, `impalad` should be running on all DataNodes in the cluster.) Put the keytab file in a secure location on each of these other hosts.
4. On systems not managed by Cloudera Manager, add an entry `impala/actual_hostname@realm` to the keytab on each host running the `impalad` daemon.
5. For each impalad node, merge the existing keytab with the proxy's keytab using `ktutil`, producing a new keytab file. For example:

```
$ ktutil
ktutil: read_kt proxy.keytab
ktutil: read_kt impala.keytab
ktutil: write_kt proxy_impala.keytab
ktutil: quit
```

6. Make sure that the `impala` user has permission to read this merged keytab file.
7.
   - Change some configuration settings for each host in the cluster that participates in the load balancing. In the `impalad` option definition, or the Cloudera Manager safety valve (Cloudera Manager 4) or advanced configuration snippet (Cloudera Manager 5), add:

   ```
   --principal=impala/proxy_host@realm
   --be_principal=impala/actual_host@realm
   --keytab_file=path_to_merged_keytab
   ```

   Note: Every host has a different `--be_principal` because the actual host name is different on each host.

   - On a cluster managed by Cloudera Manager, create a role group to set the configuration values from the preceding step on a per-host basis.
   - On a cluster not managed by Cloudera Manager, see Modifying Impala Startup Options for the procedure to modify the startup options.
8.
   - On a cluster managed by Cloudera Manager, restart the Impala service.
   - On a cluster not managed by Cloudera Manager, restart the `impalad` daemons on all hosts in the cluster, as well as the `statestored` and `catalogd` daemons.

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 below for a sample configuration file for one particular load balancer (HAProxy).
- 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 *haproxy_host*`:25003`, rather than port 25000 on a host actually running `impalad`.

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.
#---------------------------------------------------------------------
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
    contimeout 5000
    clitimeout 50000
    srvtimeout 50000

#
# 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
    server symbolic_name_2 impala-host-2.example.com:21000
    server symbolic_name_3 impala-host-3.example.com:21000
    server symbolic_name_4 impala-host-4.example.com:21000
```

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 239 for details about file formats, especially Using the Parquet File Format with Impala Tables on page 246. See CREATE TABLE Statement on page 90 and ALTER TABLE Statement on page 79 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 96 for details.
  - For Impala-managed ("internal") tables, use `DROP TABLE` statements to remove data files. See DROP TABLE Statement on page 101 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 116 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 74 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 114 for details.
- Make sure that 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 101 and the FAQ entry, Why is space not freed up when I issue DROP TABLE? in the SQL section for details. See User Account Requirements 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 100 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` 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 operations are stored in the directory `/tmp/impala-scratch`. These files are removed when the sort operation finishes. (Multiple concurrent queries can perform `ORDER BY` queries that use the external sort 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. The scratch directory 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. Impala will not start if it cannot create or read and write files in the "scratch" directory. 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.

Impala SQL Language Reference

Cloudera 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`.

Most HiveQL `SELECT` and `INSERT` statements run unmodified with Impala. See SQL Differences Between Impala and Hive on page 177 for the current differences.

For full details about Impala SQL syntax and semantics, see SQL Statements on page 78. For information about Hive syntax not available in Impala, see SQL Differences Between Impala and Hive on page 177. For a list of the built-in functions available in Impala queries, see Built-in Functions on page 138.

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 63.

See SQL Differences Between Impala and Hive on page 177 for differences between Impala and Hive data types.

BIGINT Data Type

An 8-byte integer data type used in `CREATE TABLE` and `ALTER TABLE` statements.

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 value `N` to `TIMESTAMP` produces a value that is `N` seconds past the start of the epoch date (January 1, 1970).

Examples:

```
CREATE TABLE t1 (x BIGINT);
SELECT CAST(1000 AS BIGINT);
```

Related information: Literals on page 63, INT Data Type on page 59, SMALLINT Data Type on page 60, TINYINT Data Type on page 62, Mathematical Functions on page 139

BOOLEAN Data Type

A data type used in `CREATE TABLE` and `ALTER TABLE` statements, representing a single true/false choice.

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`. You can use `CAST()` to convert any integer or float-point type to `BOOLEAN`: a value of 0 represents `false`, and any non-zero value is converted to `true`. 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.

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
```

Related information: Literals on page 63, Conditional Functions on page 151

DECIMAL Data Type (CDH 5.1 or higher only)

A numeric data type with fixed scale and precision, used in `CREATE TABLE` and `ALTER TABLE` statements. Suitable for financial and other arithmetic calculations where the imprecise representation and rounding behavior of `FLOAT` and `DOUBLE` make those types impractical.

Syntax:

```
DECIMAL[(precision[,scale])]
```

`DECIMAL` with no precision or scale values is equivalent to `DECIMAL(9,0)`.

Precision and Scale:

precision represents the total number of digits that can be represented by the column, regardless of the location of the decimal point. This value must be between 1 and 38. For example, representing integer values up to 9999, and floating-point values up to 99.99, both require a precision of 4. You can also represent corresponding negative values, without any change in the precision. For example, the range -9999 to 9999 still only requires a precision of 4.

scale represents the number of fractional digits. This value must be less than or equal to *precision*. A scale of 0 produces integral values, with no fractional part. If precision and scale are equal, all the digits come after the decimal point, making all the values between 0 and 0.999... or 0 and -0.999...

When *precision* and *scale* are omitted, a `DECIMAL` value is treated as `DECIMAL(9,0)`, that is, an integer value ranging from `-999,999,999` to `999,999,999`. This is the largest `DECIMAL` value that can still be represented in 4 bytes. If precision is specified but scale is omitted, Impala uses a value of zero for the scale.

Both *precision* and *scale* must be specified as integer literals, not any other kind of constant expressions.

To check the precision or scale for arbitrary values, you can call the `precision()` and `scale()` built-in functions. For example, you might use these values to figure out how many characters are required for various fields in a report, or to understand the rounding characteristics of a formula as applied to a particular `DECIMAL` column.

Range:

The maximum precision value is 38. Thus, the largest integral value is represented by `DECIMAL(38,0)` (999... with 9 repeated 38 times). 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).

For a given precision and scale, the range of `DECIMAL` values is the same in the positive and negative directions. For example, `DECIMAL(4,2)` can represent from -99.99 to 99.99. This is different from other integral numeric types where the positive and negative bounds differ slightly.

When you use `DECIMAL` values in arithmetic expressions, the precision and scale of the result value are determined as follows:

- For addition and subtraction, the precision and scale are based on the maximum possible result, that is, if all the digits of the input values were 9s and the absolute values were added together.
- For multiplication, the precision is the sum of the precisions of the input values. The scale is the sum of the scales of the input values.
- For division, Impala sets the precision and scale to values large enough to represent the whole and fractional parts of the result.
- For `UNION`, the scale is the larger of the scales of the input values, and the precision is increased if necessary to accommodate any additional fractional digits. If the same input value has the largest precision and the largest scale, the result value has the same precision and scale. If one value has a larger precision but smaller scale, the scale of the result value is increased. For example, `DECIMAL(20,2) UNION DECIMAL(8,6)` produces a result of type `DECIMAL(24,6)`. The extra 4 fractional digits of scale (6-2) are accommodated by extending the precision by the same amount (20+4).
- To doublecheck, you can always call the `PRECISION()` and `SCALE()` functions on the results of an arithmetic expression to see the relevant values, or use a `CREATE TABLE AS SELECT` statement to define a column based on the return type of the expression.

Compatibility:

- Using the `DECIMAL` type is only supported under CDH 5.1.0 and higher.
- 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 or negative values for precision.

Conversions and casting:

Impala automatically converts between DECIMAL and other numeric types where possible. A DECIMAL with zero scale is converted to or from the smallest appropriate integral type. A DECIMAL with a fractional part is automatically converted to or from the smallest appropriate floating-point type. If the destination type does not have sufficient precision or scale to hold all possible values of the source type, Impala raises an error and does not convert the value.

For example, these statements show how expressions of DECIMAL and other types are reconciled to the same type in the context of UNION queries and INSERT statements:

```
[localhost:21000] > select cast(1 as int) as x union select cast(1.5 as decimal(9,4))
 as x;
+----------------+
| x              |
+----------------+
| 1.5000         |
| 1.0000         |
+----------------+
[localhost:21000] > create table int_vs_decimal as select cast(1 as int) as x union
select cast(1.5 as decimal(9,4)) as x;
+-------------------+
| summary           |
+-------------------+
| Inserted 2 row(s) |
+-------------------+
[localhost:21000] > desc int_vs_decimal;
+------+---------------+---------+
| name | type          | comment |
+------+---------------+---------+
| x    | decimal(14,4) |         |
+------+---------------+---------+
```

To avoid potential conversion errors, you can use CAST() to convert DECIMAL values to FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING, TIMESTAMP, or BOOLEAN. You can use exponential notation in DECIMAL literals or when casting from STRING, for example 1.0e6 to represent one million.

If you cast a value with more fractional digits than the scale of the destination type, any extra fractional digits are truncated (not rounded). Casting a value to a target type with not enough precision produces a result of NULL and displays a runtime warning.

```
[localhost:21000] > select cast(1.239 as decimal(3,2));
+-----------------------------+
| cast(1.239 as decimal(3,2)) |
+-----------------------------+
| 1.23                        |
+-----------------------------+
[localhost:21000] > select cast(1234 as decimal(3));
+----------------------------+
| cast(1234 as decimal(3,0)) |
+----------------------------+
| NULL                       |
+----------------------------+
WARNINGS: Expression overflowed, returning NULL
```

When you specify integer literals, for example in INSERT ... VALUES statements or arithmetic expressions, those numbers are interpreted as the smallest applicable integer type. You must use CAST() calls for some combinations of integer literals and DECIMAL precision. For example, INT has a maximum value that is 10 digits long, TINYINT has a maximum value that is 3 digits long, and so on. If you specify a value such as 123456 to go into a DECIMAL column, Impala checks if the column has enough precision to represent the largest value of that integer type, and raises an error if not. Therefore, use an expression like CAST(123456 TO DECIMAL(9,0)) for DECIMAL columns with precision 9 or less, CAST(50 TO DECIMAL(2,0)) for DECIMAL columns with precision 2 or less, and so on. For DECIMAL columns with precision 10 or greater, Impala automatically interprets the value as the correct DECIMAL type; however, because DECIMAL(10) requires 8 bytes of storage while DECIMAL(9) requires only 4 bytes, only use precision of 10 or higher when actually needed.

```
[localhost:21000] > create table decimals_9_0 (x decimal);
[localhost:21000] > insert into decimals_9_0 values (1), (2), (4), (8), (16), (1024),
 (32768), (65536), (1000000);
ERROR: AnalysisException: Possible loss of precision for target table
'decimal_testing.decimals_9_0'.
Expression '1' (type: INT) would need to be cast to DECIMAL(9,0) for column 'x'
[localhost:21000] > insert into decimals_9_0 values (cast(1 as decimal)), (cast(2 as
decimal)), (cast(4 as decimal)), (cast(8 as decimal)), (cast(16 as decimal)), (cast(1024
 as decimal)), (cast(32768 as decimal)), (cast(65536 as decimal)), (cast(1000000 as
decimal));

[localhost:21000] > create table decimals_10_0 (x decimal(10,0));
[localhost:21000] > insert into decimals_10_0 values (1), (2), (4), (8), (16), (1024),
 (32768), (65536), (1000000);
[localhost:21000] >
```

Be aware that in memory and for binary file formats such as Parquet or Avro, DECIMAL(10) or higher consumes 8 bytes while DECIMAL(9) (the default for DECIMAL) or lower consumes 4 bytes. Therefore, to conserve space in large tables, use the smallest-precision DECIMAL type that is appropriate and CAST() literal values where necessary, rather than declaring DECIMAL columns with high precision for convenience.

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() to the desired DECIMAL type:

```
insert into decimals_38_5 values (1), (2), (4), (8), (16), (1024), (32768), (65536),
(1000000),
  (cast("999999999999999999999999999999" as decimal(38,5))),
  (cast(999999999999999999999999999999. as decimal(38,5)));
```

- The result of an aggregate function such as MAX(), SUM(), or AVG() on DECIMAL values is promoted to a scale of 38, with the same precision as the underlying column. Thus, the result can represent the largest possible value at that particular precision.
- STRING columns, literals, or expressions can be converted to DECIMAL as long as the overall number of digits and digits to the right of the decimal point fit within the specified precision and scale for the declared DECIMAL type. By default, a DECIMAL value with no specified scale or precision can hold a maximum of 9 digits of an integer value. If there are more digits in the string value than are allowed by the DECIMAL scale and precision, the result is NULL.

  The following examples demonstrate how STRING values with integer and fractional parts are represented when converted to DECIMAL. If the precision is 0, the number is treated as an integer value with a maximum of *scale* digits. If the precision is greater than 0, the scale must be increased to account for the digits both to the left and right of the decimal point. As the precision increases, output values are printed with additional trailing zeros after the decimal point if needed. Any trailing zeros after the decimal point in the STRING value must fit within the number of digits specified by the precision.

  ```
  [localhost:21000] > select cast('100' as decimal); -- Small integer value fits
  within 9 digits of scale.
  +-----------------------------+
  | cast('100' as decimal(9,0)) |
  +-----------------------------+
  | 100                         |
  +-----------------------------+
  [localhost:21000] > select cast('100' as decimal(3,0)); -- Small integer value
  fits within 3 digits of scale.
  +-----------------------------+
  | cast('100' as decimal(3,0)) |
  +-----------------------------+
  | 100                         |
  +-----------------------------+
  [localhost:21000] > select cast('100' as decimal(2,0)); -- 2 digits of scale is
  ```

```
not enough!
+------------------------------+
| cast('100' as decimal(2,0)) |
+------------------------------+
| NULL                         |
+------------------------------+
[localhost:21000] > select cast('100' as decimal(3,1)); -- (3,1) = 2 digits left
of the decimal point, 1 to the right. Not enough.
+------------------------------+
| cast('100' as decimal(3,1)) |
+------------------------------+
| NULL                         |
+------------------------------+
[localhost:21000] > select cast('100' as decimal(4,1)); -- 4 digits total, 1 to
the right of the decimal point.
+------------------------------+
| cast('100' as decimal(4,1)) |
+------------------------------+
| 100.0                        |
+------------------------------+
[localhost:21000] > select cast('98.6' as decimal(3,1)); -- (3,1) can hold a 3
digit number with 1 fractional digit.
+-------------------------------+
| cast('98.6' as decimal(3,1)) |
+-------------------------------+
| 98.6                          |
+-------------------------------+
[localhost:21000] > select cast('98.6' as decimal(15,1)); -- Larger scale allows
bigger numbers but still only 1 fractional digit.
+--------------------------------+
| cast('98.6' as decimal(15,1)) |
+--------------------------------+
| 98.6                           |
+--------------------------------+
[localhost:21000] > select cast('98.6' as decimal(15,5)); -- Larger precision
allows more fractional digits, outputs trailing zeros.
+--------------------------------+
| cast('98.6' as decimal(15,5)) |
+--------------------------------+
| 98.60000                       |
+--------------------------------+
[localhost:21000] > select cast('98.60000' as decimal(15,1)); -- Trailing zeros
in the string must fit within 'scale' digits (1 in this case).
+------------------------------------+
| cast('98.60000' as decimal(15,1)) |
+------------------------------------+
| NULL                               |
+------------------------------------+
```

- Most built-in arithmetic functions such as `SIN()` and `COS()` continue to accept only `DOUBLE` values because they are so commonly used in scientific context for calculations of IEEE 954-compliant values. The built-in functions that accept and return `DECIMAL` are:
  - `ABS()`
  - `CEIL()`
  - `COALESCE()`
  - `FLOOR()`
  - `FNV_HASH()`
  - `GREATEST()`
  - `IF()`
  - `ISNULL()`
  - `LEAST()`
  - `NEGATIVE()`
  - `NULLIF()`
  - `POSITIVE()`
  - `PRECISION()`

  - `ROUND()`
  - `SCALE()`
  - `TRUNCATE()`
  - `ZEROIFNULL()`

  See Built-in Functions on page 138 for details.

- `BIGINT`, `INT`, `SMALLINT`, and `TINYINT` values can all be cast to `DECIMAL`. The number of digits to the left of the decimal point in the `DECIMAL` type must be sufficient to hold the largest value of the corresponding integer type. Note that integer literals are treated as the smallest appropriate integer type, meaning there is sometimes a range of values that require one more digit of `DECIMAL` scale than you might expect. For integer values, the precision of the `DECIMAL` type can be zero; if the precision is greater than zero, remember to increase the scale value by an equivalent amount to hold the required number of digits to the left of the decimal point.

  The following examples show how different integer types are converted to `DECIMAL`.

```
[localhost:21000] > select cast(1 as decimal(1,0));
+-------------------------+
| cast(1 as decimal(1,0)) |
+-------------------------+
| 1                       |
+-------------------------+
[localhost:21000] > select cast(9 as decimal(1,0));
+-------------------------+
| cast(9 as decimal(1,0)) |
+-------------------------+
| 9                       |
+-------------------------+
[localhost:21000] > select cast(10 as decimal(1,0));
+--------------------------+
| cast(10 as decimal(1,0)) |
+--------------------------+
| 10                       |
+--------------------------+
[localhost:21000] > select cast(10 as decimal(1,1));
+--------------------------+
| cast(10 as decimal(1,1)) |
+--------------------------+
| 10.0                     |
+--------------------------+
[localhost:21000] > select cast(100 as decimal(1,1));
+---------------------------+
| cast(100 as decimal(1,1)) |
+---------------------------+
| 100.0                     |
+---------------------------+
[localhost:21000] > select cast(1000 as decimal(1,1));
+----------------------------+
| cast(1000 as decimal(1,1)) |
+----------------------------+
| 1000.0                     |
+----------------------------+
```

- When a `DECIMAL` value is converted to any of the integer types, any fractional part is truncated (that is, rounded towards zero):

```
[localhost:21000] > create table num_dec_days (x decimal(4,1));
[localhost:21000] > insert into num_dec_days values (1), (2), (cast(4.5 as
decimal(4,1)));
[localhost:21000] > insert into num_dec_days values (cast(0.1 as decimal(4,1))),
(cast(.9 as decimal(4,1))), (cast(9.1 as decimal(4,1))), (cast(9.9 as
decimal(4,1)));
[localhost:21000] > select cast(x as int) from num_dec_days;
+----------------+
| cast(x as int) |
+----------------+
| 1              |
| 2              |
| 4               |
| 0               |
| 0               |
| 9               |
| 9               |
+-----------------+
```

- You cannot directly cast `TIMESTAMP` or `BOOLEAN` values to or from `DECIMAL` values. You can turn a `DECIMAL` value into a time-related representation using a two-step process, by converting it to an integer value and then using that result in a call to a date and time function such as `from_unixtime()`.

```
[localhost:21000] > select from_unixtime(cast(cast(1000.0 as decimal) as bigint));
+-------------------------------------------------------------+
| from_unixtime(cast(cast(1000.0 as decimal(9,0)) as bigint)) |
+-------------------------------------------------------------+
| 1970-01-01 00:16:40                                         |
+-------------------------------------------------------------+
[localhost:21000] > select now() + interval cast(x as int) days from num_dec_days;
 -- x is a DECIMAL column.

[localhost:21000] > create table num_dec_days (x decimal(4,1));
[localhost:21000] > insert into num_dec_days values (1), (2), (cast(4.5 as
decimal(4,1)));
[localhost:21000] > select now() + interval cast(x as int) days from num_dec_days;
 -- The 4.5 value is truncated to 4 and becomes '4 days'.
+--------------------------------------+
| now() + interval cast(x as int) days |
+--------------------------------------+
| 2014-05-13 23:11:55.163284000        |
| 2014-05-14 23:11:55.163284000        |
| 2014-05-16 23:11:55.163284000        |
+--------------------------------------+
```

- Because values in `INSERT` statements are checked rigorously for type compatibility, be prepared to use `CAST()` function calls around literals, column references, or other expressions that you are inserting into a `DECIMAL` column.

DECIMAL differences from integer and floating-point types:

With the `DECIMAL` type, you are concerned with the number of overall digits of a number rather than powers of 2 (as in `TINYINT`, `SMALLINT`, and so on). Therefore, the limits with integral values of `DECIMAL` types fall around 99, 999, 9999, and so on rather than 32767, 65535, 2^{32} -1, and so on. For fractional values, you do not need to account for imprecise representation of the fractional part according to the IEEE-954 standard (as in `FLOAT` and `DOUBLE`). Therefore, when you insert a fractional value into a `DECIMAL` column, you can compare, sum, query, `GROUP BY`, and so on that column and get back the the original values rather than some "close but not identical" value.

`FLOAT` and `DOUBLE` 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. `DECIMAL` can help reduce unexpected behavior and rounding errors, at the expense of some performance overhead for assignments and comparisons.

Literals and expressions:

- When you use an integer literal such as `1` or `999` in a SQL statement, depending on the context, Impala will treat it as either the smallest appropriate `DECIMAL` type, or the smallest integer type (`TINYINT`, `SMALLINT`, `INT`, or `BIGINT`). To minimize memory usage, Impala prefers to treat the literal as the smallest appropriate integer type.
- When you use a floating-point literal such as `1.1` or `999.44` in a SQL statement, depending on the context, Impala will treat it as either the smallest appropriate `DECIMAL` type, or the smallest floating-point type (`FLOAT` or `DOUBLE`). To avoid loss of accuracy, Impala prefers to treat the literal as a `DECIMAL`.

Storage considerations:

- Only the precision determines the storage size for `DECIMAL` values; the scale setting has no effect on the storage size.
- Text, RCFile, and SequenceFile tables all use ASCII-based formats. In these text-based file formats, leading zeros are not stored, but trailing zeros are stored. In these tables, each `DECIMAL` value takes up as many bytes as there are digits in the value, plus an extra byte if the decimal point is present and an extra byte for negative values. Once the values are loaded into memory, they are represented in 4, 8, or 16 bytes as described in the following list items. The on-disk representation varies depending on the file format of the table.
-
- Parquet and Avro tables use binary formats, In these tables, Impala stores each value in as few bytes as possible depending on the precision specified for the `DECIMAL` column.
  - In memory, `DECIMAL` values with precision of 9 or less are stored in 4 bytes.
  - In memory, `DECIMAL` values with precision of 10 through 18 are stored in 8 bytes.
  - In memory, `DECIMAL` values with precision greater than 18 are stored in 16 bytes.

File format considerations:

- The `DECIMAL` data type can be stored in any of the file formats supported by Impala, as described in How Impala Works with Hadoop File Formats on page 239. Impala only writes to tables that use the Parquet and text formats, so those formats are the focus for file format compatibility.
- Impala can query Avro, RCFile, or SequenceFile tables containing `DECIMAL` columns, created by other Hadoop components, on CDH 5.1 or higher only.
- You can use `DECIMAL` columns in Impala tables that are mapped to HBase tables. Impala can query and insert into such tables.
- Text, RCFile, and SequenceFile tables all use ASCII-based formats. In these tables, each `DECIMAL` value takes up as many bytes as there are digits in the value, plus an extra byte if the decimal point is present. The binary format of Parquet or Avro files offers more compact storage for `DECIMAL` columns.
- Parquet and Avro tables use binary formats, In these tables, Impala stores each value in 4, 8, or 16 bytes depending on the precision specified for the `DECIMAL` column.
- Parquet files containing `DECIMAL` columns are not expected to be readable under CDH 4. See the **Compatibility** section for details.

UDF considerations: When writing a C++ UDF, use the `DecimalVal` data type defined in `/usr/include/impala_udf/udf.h`.

Partitioning:

You can use a `DECIMAL` column as a partition key. Doing so provides a better match between the partition key values and the HDFS directory names than using a `DOUBLE` or `FLOAT` partitioning column:

Schema evolution:

- For text-based formats (text, RCFile, and SequenceFile tables), you can issue an `ALTER TABLE ... REPLACE COLUMNS` statement to change the precision and scale of an existing `DECIMAL` column. As long as the values in the column fit within the new precision and scale, they are returned correctly by a query. Any values that do not fit within the new precision and scale are returned as `NULL`, and Impala reports the conversion error. Leading zeros do not count against the precision value, but trailing zeros after the decimal point do.

```
[localhost:21000] > create table text_decimals (x string);
[localhost:21000] > insert into text_decimals values ("1"), ("2"), ("99.99"),
("1.234"), ("000001"), ("1.000000000");
[localhost:21000] > select * from text_decimals;
+-------------+
| x           |
+-------------+
| 1           |
| 2           |
| 99.99       |
| 1.234       |
| 000001      |
| 1.000000000 |
+-------------+
[localhost:21000] > alter table text_decimals replace columns (x decimal(4,2));
[localhost:21000] > select * from text_decimals;
+-------+
| x     |
+-------+
| 1.00  |
| 2.00  |
| 99.99 |
| NULL  |
| 1.00  |
| NULL  |
+-------+
ERRORS:
Backend 0:Error converting column: 0 TO DECIMAL(4, 2) (Data is: 1.234)
file:
hdfs://127.0.0.1:8020/user/hive/warehouse/decimal_testing.db/text_decimals/634d4bd3aa0
e8420-b4b13bab7f1be787_56794587_data.0
record: 1.234
Error converting column: 0 TO DECIMAL(4, 2) (Data is: 1.000000000)
file:
hdfs://127.0.0.1:8020/user/hive/warehouse/decimal_testing.db/text_decimals/cd40dc68e20
c565a-cc4bd86c724c96ba_311873428_data.0
record: 1.000000000
```

- 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. (The other columns can still be queried successfully.) 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. If the metadata in the data files disagrees with the metadata in the metastore database, Impala cancels the query.

Examples:

```
CREATE TABLE t1 (x DECIMAL, y DECIMAL(5,2), z DECIMAL(25,0));
INSERT INTO t1 VALUES (5, 99.44, 123456), (300, 6.7, 999999999);
SELECT x+y, ROUND(y,1), z/98.6 FROM t1;
SELECT CAST(1000.5 AS DECIMAL);
```

Restrictions:

Currently, the `COMPUTE STATS` statement under CDH 4 does not store any statistics for `DECIMAL` columns. When Impala runs under CDH 5, which has better support for `DECIMAL` in the metastore database, `COMPUTE STATS` does collect statistics for `DECIMAL` columns and Impala uses the statistics to optimize query performance.

Related information: Literals on page 63 for how numeric literals are sometimes interpreted as `DECIMAL` values; Mathematical Functions on page 139 for the `PRECISION()` and `SCALE()` functions, and other functions whose signatures now include `DECIMAL` arguments or return values

DOUBLE Data Type

An 8-byte (double precision) floating-point data type used in `CREATE TABLE` and `ALTER TABLE` statements.

Range: 4.94065645841246544e-324d .. 1.79769313486231570e+308, positive or negative

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.

The data type `REAL` is an alias for `DOUBLE`.

Examples:

```
CREATE TABLE t1 (x DOUBLE);
SELECT CAST(1000.5 AS DOUBLE);
```

Related information: Literals on page 63, Mathematical Functions on page 139

FLOAT Data Type

A 4-byte (single precision) floating-point data type used in `CREATE TABLE` and `ALTER TABLE` statements.

Range: 1.40129846432481707e-45 .. 3.40282346638528860e+38, positive or negative

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.

Examples:

```
CREATE TABLE t1 (x FLOAT);
SELECT CAST(1000.5 AS FLOAT);
```

Related information: Literals on page 63, Mathematical Functions on page 139

INT Data Type

A 4-byte integer data type used in `CREATE TABLE` and `ALTER TABLE` statements.

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 value `N` to `TIMESTAMP` produces a value that is `N` seconds past the start of the epoch date (January 1, 1970).

The data type `INTEGER` is an alias for `INT`.

Examples:

```
CREATE TABLE t1 (x INT);
SELECT CAST(1000 AS INT);
```

Related information: Literals on page 63, TINYINT Data Type on page 62, BIGINT Data Type on page 50, SMALLINT Data Type on page 60, TINYINT Data Type on page 62, Mathematical Functions on page 139

REAL Data Type

An alias for the `DOUBLE` data type. See DOUBLE Data Type on page 58 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.

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 value `N` to `TIMESTAMP` produces a value that is `N` seconds past the start of the epoch date (January 1, 1970).

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.

Related information: Literals on page 63, TINYINT Data Type on page 62, BIGINT Data Type on page 50, TINYINT Data Type on page 62, INT Data Type on page 59, Mathematical Functions on page 139

STRING Data Type

A data type used in `CREATE TABLE` and `ALTER TABLE` statements.

Length: 32,767 bytes. Do not use any length constraint when declaring `STRING` columns, as you might be familiar with from `VARCHAR`, `CHAR`, or similar column types from relational database systems.

Character sets: For full support in all Impala subsystems, restrict string values to the ASCII character set. UTF-8 character data can be stored in Impala and retrieved through queries, but UTF-8 strings containing non-ASCII characters are not guaranteed to work properly with string manipulation functions, comparison operators, or the `ORDER BY` clause. 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 61 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.

Related information: Literals on page 63, String Functions on page 153, Date and Time Functions on page 146

TIMESTAMP Data Type

A data type used in `CREATE TABLE` and `ALTER TABLE` statements, representing a point in time.

Range: Allowed date values range from 1400-01-01 to 9999-12-31; this range is different from the Hive `TIMESTAMP` type. Internally, the resolution of the time portion of a `TIMESTAMP` value is in nanoseconds.

INTERVAL expressions:

You can perform date arithmetic by adding or subtracting a specified number of time units, using the `INTERVAL` keyword and the + and - operators or `date_add()` and `date_sub()` functions. You can specify units as `YEAR[S]`, `MONTH[S]`, `WEEK[S]`, `DAY[S]`, `HOUR[S]`, `MINUTE[S]`, `SECOND[S]`, `MILLISECOND[S]`, `MICROSECOND[S]`, and `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`.

For example:

```
select now() + interval 1 day;
select date_sub(now(), interval 5 minutes);
insert into auction_details
  select auction_id, auction_start_time, auction_start_time + interval 2 days + interval
 12 hours
   from new_auctions;
```

Time zones: Impala does not store timestamps using the local timezone to avoid undesired results from unexpected time zone issues. Timestamps are stored relative to GMT.

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.sssssssss`, 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'`. You can cast an integer or floating-point value `N` to `TIMESTAMP`, producing a value that is `N` seconds past the start of the epoch date (January 1, 1970).

Note: 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. See Date and Time Functions on page 146 for details.

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 236 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
```

Examples:

```
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 UTC
timezone.

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());
```

Restrictions:

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.

Related information: Literals on page 63; to convert to or from different date formats, or perform date arithmetic, use the date and time functions described in Date and Time Functions on page 146. 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 `TIMESTAMP` value (date plus time, only date, only time, optional fractional seconds).

Related information:

See SQL Differences Between Impala and Hive on page 177 for details about differences in `TIMESTAMP` handling between Impala and Hive.

TINYINT Data Type

A 1-byte integer data type used in `CREATE TABLE` and `ALTER TABLE` statements.

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 value `N` to `TIMESTAMP` produces a value that is `N` seconds past the start of the epoch date (January 1, 1970).

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.

Related information: Literals on page 63, INT Data Type on page 59, BIGINT Data Type on page 50, SMALLINT Data Type on page 60, Mathematical Functions on page 139

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 49 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.

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.

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. This might cause extra line breaks in `impala-shell` output.
- `\r` represents a linefeed. 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 escape the single quote as a `\'` sequence. (The requirement to escape a single quote inside double quotes might be lifted in later releases; if so, the escape character will be optional in that case.)

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 (without quotation marks or 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 Command-Line Options on page 187 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

For `TIMESTAMP` values, 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.sssssssss`, 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'`. You can cast an integer or floating-point value `N` to `TIMESTAMP`, producing a value that is `N` seconds past the start of the epoch date (January 1, 1970).

You can also use `INTERVAL` expressions to add or subtract from timestamp literal values, such as `'1966-07-30' + INTERVAL 5 YEARS + INTERVAL 3 DAYS`. See TIMESTAMP Data Type on page 61 for details.

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 Conditional Functions on page 151 for details.

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.

- `+` 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.

- `%`: 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, 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, 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 Mathematical Functions on page 139 for the arithmetic functions you can use.

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.

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;
```

Comparison Operators

Impala supports the familiar comparison operators for checking equality and sort order for the column data types:

- =, !=, <>: apply to all types.
- <, <=, >, >=: apply to all 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.

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 argument and the set of comparison values must be of compatible types.

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.

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');
```

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.

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.

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;
```

LIKE Operator

A 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.

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 a more general kind of search operator using regular expressions, see REGEXP Operator on page 70.

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.

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.)

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.

The regular expression must match the entire value, not just occur somewhere inside it. Use `.*` at the beginning and/or the end if you only need to match characters anywhere in the middle. Thus, the ^ and $ atoms are often redundant, although you might already have them in your expression strings that you reuse from elsewhere.

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.

Note:

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 70 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.

Examples:

```
-- 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.*';

-- 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$';

-- 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$';

-- 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.*$';

-- 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,}$';

-- 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].*';

-- 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.*';
```

RLIKE Operator

Synonym for the `REGEXP` operator.

Schema Objects and Object Names

With Impala, you work 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 SQL Statements on page 78. 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 on page 233 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.

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.

To use an alias name that matches one of the Impala reserved keywords (listed in Appendix C - Impala Reserved Words on page 285), surround the identifier with either single or double quotation marks, or `` characters (backticks).

```
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;
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;
```

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.

Alternatives:

Another way to define different names for the same tables or columns is to create views. See Views on page 74 for details.

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 no database-specific properties to configure. Therefore, there is no `ALTER DATABASE`

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.

Each database is physically represented by a directory in HDFS.

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 87, DROP DATABASE Statement on page 100, USE Statement on page 138

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 Built-in Functions on page 138.

Aggregate functions combine the results from multiple rows. See Aggregate Functions on page 158.

User-defined functions let you code your own logic. They can be either scalar or aggregate functions. See User-Defined Functions (UDFs) on page 163.

Related statements: CREATE FUNCTION Statement on page 88, DROP FUNCTION Statement on page 101

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 alphabetic 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 Appendix C - Impala Reserved Words on page 285), surround the identifier with `` characters (backticks).
- 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 Aliases on page 71 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 Views on page 74 for details.

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 is associated with a directory in HDFS. The table data consists of all the data files underneath that directory:

- Internal tables, managed by Impala, 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.

Related statements: CREATE TABLE Statement on page 90, DROP TABLE Statement on page 101, ALTER TABLE Statement on page 79 INSERT Statement on page 105, LOAD DATA Statement on page 114, SELECT Statement on page 118

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 load data by issuing `INSERT` statements in `impala-shell` or by using the `LOAD DATA` statement in Hive.
- When you issue a `DROP TABLE` statement, Impala physically removes all the data files from the directory.

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.
- 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.

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:

- Set up fine-grained security where a user can query some columns from a table but not other columns. See Controlling Access at the Column Level through Views for details.
- 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.

The SQL statements that configure views are CREATE VIEW Statement on page 95, ALTER VIEW Statement on page 83, and DROP VIEW Statement on page 102. You can specify view names when querying data (SELECT Statement on page 118) and copying data from one table to another (INSERT Statement on page 105). 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:                       | cloudera                     | 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                 |
+------------------------------+------------------------------+----------------------+
Returned 29 row(s) in 0.05s
```

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.

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 is determined by the clauses in the original query. Impala does not prune additional columns if the query on the view includes extra `WHERE` clauses referencing the partition key columns.
- 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 |
+---+
```

Related statements: CREATE VIEW Statement on page 95, ALTER VIEW Statement on page 83, DROP VIEW Statement on page 102

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 79
- ALTER VIEW Statement on page 83
- COMPUTE STATS Statement on page 84
- CREATE DATABASE Statement on page 87
- CREATE FUNCTION Statement on page 88
- CREATE TABLE Statement on page 90
- CREATE VIEW Statement on page 95
- DROP DATABASE Statement on page 100
- DROP FUNCTION Statement on page 101
- DROP TABLE Statement on page 101
- DROP VIEW Statement on page 102

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.

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 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 79) and queries (see SELECT Statement on page 118).

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:

- INSERT Statement on page 105
- LOAD DATA Statement on page 114

`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 265 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.)

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.

Although Impala currently does not have an `UPDATE` statement, you can achieve a similar result 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`.

Related information:

The other major classifications of SQL statements are data definition language (see DDL Statements on page 78) and queries (see SELECT Statement on page 118).

ALTER TABLE Statement

The `ALTER TABLE` statement changes the structure or properties of an existing table. In Impala, this is a logical operation that updates the table metadata in the metastore database that Impala shares with Hive; `ALTER TABLE` does not actually rewrite, move, and so on the actual data files. Thus, you might need to perform corresponding physical filesystem operations, such as moving data files to a different HDFS directory, rewriting the data files to include extra fields, or converting them to a different file format.

Syntax:

```
ALTER TABLE name RENAME TO new_name

ALTER TABLE name ADD COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name new_name new_type
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])

ALTER TABLE name { ADD | DROP } PARTITION (partition_spec)

ALTER TABLE name [PARTITION (partition_spec)]
  SET { FILEFORMAT format
  | LOCATION 'hdfs_path_of_directory'
```

```
   | TBLPROPERTIES (table_properties)
   | SERDEPROPERTIES (serde_properties) }

ALTER TABLE name [PARTITION (partition_spec)] SET { CACHED IN 'pool_name' | UNCACHED
}

new_name ::= [new_database.]new_table_name

col_spec ::= col_name type_name

partition_spec ::= partition_col=constant_value

table_properties ::= 'name'='value'[, 'name'='value' ...]

serde_properties ::= 'name'='value'[, 'name'='value' ...]
```

Statement type: DDL

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.

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 on page 202 for details.

Related information:

CREATE TABLE Statement on page 90, DROP TABLE Statement on page 101, Partitioning on page 233 Internal Tables on page 73, External Tables on page 74

The following sections show examples of the use cases for various `ALTER TABLE` clauses.

To rename a table:

```
ALTER TABLE old_name RENAME TO new_name;
```

For internal tables, his 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 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';
```

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:

```
-- This table begins life as pipe-separated text format.
create table change_to_csv (s1 string, s2 string) row format delimited fields terminated
 by '|';
-- Then we change it to a CSV table.
alter table change_to_csv set SERDEPROPERTIES ('serialization.format'=',',
'field.delim'=',');
insert overwrite change_to_csv values ('stop','go'), ('yes','no');
!hdfs dfs -cat 'hdfs://hostname:8020/data_directory/dbname.db/change_to_csv/data_file';
stop,go
yes,no
```

Use the `DESCRIBE FORMATTED` statement to see the current values of these properties for an existing table. See CREATE TABLE Statement on page 90 for more details about these clauses. See Setting Statistics Manually through ALTER TABLE on page 213 for an example of using table properties to fine-tune the performance-related table statistics.

To reorganize columns for a table:

```
ALTER TABLE table_name ADD COLUMNS (column_defs);
ALTER TABLE table_name REPLACE COLUMNS (column_defs);
ALTER TABLE table_name CHANGE column_name new_name new_type;
ALTER TABLE table_name DROP column_name;
```

The *column_spec* is the same as in the `CREATE TABLE` statement: the column name, then its data type, then an optional comment. You can add multiple columns at a time. The parentheses are required whether you add a single column or multiple columns. When you replace columns, all the original column definitions are discarded. You might use this technique 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.)

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.

To change the file format that Impala expects data to be in, for a table or partition:

```
ALTER TABLE table_name [PARTITION (partition_spec)] SET FILEFORMAT { PARQUET | TEXTFILE
 | RCFILE | SEQUENCEFILE }
```

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 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.

```
-- 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.

Cancellation: Cannot be cancelled.

ALTER VIEW Statement

Changes the query associated with a view, or the associated database and/or name of the view.

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:

```
ALTER VIEW [database_name.]view_name AS select_statement
ALTER VIEW [database_name.]view_name RENAME TO [database_name.]view_name
```

Statement type: DDL

Related information:

Views on page 74, CREATE VIEW Statement on page 95, DROP VIEW Statement on page 102

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 on page 202 for details.

Examples:

```
create table t1 (x int, y int, s string);
create table t2 like t1;
create view v1 as select * from t1;
alter view v1 as select * from t2;
alter view v1 as select x, upper(s) s from t2;
```

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:                       | cloudera                     | 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
|
+------------------------------+------------------------------+----------------------+
Returned 29 row(s) in 0.05s
```

Cancellation: Cannot be cancelled.

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 parallelize the work appropriately for a join query or insert operation. For details about the kinds of information gathered by this statement, see Table Statistics on page 212.

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 make use of the `NULL` count during query planning, Impala 1.4.0 and higher speeds up the `COMPUTE STATS` statement by skipping this `NULL` counting.

Statement type: DDL

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 is built from the ground up 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.

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.

For related information, see SHOW Statement on page 135, Table Statistics on page 212, and Column Statistics on page 213.

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.

Performance considerations: The statistics collected by `COMPUTE STATS` are used to optimize join queries and resource-intensive `INSERT` operations.

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     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
```

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 239 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.

> **Note:** Currently, a known issue (IMPALA-488) could cause excessive memory usage during a `COMPUTE STATS` operation on a Parquet table. As a workaround, issue the command `SET NUM_SCANNER_THREADS=2` in `impala-shell` before issuing the `COMPUTE STATS` statement. Then issue `UNSET NUM_SCANNER_THREADS` before continuing with queries.

The `COMPUTE STATS` statement works with Avro tables, as long as they are created with SQL-style column names and types rather than an Avro-style schema specification. These tables are currently always created through Hive rather than Impala.

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.

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...

Restrictions:

Currently, the `COMPUTE STATS` statement under CDH 4 does not store any statistics for `DECIMAL` columns. When Impala runs under CDH 5, which has better support for `DECIMAL` in the metastore database, `COMPUTE STATS` does collect statistics for `DECIMAL` columns and Impala uses the statistics to optimize query performance.

CREATE DATABASE Statement

In Impala, a database is both:

- A logical construct for grouping together related tables 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 back it up, measure space usage, or remove it (if it is empty) with a `DROP DATABASE` statement.

Syntax:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name[COMMENT 'database_comment']
  [LOCATION hdfs_path];
```

Statement type: DDL

Related information:

Databases on page 72, DROP DATABASE Statement on page 100, USE Statement on page 138

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.

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 on page 202 for details.

Examples:

```
create database first;
use first;
create table t1 (x int);

create database second;
use second;
-- 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 do not have to USE a database after creating it.
-- Just qualify the table name with the name of the 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.
use default;
-- Dropping the database is a fast way to drop all the tables within it.
drop database temp;
```

Cancellation: Cannot be cancelled.

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.

To create a scalar UDF, issue a `CREATE FUNCTION` statement:

```
CREATE FUNCTION [IF NOT EXISTS] [db_name.]function_name([arg_type[, arg_type...])
  RETURNS return_type
  LOCATION 'hdfs_path'
  SYMBOL='symbol_or_class'
```

To create a UDA, issue a `CREATE AGGREGATE FUNCTION` statement:

```
CREATE [AGGREGATE] FUNCTION [IF NOT EXISTS] [db_name.]function_name([arg_type[,
arg_type...])
  RETURNS return_type
  LOCATION 'hdfs_path'
  [INIT_FN='function]
  UPDATE_FN='function
  MERGE_FN='function
  [PREPARE_FN='function]
  [CLOSEFN='function]
  [FINALIZE_FN='function]
```

Statement type: DDL

Related information:

User-Defined Functions (UDFs) on page 163, DROP FUNCTION Statement on page 101

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`, `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.
- 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.

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 163.

Usage notes:

- 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.)
- 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 `struct`s, 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 directory supplied with Impala.
- 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 on page 202 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++.

Cancellation: Cannot be cancelled.

Related information:

See User-Defined Functions (UDFs) on page 163 for more background information, usage instructions, and examples for Impala UDFs.

CREATE TABLE Statement

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'], ...)]
  [COMMENT 'table_comment']
  [PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [
   [ROW FORMAT row_format] [STORED AS file_format]
  ]
  [LOCATION 'hdfs_path']
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [CACHED IN 'pool_name']
```

```
Column definitions inferred from data file:

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  LIKE PARQUET 'hdfs_path_of_parquet_file'
  [COMMENT 'table_comment']
  [PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [
   [ROW FORMAT row_format] [STORED AS file_format]
  ]
  [LOCATION 'hdfs_path']
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [CACHED IN 'pool_name']
data_type
  : primitive_type
```

```
primitive_type
  : TINYINT
  | SMALLINT
  | INT
  | BIGINT
  | BOOLEAN
  | FLOAT
  | DOUBLE
  | DECIMAL
  | STRING
  | TIMESTAMP

row_format
  : DELIMITED [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
    [LINES TERMINATED BY 'char']

file_format:
    PARQUET
  | TEXTFILE
  | AVRO
  | SEQUENCEFILE
  | RCFILE
```

Statement type: DDL

Related information:

ALTER TABLE Statement on page 79, DROP TABLE Statement on page 101, Partitioning on page 233 Internal Tables on page 73, External Tables on page 74

Internal and external tables:

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 Tables on page 73.

Partitioned tables:

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 on page 233.

Specifying file format:

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 a SequenceFile 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 239.

> **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 255 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 linefeed. When specifying delimiter and line end characters with the `FIELDS TERMINATED BY` and `LINES TERMINATED BY` clauses, use `'\t'` for tab, `'\n'` for carriage return, `'\r'` for linefeed, and `\0` for ASCII `nul` (hex 00). For more examples of text tables, see Using Text Data Files with Impala Tables on page 240.

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 (without quotation marks or 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:

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 239 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 `CREATE TABLE AS SELECT` syntax is as follows:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] db_name.]table_name
  [COMMENT 'table_comment']
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
AS
  select_statement
```

See SELECT Statement on page 118 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.

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 on different data nodes, and therefore the notion of the data being stored in sorted order is impractical.

For example, the following statements show how you can clone all the data in a table, or a subset of the columns and/or rows, or reorder columns, rename them, or construct them out of expressions:

```
-- Create new table and copy all data.
CREATE TABLE clone_of_t1 AS SELECT * FROM t1;
-- Same idea as CREATE TABLE LIKE, don't copy any data.
CREATE TABLE empty_clone_of_t1 AS SELECT * FROM t1 WHERE 1=0;
-- Copy some data.
CREATE TABLE subset_of_t1 AS SELECT * FROM t1 WHERE x > 100 AND y LIKE 'A%';
CREATE TABLE summary_of_t1 AS SELECT c1, sum(c2) AS total, avg(c2) AS average FROM t1
 GROUP BY c2;
-- Switch file format.
CREATE TABLE parquet_version_of_t1 STORED AS PARQUET AS SELECT * FROM t1;
-- Create tables with different column order, names, or types than the original.
CREATE TABLE some_columns_from_t1 AS SELECT c1, c3, c5 FROM t1;
CREATE TABLE reordered_columns_from_t1 AS SELECT c4, c3, c1, c2 FROM t1;
CREATE TABLE synthesized_columns AS SELECT upper(c1) AS all_caps, c2+c3 AS total,
"California" AS state FROM t1;
```

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. Although you can use a partitioned table as the source and copy data from it, you cannot specify any partitioning clauses for the new table.

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.
- The `CREATE TABLE AS SELECT ... STORED AS PARQUET` and `INSERT ... SELECT ... STORED AS PARQUET` statements always create at least one data file in the destination table, 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 246.

Visibility and Metadata:

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. To query the table through the Impala daemon on a different node, issue the `INVALIDATE METADATA` statement first while connected to that other node.

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 265.

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 on different data nodes, 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 caching:

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 5.1 or higher only) on page 218 for details about using the HDFS caching feature.

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...

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_list)]
  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.

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 on page 202 for details.

Examples:

```
create view v1 as select * from t1;
create view v2 as select c1, c3, c7 from t1;
create view v3 as select c1, cast(c3 as string) c3, concat(c4,c5) c5, trim(c6) c6,
"Constant" c8 from t1;
create view v4 as select t1.c1, t2.c2 from t1 join t2 on t1.id = t2.id;
create view some_db.v5 as select * from some_other_db.t1;
```

Cancellation: Cannot be cancelled.

Related information:

Views on page 74, ALTER VIEW Statement on page 83, DROP VIEW Statement on page 102

DESCRIBE Statement

The `DESCRIBE` statement displays metadata about a table, such as the column names and their data types. Its syntax is:

```
DESCRIBE [FORMATTED] table
```

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.

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 host name 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 on page 202 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 135 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:                       | cloudera
               | 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:                       | cloudera                     | 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/cloudera/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:                       | cloudera                                                      |
 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/cloudera/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.

DROP DATABASE Statement

Removes a database from the system, and deletes the corresponding `*.db` directory from HDFS. The database must be empty before it can be dropped, to avoid losing any data.

Syntax:

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name;
```

Statement type: DDL

Related information:

Databases on page 72, CREATE DATABASE Statement on page 87, USE Statement on page 138

Usage notes:

Before dropping a database, use a combination of `DROP TABLE`, `DROP VIEW`, `ALTER TABLE`, and `ALTER VIEW` statements, to drop all the tables and views in the database or move them to other databases.

Examples:

See CREATE DATABASE Statement on page 87 for examples covering `CREATE DATABASE`, `USE`, and `DROP DATABASE`.

Cancellation: Cannot be cancelled.

DROP FUNCTION Statement

Removes a user-defined function (UDF), so that it is not available for execution during Impala `SELECT` or `INSERT` operations.

Syntax:

```
DROP [AGGREGATE] FUNCTION [IF EXISTS] [db_name.]function_name(type[, type...])
```

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.

Cancellation: Cannot be cancelled.

Related information:

User-Defined Functions (UDFs) on page 163, CREATE FUNCTION Statement on page 88

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
```

Statement type: DDL

Related information:

ALTER TABLE Statement on page 79, CREATE TABLE Statement on page 90, Partitioning on page 233 Internal Tables on page 73, External Tables on page 74

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 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*.

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).

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;
```

Related information:

For other tips about managing and reclaiming Impala disk space, see Managing Disk Space for Impala Data on page 47.

Cancellation: Cannot be cancelled.

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] [database_name.]view_name
```

Statement type: DDL

Cancellation: Cannot be cancelled.

Related information:

Views on page 74, CREATE VIEW Statement on page 95, ALTER VIEW Statement on page 83

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 118 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 105 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 90 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 84 for how to gather statistics, and How Impala Uses Statistics for Query Optimization on page 212 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 on page 194 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 224.

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 that you can use to fine-tune the resource management options explained in impalad Startup Options for Resource Management on page 43. (The estimated memory requirements are intentionally on the high side, to allow a margin for error, to avoid cancelling a query unnecessarily if you set the MEM_LIMIT option to the estimated memory figure.)

See EXPLAIN_LEVEL on page 194 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                            |
+----------------------------------------------------------+
```

These examples show how the extended EXPLAIN output becomes more accurate and informative as statistics are gathered by the COMPUTE STATS statement. Initially, much of the information about data size and distribution is marked "unavailable". Impala can determine the raw data size, but not the number of rows or number of distinct values for each column without additional analysis. The COMPUTE STATS statement performs this analysis, so a subsequent EXPLAIN statement has additional information to use in deciding how to optimize the distributed query.

```
[localhost:21000] > set explain_level=extended;
EXPLAIN_LEVEL set to extended
[localhost:21000] > explain select x from t1;
[localhost:21000] > explain select x from t1;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=32.00MB VCores=1 |
|                                                          |
| 01:EXCHANGE [PARTITION=UNPARTITIONED]                    |
| |  hosts=1 per-host-mem=unavailable                      |
| |  tuple-ids=0 row-size=4B cardinality=unavailable       |
| |                                                        |
| 00:SCAN HDFS [default.t2, PARTITION=RANDOM]              |
|    partitions=1/1 size=36B                               |
|    table stats: unavailable                              |
|    column stats: unavailable                             |
|    hosts=1 per-host-mem=32.00MB                          |
|    tuple-ids=0 row-size=4B cardinality=unavailable       |
+----------------------------------------------------------+
```

```
[localhost:21000] > compute stats t1;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
+----------------------------------------+
[localhost:21000] > explain select x from t1;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=64.00MB VCores=1 |
|                                                          |
| 01:EXCHANGE [PARTITION=UNPARTITIONED]                    |
| |  hosts=1 per-host-mem=unavailable                      |
| |  tuple-ids=0 row-size=4B cardinality=0                 |
| |                                                        |
| 00:SCAN HDFS [default.t1, PARTITION=RANDOM]              |
|    partitions=1/1 size=36B                               |
|    table stats: 0 rows total                             |
|    column stats: all                                     |
|    hosts=1 per-host-mem=64.00MB                          |
|    tuple-ids=0 row-size=4B cardinality=0                 |
+----------------------------------------------------------+
```

Cancellation: Cannot be cancelled.

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 { 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 ::= [SHUFFLE] | [NOSHUFFLE]    (Note: the square brackets are part of the
 syntax.)
```

Usage notes:

Impala currently supports:

- `INSERT INTO` to append data to a table.
- `INSERT OVERWRITE` to replace the data in a table.
- 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.)
- Specify the names or order of columns to be inserted, different than the columns of the table being queried by the `INSERT` statement. (This feature was added in Impala 1.1.)
- An optional hint clause immediately before the `SELECT` keyword, to fine-tune the behavior when doing an `INSERT ... SELECT` operation into partitioned Parquet tables. 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 1 GB memory buffers being allocated to buffer the data for each partition. For usage details, see Loading Data into Parquet Tables on page 247.

Note:

- Insert commands that partition or add files result in changes to Hive metadata. Because Impala uses Hive metadata, such changes may necessitate a Hive 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.

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.

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 on page 202 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.
```

Inserting data into partitioned tables requires slightly different syntax that divides the partitioning columns from the others:

```
create table t1 (i int) partitioned by (x int, y string);
-- Select an INT column from another table.
-- All inserted rows will have the same x and y values, as specified in the INSERT
statement.
-- This technique of specifying all the partition key values is known as static
partitioning.
insert into t1 partition(x=10, y='a') select c1 from some_other_table;
-- Select two INT columns from another table.
-- All inserted rows will have the same y value, as specified in the INSERT statement.
-- Values from c2 go into t1.x.
-- Any partitioning columns whose value is not specified are filled in
-- from the columns specified last in the SELECT list.
-- This technique of omitting some partition key values is known as dynamic partitioning.
insert into t1 partition(x, y='b') select c1, c2 from some_other_table;
-- Select an INT and a STRING column from another table.
-- All inserted rows will have the same x value, as specified in the INSERT statement.
-- Values from c3 go into t1.y.
insert into t1 partition(x=20, y) select c1, c3  from some_other_table;
```

The following example shows 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;

-- For partitioned tables, all the partitioning columns must be mentioned in the ()
column list
-- or a PARTITION clause; these columns cannot be defaulted to NULL.
create table pt1 (x int, y int) partitioned by (z int);
-- The values from c1 are copied into the column x in the new table,
-- all in the same partition based on a constant value for z.
-- The values of y in the new table are all NULL.
insert into pt1 (x) partition (z=5) select c1 from t1;
-- Again we omit the values for column y so they are all NULL.
-- The inserted x values can go into different partitions, based on
-- the different values inserted into the partitioning column z.
insert into pt1 (x,z) select x, z from t2;
```

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 on different data nodes, 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 63 for the notation to use for literal values, especially String Literals on page 63 for quoting and escaping conventions for strings. See SQL Operators on page 65 and Built-in Functions on page 138 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`.

To use a `VALUES` clause like a table in other statements, wrap it in parentheses and use `AS` clauses to specify aliases for the entire object and any columns you need to refer to:

```
[localhost:21000] > select * from (values(4,5,6),(7,8,9)) as t;
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 4 | 5 | 6 |
| 7 | 8 | 9 |
+---+---+---+
[localhost:21000] > 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 |
+-----+-------+-----+
```

For example, you might use a tiny table constructed like this from constant literals or function return values as part of a longer statement involving joins or `UNION ALL`.

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 265 for more details about using Impala with HBase.

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).

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.

INVALIDATE METADATA Statement

Marks the metadata for one or all tables as stale. Required after a table is created through the Hive shell, before the table is available for Impala queries. The next time the current Impala node performs a query against a table whose metadata is invalidated, Impala reloads the associated metadata before the query proceeds. This is a relatively expensive operation compared to the incremental metadata update done by the `REFRESH` statement, so in the common scenario of adding new data files to an existing table, prefer `REFRESH` rather than `INVALIDATE METADATA`. If you are not familiar with the way Impala uses metadata and how it shares the same metastore database as Hive, see Overview of Impala Metadata and the Metastore on page 16 for background information.

To accurately respond to queries, Impala must have current metadata about those databases and tables that clients query directly. Therefore, if some other entity modifies information used by Impala in the metastore that Impala and Hive share, the information cached by Impala must be updated. However, this does not mean that all metadata updates require an Impala update.

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. See New Features in Impala Version 1.2.4 for details.

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. See New Features in Impala Version 1.2.4 for details.

In Impala 1.2 and higher, a dedicated daemon (catalogd) broadcasts DDL changes made through Impala to all Impala nodes. Formerly, after you created a database or table while connected to one Impala node, you needed to issue an INVALIDATE METADATA statement on another Impala node before accessing the new database or table from the other node. Now, newly created or altered objects are picked up automatically by all Impala nodes. You must still use the INVALIDATE METADATA technique after creating or altering objects through Hive. See The Impala Catalog Service on page 14 for more information on the catalog service.

The INVALIDATE METADATA statement is new in Impala 1.1 and higher, and takes over some of the use cases of the Impala 1.0 REFRESH statement. Because REFRESH now requires a table name parameter, to flush the metadata for all tables at once, use the INVALIDATE METADATA statement.

Because REFRESH *table_name* only works for tables that the current Impala node is already aware of, when you create a new table in the Hive shell, you must enter INVALIDATE METADATA with no table parameter before you can see the new table in impala-shell. Once the table is known the the Impala node, you can issue REFRESH *table_name* after you add data files for that table.

INVALIDATE METADATA and REFRESH are counterparts: INVALIDATE METADATA waits to reload the metadata when needed for a subsequent query, but reloads all the metadata for the table, which can be an expensive operation, especially for large tables with many partitions. REFRESH reloads the metadata immediately, but only loads the block location data for newly added data files, making it a less expensive operation overall. If data was altered in some more extensive way, such as being reorganized by the HDFS balancer, use INVALIDATE METADATA to avoid a performance penalty from reduced local reads. If you used Impala version 1.0, the INVALIDATE METADATA statement works just like the Impala 1.0 REFRESH statement did, while the Impala 1.1 REFRESH is optimized for the common use case of adding new data files to an existing table, thus the table name argument is now required.

The syntax for the INVALIDATE METADATA command is:

```
INVALIDATE METADATA [table_name]
```

By default, the cached metadata for all tables is flushed. If you specify a table name, only the metadata for that one table is flushed. Even for a single table, INVALIDATE METADATA is more expensive than REFRESH, so prefer REFRESH in the common case where you add new data files for an existing table.

A metadata update for an impalad instance **is** required if:

- A metadata change occurs.
- **and** the change is made from another impalad instance in your cluster, or through Hive.
- **and** the change is made to a database to which clients such as the Impala shell or ODBC directly connect.

A metadata update for an Impala node is **not** required when you issue queries from the same Impala node where you ran ALTER TABLE, INSERT, or other table-modifying statement.

Database and table metadata is typically modified by:

- Hive - via ALTER, CREATE, DROP or INSERT operations.
- Impalad - via CREATE TABLE, ALTER TABLE, and INSERT operations.

INVALIDATE METADATA causes the metadata for that table to be marked as stale, and reloaded the next time the table is referenced. For a huge table, that process could take a noticeable amount of time; thus you might prefer to use REFRESH where practical, to avoid an unpredictable delay later, for example if the next reference to the table is during a benchmark test.

The following example shows how you might use the INVALIDATE METADATA statement after creating new tables (such as SequenceFile or HBase tables) through the Hive shell. Before the INVALIDATE METADATA statement was issued, Impala would give a "table not found" error if you tried to refer to those table names. The DESCRIBE statements cause the latest metadata to be immediately loaded for the tables, avoiding a delay the next time those tables are queried.

```
[impalad-host:21000] > invalidate metadata;
[impalad-host:21000] > describe t1;
...
[impalad-host:21000] > describe t2;
...
```

For more 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 30.

If you need to ensure that the metadata is up-to-date when you start an impala-shell session, run impala-shell with the -r or --refresh_after_connect command-line option. Because this operation adds a delay to the next query against each table, potentially expensive for large tables with many partitions, try to avoid using this option for day-to-day operations in a production environment.

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 if you have set the catalogd configuration option --load_catalog_in_background=false.) Impala reports any lack of write permissions as an INFO message in the log file, in case that represents an oversight. 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.

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. The ability to specify INVALIDATE METADATA *table_name* for a table created in Hive is a new capability in Impala 1.2.4. In earlier releases, that statement would have returned an error indicating an unknown table, requiring you to do INVALIDATE METADATA with no table name, a more expensive operation that reloaded metadata for all tables and databases.

```
$ hive
hive> create database new_db_from_hive;
OK
Time taken: 4.118 seconds
hive> create table new_db_from_hive.new_table_from_hive (x int);
OK
Time taken: 0.618 seconds
hive> quit;
$ impala-shell
[localhost:21000] > show databases like 'new*';
[localhost:21000] > refresh new_db_from_hive.new_table_from_hive;
ERROR: AnalysisException: Database does not exist: new_db_from_hive
[localhost:21000] > invalidate metadata new_db_from_hive.new_table_from_hive;
[localhost:21000] > show databases like 'new*';
+--------------------+
| name               |
+--------------------+
| new_db_from_hive   |
+---------------------+
[localhost:21000] > show tables in new_db_from_hive;
+---------------------+
| name                |
+---------------------+
| new_table_from_hive |
+---------------------+
```

Cancellation: Cannot be cancelled.

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 ...)]
```

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 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.

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 on page 202 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 `cloudera` 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/cloudera/thousand_strings.txt
$ random_strings.py 100 | hdfs dfs -put - /user/cloudera/hundred_strings.txt
$ random_strings.py 10 | hdfs dfs -put - /user/cloudera/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/cloudera/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/cloudera/thousand_strings.txt' into table
 t1;
ERROR: AnalysisException: INPATH location '/user/cloudera/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 cloudera cloudera      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/cloudera/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/cloudera/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
```

Cancellation: Cannot be cancelled.

REFRESH Statement

To accurately respond to queries, the Impala node that acts as the coordinator (the node to which you are connected through `impala-shell`, JDBC, or ODBC) must have current metadata about those databases and tables that are referenced in Impala queries. If you are not familiar with the way Impala uses metadata and how it shares the same metastore database as Hive, see Overview of Impala Metadata and the Metastore on page 16 for background information.

Use the `REFRESH` statement to load the latest metastore metadata and block location data for a particular table in these scenarios:

- After loading new data files into the HDFS data directory for the table. (Once you have set up an ETL pipeline to bring data into Impala on a regular basis, this is typically the most frequent reason why metadata needs to be refreshed.)
- After issuing `ALTER TABLE`, `INSERT`, `LOAD DATA`, or other table-modifying SQL statement in Hive.

You only need to issue the `REFRESH` statement on the node to which you connect to issue queries. The coordinator node divides the work among all the Impala nodes in a cluster, and sends read requests for the correct HDFS blocks without relying on the metadata on the other nodes.

`REFRESH` reloads the metadata for the table from the metastore database, and does an incremental reload of the low-level block location data to account for any new data files added to the HDFS data directory for the table. It is a low-overhead, single-table operation, specifically tuned for the common scenario where new data files are added to HDFS.

The syntax for the `REFRESH` command is:

```
REFRESH table_name
```

Only the metadata for the specified table is flushed. The table must already exist and be known to Impala, either because the `CREATE TABLE` statement was run in Impala rather than Hive, or because a previous `INVALIDATE METADATA` statement caused Impala to reload its entire metadata catalog.

Note:

In Impala 1.2 and higher, the catalog service broadcasts any changed metadata as a result of Impala `ALTER TABLE`, `INSERT` and `LOAD DATA` statements to all Impala nodes. Thus, the `REFRESH` statement is only required if you load data through Hive or by manipulating data files in HDFS directly. See The Impala Catalog Service on page 14 for more information on the catalog service.

In Impala 1.2.1 and higher, another way to avoid inconsistency across nodes is to enable the `SYNC_DDL` query option before performing a DDL statement or an `INSERT` or `LOAD DATA`.

The functionality of the `REFRESH` statement has changed in Impala 1.1 and higher. Now the table name is a required parameter. To flush the metadata for all tables, use the `INVALIDATE METADATA` command.

Because `REFRESH` *`table_name`* only works for tables that Impala is already aware of, when you create a new table in the Hive shell, you must enter `INVALIDATE METADATA` with no table parameter before you can see the new table in `impala-shell`. Once the table is known to Impala, you can issue `REFRESH` *`table_name`* as needed after you add more data files for that table.

`INVALIDATE METADATA` and `REFRESH` are counterparts: `INVALIDATE METADATA` waits to reload the metadata when needed for a subsequent query, but reloads all the metadata for the table, which can be an expensive operation, especially for large tables with many partitions. `REFRESH` reloads the metadata immediately, but only loads the block location data for newly added data files, making it a less expensive operation overall. If data was altered in some more extensive way, such as being reorganized by the HDFS balancer, use `INVALIDATE METADATA` to avoid a performance penalty from reduced local reads. If you used Impala version 1.0, the `INVALIDATE METADATA` statement works just like the Impala 1.0 `REFRESH` statement did, while the Impala 1.1 `REFRESH` is optimized for the common use case of adding new data files to an existing table, thus the table name argument is now required.

A metadata update for an `impalad` instance **is** required if:

- A metadata change occurs.
- **and** the change is made through Hive.
- **and** the change is made to a database to which clients such as the Impala shell or ODBC directly connect.

A metadata update for an Impala node is **not** required after you run `ALTER TABLE`, `INSERT`, or other table-modifying statement in Impala rather than Hive. Impala handles the metadata synchronization automatically through the catalog service.

Database and table metadata is typically modified by:

- Hive - through `ALTER`, `CREATE`, `DROP` or `INSERT` operations.
- Impalad - through `CREATE TABLE`, `ALTER TABLE`, and `INSERT` operations. In Impala 1.2 and higher, such changes are propagated to all Impala nodes by the Impala catalog service.

`REFRESH` causes the metadata for that table to be immediately reloaded. For a huge table, that process could take a noticeable amount of time; but doing the refresh up front avoids an unpredictable delay later, for example if the next reference to the table is during a benchmark test.

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 on page 202 for details.

Examples:

The following example shows how you might use the `REFRESH` statement after manually adding new HDFS data files to the Impala data directory for a table:

```
[impalad-host:21000] > refresh t1;
[impalad-host:21000] > refresh t2;
[impalad-host:21000] > select * from t1;
...
```

```
[impalad-host:21000] > select * from t2;
...
```

For more 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 30.

Related impalad options:

In Impala 1.0, the -r option of impala-shell issued REFRESH to reload metadata for all tables.

In Impala 1.1 and higher, this option issues INVALIDATE METADATA because REFRESH now requires a table name parameter. Due to the expense of reloading the metadata for all tables, the impala-shell -r option is not recommended for day-to-day use in a production environment.

In Impala 1.2 and higher, the -r option is needed even less frequently, because metadata changes caused by SQL statements in Impala are automatically broadcast to all Impala nodes.

HDFS considerations:

The REFRESH 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. Impala reports any lack of write permissions as an INFO message in the log file, in case that represents an oversight. 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.

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.

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.

Impala SELECT queries support:

- SQL data types: BOOLEAN, TINYINT, SMALLINT, INT, BIGINT, FLOAT, DOUBLE, TIMESTAMP, STRING.
- 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.
- DISTINCT clause per query. See DISTINCT Operator on page 134 for details.
- Subqueries in a FROM clause.
- 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 keyword

  See Joins on page 119 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 Built-in Functions on page 138.

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).

Joins

A join query is one that combines data from two or more tables, and returns a result set containing items from some or all of those 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 keyword

```
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 SEMI 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;
```

Cartesian joins:

To avoid producing huge result sets by mistake, Impala does not allow Cartesian joins of the form:

```
SELECT ... FROM t1 JOIN t2;
SELECT ... FROM t1, t2;
```

If you intend to join the tables based on common values, add ON or WHERE clauses to compare columns across the tables. If you truly intend to do a Cartesian join, use the CROSS JOIN keyword as the join operator. The CROSS JOIN form does not use any ON clause, because it produces a result set with all combinations of rows from the left-hand and right-hand tables. The result set can still be filtered by subsequent WHERE clauses. For example:

```
SELECT ... FROM t1 CROSS JOIN t2;
SELECT ... FROM t1 CROSS JOIN t2 WHERE tests_on_non_join_columns;
```

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;
```

Semi-joins:

Semi-joins are a relatively rarely used variation. With the left semi-join (the only kind of semi-join available with Impala), 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.

```
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 (not supported):

Impala does not support `WHERE` clauses such as `IN (`*`subquery`*`)`, `NOT IN (`*`subquery`*`)`, `EXISTS (`*`subquery`*`)`, and `NOT EXISTS (`*`subquery`*`)`. Therefore from a practical standpoint, you cannot express an anti-join condition, where values from one table are returned only if no matching values are present in another table.

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.

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. Use `SHOW TABLE STATS` *`table_name`* and `SHOW COLUMN STATS` *`table_name`* to check.
> - 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. In Impala 1.2.2 and higher, use the Impala `COMPUTE STATS` statement to collect statistics at both the table and column levels, and keep the statistics up to date after any substantial `INSERT` or `LOAD DATA` operation.
> - 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 206.

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 Hints on page 133.

See these tutorials for examples of different kinds of joins:

- Cross Joins and Cartesian Products with the CROSS JOIN Operator on page 31

ORDER BY Clause

The familiar ORDER BY clause of a SELECT statement sorts the result set based on the values from one or more columns.

For distributed queries, this is a relatively expensive operation, because the entire result set must be produced and transferred to one node before the sorting can happen. This 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.

Note:

In Impala 1.4.0 and higher, the LIMIT clause is now optional (rather than required) for queries that use the ORDER BY clause. Impala automatically uses a temporary disk work area to perform the sort if the sort operation would otherwise exceed the Impala memory limit for a particular data node.

Syntax:

The full syntax for the ORDER BY clause is:

```
ORDER BY col1 [, col2 ...] [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

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.

See NULL on page 64 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 node, Impala automatically uses a temporary disk work area to perform the sort operation.

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 data node 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 data nodes, each node 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 data node 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 data node, 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 data node (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 operations are stored in the directory `/tmp/impala-scratch`. These files are removed when the sort operation finishes. (Multiple concurrent queries can perform `ORDER BY` queries that use the external sort 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. The scratch directory 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. Impala will not start if it cannot create or read and write files in the "scratch" directory. 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.

Restrictions:

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 on different data nodes, 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 data node, 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.
- The query options DEFAULT_ORDER_BY_LIMIT and ABORT_ON_DEFAULT_LIMIT_EXCEEDED, which formerly controlled the behavior of `ORDER BY` queries with no limit specified, are now ignored.

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.

Related information:

See SELECT Statement on page 118 for further examples of queries with the ORDER BY clause.

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.

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 times_purchased >= 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_revenue_per_sale 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.

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.

HAVING Clause

Performs a filter operation on a `SELECT` query, by examining the results of aggregation functions rather than testing each individual table row. Thus always used in conjunction with a function such as `COUNT()`, `SUM()`, `AVG()`, `MIN()`, or `MAX()`, and typically with the `GROUP BY` clause also.

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 node, Impala automatically uses a temporary disk work area to perform the sort operation.

See ORDER BY Clause on page 124 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.

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  |
+----+
Returned 5 row(s) in 0.26s
[localhost:21000] > select x from numbers order by x limit 5 offset 5;
+----+
| x  |
+----+
| 6  |
| 7  |
| 8  |
| 9  |
| 10 |
+----+
Returned 5 row(s) in 0.23s
```

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, we 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
```

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;
```

Hints

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. The hints are represented as keywords surrounded by `[]` square brackets; include the brackets in the text of the SQL statement.

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.)

To see which join strategy is used for a particular query, examine the `EXPLAIN` output for that query.

Note:

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 206 for details.

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 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 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 joins are processed 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 t1.name, t2.id, t3.price
  from t1 join [shuffle] t2 join [broadcast] t3
  on t1.id = t2.id and t2.id = t3.id;
```

For more background information and performance considerations for join queries, see Joins on page 119.

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:

- These hints are available in Impala 1.2.2 and higher.
- You would only use these hints if an INSERT into a partitioned Parquet table was failing due to capacity limits, or if such an INSERT was succeeding but with less-than-optimal performance.
- To use these hints, put the hint keyword [SHUFFLE] or [NOSHUFFLE] (including the square brackets) after the PARTITION clause, immediately before the SELECT keyword.
- [SHUFFLE] selects an execution plan that minimizes the number of files being written simultaneously to HDFS, and the number of 1 GB memory buffers holding data for individual partitions. Thus it reduces overall resource usage for the INSERT operation, allowing some INSERT 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 constructed on the same node.
- [NOSHUFFLE] selects an execution plan that might be faster overall, but might also produce a larger number of small data files or exceed capacity limits, causing the INSERT operation to fail. Use [SHUFFLE] in cases where an INSERT statement fails or runs inefficiently due to all nodes attempting to construct data for all partitions.
- Impala automatically uses the [SHUFFLE] method if any partition key column in the source table, mentioned in the INSERT ... SELECT query, does not have column statistics. In this case, only the [NOSHUFFLE] hint would have any effect.
- If column statistics are available for all partition key columns in the source table mentioned in the INSERT ... SELECT 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 INSERT operation. In this case, you might need the [SHUFFLE] or the [NOSHUFFLE] hint to override the execution plan selected by Impala.

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;
```

One construct that Impala SQL does *not* support is using `DISTINCT` in more than one aggregation function in the same query. For example, you could not have a single query with both `COUNT(DISTINCT c_first_name)` and `COUNT(DISTINCT c_last_name)` in the `SELECT` list.

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:

Impala only allows a single `COUNT(DISTINCT *columns*)` expression in each query.

If you do not need precise accuracy, you can produce an estimate of the distinct values for a column by specifying `NDV(*column*)`; a query can contain multiple instances of `NDV(*column*)`.

To produce the same result as multiple `COUNT(DISTINCT)` expressions, you can use the following technique for queries involving a single table:

```
select v1.c1 result1, v2.c1 result2 from
  (select count(distinct col1) as c1 from t1) v1
    cross join
  (select count(distinct col2) as c1 from t1) v2;
```

Because `CROSS JOIN` is an expensive operation, prefer to use the `NDV()` technique wherever practical.

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.

SHOW Statement

The `SHOW` statement is a flexible way to get information about different types of Impala objects. You can 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.

Syntax:

To display a list of available objects of a particular kind, issue these statements:

```
SHOW DATABASES [[LIKE] 'pattern']
SHOW SCHEMAS [[LIKE] 'pattern'] - an alias for SHOW DATABASES
SHOW TABLES [IN database_name] [[LIKE] 'pattern']
SHOW [AGGREGATE] FUNCTIONS [IN database_name] [[LIKE] 'pattern']
SHOW CREATE TABLE [database_name].table_name
SHOW TABLE STATS [database_name.]table_name
SHOW COLUMN STATS [database_name.]table_name
SHOW PARTITIONS [database_name.]table_name
```

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*';
```

Usage notes:

When authorization is enabled, the output of the `SHOW` statement is limited to those objects for which you have some privilege. There might be other database, tables, and so on, but their names are concealed. 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 for how to set up authorization and add privileges for specific kinds of objects.

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.

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`.

SHOW CREATE TABLE:

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.

SHOW TABLE STATS, SHOW COLUMN STATS:

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. See How Impala Uses Statistics for Query Optimization on page 212 for usage information and examples.

SHOW PARTITIONS:

`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 84 for details. For example, on a `CENSUS` 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 |         |
+-------+-------+--------+------+---------+
```

SHOW FUNCTIONS:

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.

To display Impala built-in functions, specify the special database name `_impala_builtins`:

```
show functions in _impala_builtins;
+----------------+----------------------------------------+
| return type    | signature                              |
+----------------+----------------------------------------+
| BOOLEAN        | ifnull(BOOLEAN, BOOLEAN)               |
| TINYINT        | ifnull(TINYINT, TINYINT)               |
| SMALLINT       | ifnull(SMALLINT, SMALLINT)             |
| INT            | ifnull(INT, INT)                       |
...

show functions in _impala_builtins like '*week*';
+-------------+------------------------------+
| return type | signature                    |
+-------------+------------------------------+
| INT         | weekofyear(TIMESTAMP)        |
| TIMESTAMP   | weeks_add(TIMESTAMP, INT)    |
| TIMESTAMP   | weeks_add(TIMESTAMP, BIGINT) |
| TIMESTAMP   | weeks_sub(TIMESTAMP, INT)    |
| TIMESTAMP   | weeks_sub(TIMESTAMP, BIGINT) |
| INT         | dayofweek(TIMESTAMP)         |
+-------------+------------------------------+
```

To search for functions that use a particular data type, specify a case-sensitive data type name in all capitals:

```
show functions in _impala_builtins like '*BIGINT*';
+----------------------------------------+
| name                                   |
+----------------------------------------+
| adddate(TIMESTAMP, BIGINT)             |
| bin(BIGINT)                            |
| coalesce(BIGINT...)                    |
...
```

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               |
+--------------------+
| _impala_builtins   |
| analyze_testing    |
| avro               |
| ctas               |
| d1                 |
| d2                 |
| d3                 |
| default            |
| file_formats       |
| hbase              |
| load_data          |
| partitioning       |
| regexp_testing     |
| reports            |
| temporary          |
+--------------------+
Returned 14 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
```

Cancellation: Cannot be cancelled.

USE Statement

By default, when you connect to an Impala instance, you begin in a database named `default`. Issue the statement `USE` *`db_name`* to switch to another database within an `impala-shell` session. The current database is where any `CREATE TABLE`, `INSERT`, `SELECT`, or other statements act when you specify a table without prefixing it with a database name.

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 87 for examples covering `CREATE DATABASE`, `USE`, and `DROP DATABASE`.

Cancellation: Cannot be cancelled.

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 `SELECT` statements. The built-in functions let a SQL query return results with all formatting, calculating, and type conversions applied, rather than performing time-consuming postprocessing in another application. By applying function calls where practical, you can make a SQL query that is as convenient as an expression in a procedural programming language or a formula in a spreadsheet.

The categories of functions supported by Impala are:

- Mathematical Functions on page 139
- Type Conversion Functions on page 145
- Date and Time Functions on page 146
- Conditional Functions on page 151
- String Functions on page 153
- Aggregation functions, explained in Aggregate Functions on page 158.

You call any of these functions through the `SELECT` statement. For most functions, you can omit the `FROM` clause and supply literal values for any required arguments:

```
select abs(-1);
select concat('The rain ', 'in Spain');
select power(2,5);
```

When you use a `FROM` clause and specify a column name as a function argument, the function is applied for each item in the result set:

```
select concat('Country = ',country_code) from all_countries where population > 100000000;
select round(price) as dollar_value from product_catalog where price between 0.0 and
100.0;
```

Typically, if any argument to a built-in function is `NULL`, the result value is also `NULL`:

```
select cos(null);
select power(2,null);
select concat('a',null,'b');
```

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.

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 do 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.

Mathematical Functions

Impala supports the following mathematical functions:

`abs(double a), abs(decimal(p,s) a)`

Purpose: Returns the absolute value of the argument.

Return type: `double` or `decimal(p,s)` based on the type of the input argument

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`

`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), ceiling(double a), ceil(decimal(p,s) a), ceiling(decimal(p,s) a)`

Purpose: Returns the smallest integer that is greater than or equal to the argument.

Return type: `int` or `decimal(p,s)` based on the type of the input argument

`conv(bigint num, int from_base, int to_base), conv(string num, int from_base, int to_base)`

Purpose: Returns a string representation of an integer value in a particular base. The input value can be a string, for example to convert a hexadecimal number such as `fce2` to decimal. To use the return value as a number (for example, when converting to base 10), use `CAST()` to convert to the appropriate type.

Return type: `string`

`cos(double a)`

Purpose: Returns the cosine of the argument.

Return type: `double`

`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)`

Purpose: Returns the mathematical constant e raised to the power of the argument.

Return type: `double`

`floor(double a)`

Purpose: Returns the largest integer that is less than or equal to the argument.

Return type: `int`

`fmod(double a, double b), fmod(float a, float b)`

Purpose: Returns the modulus of a number.

Return type: `float` or `double`, depending on type of arguments

Added in: Impala 1.1.1

`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, `pow(-1, 0.5)` returns infinity and `1/0` 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, `pow(-1, 0.5)` returns infinity and `1/0` 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)`

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)`

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.

`negative(int a), negative(double a), negative(decimal(p,s) a)`

Purpose: Returns the argument with the sign reversed; returns a positive value if the argument was already negative.

Return type: `int`, `double`, or `decimal(p,s)` depending on type of argument

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(int a, int b), pmod(double a, double b)`

Purpose: Returns the positive modulus of a number.

Return type: `int` or `double`, depending on type of arguments

`positive(int a), positive(double a), positive(decimal(p,s) a`

Purpose: Returns the original argument unchanged (even if the argument is negative).

Return type: `int`, `double`, or `decimal(p,s)` depending on type of argument

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)`

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(int numerator, int denominator)`

Purpose: Returns the first argument divided by the second argument, discarding any fractional part. Avoids promoting arguments to `DOUBLE` as happens with the / SQL operator.

Return type: `int`

`radians(double a)`

Purpose: Converts argument value from degrees to radians.

Return type: `double`

`rand(), rand(int 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 ...`

`round(double a), round(double a, int d), round(decimal 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: `bigint` for single `float`argument. `double` for `double` argument when second argument greater than zero. For `DECIMAL` values, the smallest `DECIMAL(*p*,*s*)` type with appropriate precision and scale.

`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`

`sqrt(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`

`unhex(string a)`

Purpose: Returns a string of characters with ASCII values corresponding to pairs of hexadecimal digits in the argument.

Return type: `string`

Type Conversion Functions

Impala supports the following type conversion functions:

- `cast(`*`expr`*` as `*`type`*`)`

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. Use `CAST` when passing a column value or literal to a function that expects a parameter with a different type. For example:

```
select concat('Here are the first ',10,' results.'); -- Fails
select concat('Here are the first ',cast(10 as string),' results.'); -- Succeeds
```

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 61 for details. `INTERVAL` expressions are also allowed as the second argument for the `date_add()` and `date_sub()` functions, rather than integers.

Impala supports the following data and time functions:

`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.

`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`

`current_timestamp()`

Purpose: Alias for the `now()` function.

Return type: `timestamp`

`date_add(timestamp startdate, int days)`, `date_add(timestamp startdate, *interval_expression*)`

Purpose: Adds a specified number of days to a `TIMESTAMP` value. The first argument can be a string, which is automatically cast to `TIMESTAMP` if it uses the recognized format, as described in TIMESTAMP Data Type on page 61. 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 61 for details.

Return type: `timestamp`

`date_sub(timestamp startdate, int days)`, `date_sub(timestamp startdate, *interval_expression*)`

Purpose: Subtracts a specified number of days from a `TIMESTAMP` value. The first argument can be a string, which is automatically cast to `TIMESTAMP` if it uses the recognized format, as described in TIMESTAMP Data Type on page 61. 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 61 for details.

Return type: `timestamp`

`datediff(string enddate, string startdate)`

Purpose: Returns the number of days between two dates represented as strings.

Return type: `int`

`day(string date), dayofmonth(string date)`

Purpose: Returns the day field from a date represented as a string.

Return type: `int`

`dayname(string date)`

Purpose: Returns the day field from a date represented as a string, 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`

`dayofweek(string date)`

Purpose: Returns the day field from a date represented as a string, corresponding to the day of the week. The range of return values is 1 (Sunday) to 7 (Saturday).

Return type: `int`

`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`

`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`

`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`

`extract(timestamp, string unit)`

Purpose: Returns one of the numeric date or time fields from a `TIMESTAMP` value.

Unit argument: The `unit` string can be one of `year`, `month`, `day`, `hour`, `minute`, `second`, or `millisecond`. This argument value is case-insensitive.

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` 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: `int`

`from_unixtime(bigint unixtime[, string format])`

Purpose: Converts the number of seconds from the Unix epoch to the specified time into a string.

Return type: `string`

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 61 for details.

Currently, the format string is case-sensitive, especially to distinguish `m` for minutes and `M` for months. In Impala 1.3 and higher, 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.

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 casted must still have the elements in the specified order and use the specified delimiter characters, as described in TIMESTAMP Data Type on page 61.

Examples:

```
[localhost:21000] > 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                              |
+-------------------------------------------------------+
[localhost:21000] > select from_unixtime(1392394861,"yyyy-MM-dd");
+-----------------------------------------+
| from_unixtime(1392394861, 'yyyy-mm-dd') |
+-----------------------------------------+
| 2014-02-14                              |
+-----------------------------------------+
[localhost:21000] > select from_unixtime(1392394861,"HH:mm:ss.SSSS");
+--------------------------------------------+
| from_unixtime(1392394861, 'hh:mm:ss.ssss') |
+--------------------------------------------+
| 16:21:01.0000                              |
+--------------------------------------------+
[localhost:21000] > select from_unixtime(1392394861,"HH:mm:ss");
+---------------------------------------+
| from_unixtime(1392394861, 'hh:mm:ss') |
+---------------------------------------+
| 16:21:01                              |
+---------------------------------------+
```

`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');
```

`from_utc_timestamp(timestamp, string timezone)`

Purpose: Converts a specified UTC timestamp value into the appropriate value for a specified time zone.

Return type: `timestamp`

`hour(string date)`

Purpose: Returns the hour field from a date represented as a string.

Return type: `int`

`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`

`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`

`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`

`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`

`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`

`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`

`minute(string date)`

Purpose: Returns the minute field from a date represented as a string.

Return type: `int`

`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`

`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`

`month(string date)`

Purpose: Returns the month field from a date represented as a string.

Return type: `int`

`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`

`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`

`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`

`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`

`now()`

Purpose: Returns the current date and time (in the UTC 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 61 for examples.

`second(string date)`

Purpose: Returns the second field from a date represented as a string.

Return type: `int`

`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`

`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`

`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`

`to_date(timestamp)`

Purpose: Returns a string representation of the date field from a timestamp value.

Return type: `string`

`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`

`trunc(timestamp, string unit)`

Purpose: Strips off fields and optionally rounds a `TIMESTAMP` value.

Unit argument: The `unit` argument value is case-sensitive. This argument string can be one of:

- `SYYYY`, `YYYY`, `YEAR`, `SYEAR`, `YYY`, `YY`, `Y`: Year (rounds up on July 1).
- `Q`: Quarter (rounds up on the sixteenth day of the second month of the quarter).
- `MONTH`, `MON`, `MM`, `RM`: Month (rounds up on the sixteenth day).
- `WW`, `W`: Same day of the week as the first day of the month.
- `DDD`, `DD`, `J`: Day of the month.
- `DAY`, `DY`, `D`: Starting day of the week.
- `HH`, `HH12`, `HH24`: Hour. A `TIMESTAMP` value rounded or truncated to the hour is always represented in 24-hour notation, even for the `HH12` argument string.
- `MI`: Minute.

Usage notes: 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.

Return type: `timestamp`

`unix_timestamp(), unix_timestamp(string datetime), unix_timestamp(string datetime, string format), unix_timestamp(timestamp datetime)`

Purpose: Returns an integer value representing the current date and time as a delta from the Unix epoch, or converts from a specified date and time value represented as a `TIMESTAMP` or `STRING`.

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.

`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');
```

`weekofyear(string date)`

Purpose: Returns the corresponding week (1-53) from a date represented as a string.

Return type: `int`

`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`

`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`

`year(string date)`

Purpose: Returns the year field from a date represented as a string.

Return type: `int`

`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`

`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`

Conditional Functions

Impala supports the following conditional functions for testing equality, comparison operators, and nullity:

`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

`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

`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

`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 ifNotNull)`

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

`isnull(type a, type ifNotNull)`

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

`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 49. 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 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

`nvl(type a, type ifNotNull)`

Purpose: Alias for the `isnull()` function. Tests if an expression is `NULL`, and returns the expression result value if not. If the first argument is `NULL`, returns the second argument. Equivalent to the `nvl()` function from Oracle Database or `ifnull()` from MySQL.

Return type: same as the first argument value

Added in: Impala 1.1

`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 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

String Functions

Impala supports the following string functions:

`ascii(string str)`

Purpose: Returns the numeric ASCII code of the first character of the argument.

Return type: `int`

`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`

`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.

`initcap(string str)`

Purpose: Returns the input string with the first letter capitalized.

Return type: `string`

`instr(string str, string substr)`

Purpose: Returns the position (starting from 1) of the first occurrence of a substring within a longer string.

Return type: `int`

`length(string a)`

Purpose: Returns the length in characters of the argument string.

Return type: `int`

`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`

`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)`

Purpose: Returns the argument string with any leading spaces removed from the left side.

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_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`

The Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Boost library. For details, see the Boost documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on. It does not support `.*?` for non-greedy matches.

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
```

The Boost regular expression syntax does not support the `.*?` idiom for non-greedy matches. This example shows how a pattern string starting with `.*` matches the longest possible portion of the source string, effectively serving as a greedy match and 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), effectively serving as a non-greedy match.

```
[localhost:21000] > select regexp_extract('AbcdBCdefGHI','.*([[:lower:]]+)',1);
+-------------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*([[:lower:]]+)', 1) |
+-------------------------------------------------------+
| def                                                   |
+-------------------------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select regexp_extract('AbcdBCdefGHI','.*([[:lower:]]+).*',1);
+---------------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*([[:lower:]]+).*', 1) |
+----------------------------------------------------------+
| bcd                                                      |
+----------------------------------------------------------+
Returned 1 row(s) in 0.11s
```

`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`

The Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Boost library. For details, see the Boost documentation. It has most idioms familiar from regular expressions in Perl, Python, and so on. It does not support `.*?` for non-greedy matches.

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`

`reverse(string a)`

Purpose: Returns the argument string with characters in reversed order.

Return type: `string`

`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)`

Purpose: Returns the argument string with any trailing spaces removed from the right side.

Return type: `string`

`space(int n)`

Purpose: Returns a concatenated string of the specified number of spaces. Shorthand for `repeat(' ',`*`n`*`)`.

Return type: `string`

`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 a set of characters replaced by another set of characters.

Return type: `string`

`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()`.

Return type: `string`

`upper(string a), ucase(string a)`

Purpose: Returns the argument string converted to all-uppercase.

Return type: `string`

Miscellaneous Functions

Impala supports the following utility functions that do not operate on a particular column or data type:

`current_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`

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`

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 user name, 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.

Return type: `string`

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)

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.

AVG Function

An aggregate function that returns the average value from 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 `AVG` are `NULL`, `AVG` returns `NULL`.

When the query contains a `GROUP BY` clause, returns one value for each combination of grouping values.

Return type: `DOUBLE`

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;
```

COUNT Function

An aggregate function that returns the number of rows, or the number of non-`NULL` 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`

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;
```

Note:

Impala only allows a single `COUNT(DISTINCT `*`columns`*`)` expression in each query.

If you do not need precise accuracy, you can produce an estimate of the distinct values for a column by specifying `NDV(`*`column`*`)`; a query can contain multiple instances of `NDV(`*`column`*`)`.

To produce the same result as multiple `COUNT(DISTINCT)` expressions, you can use the following technique for queries involving a single table:

```
select v1.c1 result1, v2.c1 result2 from
  (select count(distinct col1) as c1 from t1) v1
    cross join
  (select count(distinct col2) as c1 from t1) v2;
```

Because `CROSS JOIN` is an expensive operation, prefer to use the `NDV()` technique wherever practical.

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.

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`

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`.

When the query contains a `GROUP BY` clause, returns one value for each combination of grouping values.

Return type: Same as the input argument

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;
```

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`.

When the query contains a `GROUP BY` clause, returns one value for each combination of grouping values.

Return type: Same as the input argument

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;
```

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.

This is the mechanism used internally by the `COMPUTE STATS` statement for computing the number of distinct values in a column.

Usage notes:

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.

Currently, the return value is always a `STRING`. The return type is subject to change in future releases. Always use `CAST()` to convert the result to whichever data type is appropriate for your computations.

STDDEV, STDDEV_SAMP, STDDEV_POP Functions

An aggregate function that standard deviation of a set of numbers.

Related information:

This function works with any numeric data type.

Currently, the return value is always a `STRING`. The return type is subject to change in future releases. Always use `CAST()` to convert the result to whichever data type is appropriate for your computations.

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) |         |
+--------------------+--------------+---------+
```

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.

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

Examples:

```
-- 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;
```

VARIANCE, VARIANCE_SAMP, VARIANCE_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.

This function works with any numeric data type.

Currently, the return value is always a STRING. The return type is subject to change in future releases. Always use CAST() to convert the result to whichever data type is appropriate for your computations.

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.

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".

Related information:

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) |         |
+--------------------+--------------+---------+
```

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 data types supported by Impala. For example, nested or composite types are not supported.
- Currently, Hive UDFs that accept or return the `TIMESTAMP` type are not supported.
- 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 will return `NULL`.
- 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++.

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.

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.

Installing the UDF Development Package

To develop UDFs for Impala, download and install the `impala-udf-devel` package containing header files, sample source, and build configuration files. Start at http://archive.cloudera.com/impala/ and locate the appropriate `.repo` or list file for your operating system version, such as the `.repo` file for RHEL 6. Use the familiar `yum`, `zypper`, or `apt-get` commands depending on your operating system, with `impala-udf-devel` for the package name.

> **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 167 for how to code UDFs, and Examples of Creating and Using UDFs on page 172 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 166.

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.

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.
- Each kind of struct has a `null()` member function that returns an instance of the struct with the `is_null` flag set.
- Because Impala currently does not support composite or nested types, Impala cannot process UDFs that accept such 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);
```

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 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`, 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: https://github.com/cloudera/impala-udf-samples/blob/master/uda-sample.cc

`uda-sample.cc`:

See this file online at: https://github.com/cloudera/impala-udf-samples/blob/master/uda-sample.h

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 ships with a sample build environment for UDFs, that you can study, experiment with, and adapt for your own use. This sample build environment starts with the `cmake` configuration command, which 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 88 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.

Prerequisites for the build environment are:

```
# Use the appropriate package installation command for your Linux distribution.
sudo yum install gcc-c++ cmake boost-devel
sudo yum install impala-udf-devel
```

Then, 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 166.

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.
- Because incorrectly coded UDFs could cause performance or capacity problems, for example by going into infinite loops or allocating excessive amounts of memory, only an administrative user can create UDFs. That is, to execute the `CREATE FUNCTION` statement requires the `ALL` privilege on the server.

See Enabling Sentry Authorization for Impala 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.
- 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.
- When the `catalogd` process is restarted, all UDFs become undefined and must be reloaded.
- Impala currently does not support user-defined table functions (UDTFs).

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 180 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.

The current release of Impala does not support the following SQL features that you might be familiar with from HiveQL:

- Non-scalar data types such as maps, arrays, structs.
- Extensibility mechanisms such as `TRANSFORM`, custom file formats, or custom SerDes.
- XML and JSON functions.
- Certain aggregate functions from HiveQL: `var_pop`, `var_samp`, `covar_pop`, `covar_samp`, `corr`, `percentile`, `percentile_approx`, `histogram_numeric`, `collect_set`; Impala supports the set of aggregate functions listed in Aggregate Functions on page 158.
- Sampling.
- Lateral views.
- Multiple `DISTINCT` clauses per query.

Note:

Impala only allows a single `COUNT(DISTINCT` *`columns`*`)` expression in each query.

If you do not need precise accuracy, you can produce an estimate of the distinct values for a column by specifying `NDV(`*`column`*`)`; a query can contain multiple instances of `NDV(`*`column`*`)`.

To produce the same result as multiple `COUNT(DISTINCT)` expressions, you can use the following technique for queries involving a single table:

```
select v1.c1 result1, v2.c1 result2 from
  (select count(distinct col1) as c1 from t1) v1
    cross join
  (select count(distinct col2) as c1 from t1) v2;
```

Because `CROSS JOIN` is an expensive operation, prefer to use the `NDV()` technique wherever practical.

User-defined functions (UDFs) are supported starting in Impala 1.2. See User-Defined Functions (UDFs) on page 163 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.

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 PARTITIONS`
- `SHOW TABLE EXTENDED`
- `SHOW INDEXES`
- `SHOW COLUMNS`

Semantic Differences Between Impala and Hive 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 (incubating) 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.

Although Impala currently does not have DDL support for managing privileges, it can make use of privileges set up through Hive `GRANT` and `REVOKE` statements. See Enabling Sentry Authorization for Impala 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 on page 119 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 61 for details.

  See Data Types on page 49 for full details on implicit and explicit casting for all types, and Type Conversion Functions on page 145 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 GMT. This difference can produce different results for some calls to similarly named date/time functions between Impala and Hive. See Date and Time Functions on page 146 for details about the Impala functions.
- 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 impose any length constraints for strings, and does not do any special processing (such as blank-padding) for character data.

- 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 60 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 61 for information about the `TIMESTAMP` data type, and Date and Time Functions on page 146 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 61 for information about the allowed formats for date and time literals, and String Functions on page 153 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 61, `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 Date and Time Functions on page 146. 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 49 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 49 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 49 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 49 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 49 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 49 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 163 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 Conditional Functions on page 151 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 Conditional Functions on page 151 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 163 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 on page 193 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

Some SQL statements or clauses that you might be familiar with are not currently supported in Impala:

- Impala has no `DELETE` statement. Impala is intended for data warehouse-style operations where you do bulk moves and transforms of large quantities of data. Instead of using `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 79 for an overview of Impala DML statements.

- Impala has no `UPDATE` statement. Impala is intended for data warehouse-style operations where you do bulk moves and transforms of large quantities of data. Instead of using `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 79 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 Appendix C - Impala Reserved Words on page 285 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 recognized.
- Impala supports subqueries only in the `FROM` clause of a query, not within the `WHERE` clauses. Therefore, you cannot use clauses such as `WHERE` *`column`* `IN (`*`subquery`*`)`. Also, Impala does not allow `EXISTS` or `NOT EXISTS` clauses (although `EXISTS` is a reserved keyword).
- 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.
- Within queries, Impala requires query aliases for any subqueries:

  ```
  -- Without the alias 'contents_of_t1' at the end, query gives syntax error.
  select count(*) from (select * from t1) contents_of_t1;
  ```
- When an alias is declared for an expression in a query, that alias cannot be referenced again within the same query block:

  ```
  -- Can't reference AVERAGE twice in the SELECT list where it's defined.
  select avg(x) as average, average+1 from t1 group by x;
  ERROR: AnalysisException: couldn't resolve column reference: 'average'

  -- Although it can be referenced again later in the same query.
  select avg(x) as average from t1 group by x having average > 3;
  ```

  For Impala, either repeat the expression again, or abstract the expression into a `WITH` clause, creating named columns that can be referenced multiple times anywhere in the base query:

  ```
  -- The following 2 query forms are equivalent.
  select avg(x) as average, avg(x)+1 from t1 group by x;
  with avg_t as (select avg(x) average from t1 group by x) select average, average+1
   from avg_t;
  ```
- Impala does not support certain rarely used join types that are less appropriate for high-volume tables used for data warehousing. In some cases, Impala supports join types but requires explicit syntax to ensure you do not do inefficient joins of huge tables by accident. For example, Impala does not support natural joins or anti-joins, and requires the `CROSS JOIN` operator for Cartesian products. See Joins on page 119 for details on the syntax for Impala join clauses.
- Impala has 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 on page 233 for usage details, and CREATE TABLE Statement on page 90 and ALTER TABLE Statement on page 79 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. Remember, to distribute work for a query across a cluster, you need at least one HDFS block per node. HDFS blocks are typically multiple megabytes, up to 1 GB by default 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 "top-N" queries, Impala uses the `LIMIT` clause rather than comparing against a pseudocolumn named `ROWNUM` or `ROW_NUM`. See LIMIT Clause on page 128 for details.

SQL Constructs to Doublecheck

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, be alert and change these things 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 239 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 246, for details about the file format most heavily optimized for large-scale data warehouse queries.
- A `CREATE TABLE` statement with no `PARTITIONED BY` clause stores all the data files in the same physical location, which can lead to scalability problems when the data volume becomes large.

  On the other hand, 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 on page 233 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, but because each such statement creates a separate tiny file in HDFS, 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 74 and LOAD DATA Statement on page 114 for details about the Impala techniques for working with data files produced outside of Impala; see Data Loading and Querying Examples on page 26 for examples of ETL workflow for Impala.
- 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 105 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 239) or from an unpartitioned table into a partitioned one (see Partitioning on page 233).
- The number of expressions allowed in an Impala query might be smaller than for some other database systems, causing failures for very complicated queries (typically produced by automated SQL generators). Where practical, keep the number of expressions in the `WHERE` clauses to approximately 2000 or fewer. As a

workaround, set the query option `DISABLE_CODEGEN=true` if queries fail for this reason. See DISABLE_CODEGEN on page 193 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

Throughout this section, some of the decisions you make during the porting process also have a substantial impact on performance. After your SQL code is ported and working correctly, doublecheck 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.

- Have you run the `COMPUTE STATS` statement on each table involved in join queries? Have you also run `COMPUTE STATS` for each table used as the source table in an `INSERT ... SELECT` or `CREATE TABLE AS SELECT` statement?
- Are you using the most efficient file format for your data volumes, table structure, and query characteristics?
- Are you using partitioning effectively? That is, have you partitioned on columns that are often used for filtering in `WHERE` clauses? Have you partitioned at the right granularity so that there is enough data in each partition to parallelize the work for each query?
- Does your ETL process produce a relatively small number of multi-megabyte data files (good) rather than a huge number of small files (bad)?

See Tuning Impala for Performance on page 203 for details about the whole performance tuning process.

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 SQL Statements on page 78, 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 `-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.

For information on installing the Impala shell, see Impala Installation. In Cloudera Manager 4.1 and higher, 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 DataNode running the `impalad` daemon through the `impala-shell` command, see Connecting to impalad through impala-shell on page 189.

For a list of the `impala-shell` command-line options, see impala-shell Command-Line Options on page 187. For reference information about the `impala-shell` interactive commands, see impala-shell Command Reference on page 190.

impala-shell Command-Line Options

You can specify the following command-line options when starting the `impala-shell` command to change how shell commands are executed.

Note:

These options are different than the configuration options for the `impalad` daemon itself. For the `impalad` options, see Modifying Impala Startup Options.

Option	Explanation
-B or --delimited	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.
--print_header	

Option	Explanation
-o *filename* or --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*	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 and/or escaped. Added in Impala 1.0.1.
-p or --show_profiles	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	Displays help information.
-i *hostname* or --impalad=*hostname*	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*	Passes a query or other shell command from the command line. The shell 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*	Passes a SQL query from a file. Files must be semicolon (;) delimited.
-k or --kerberos	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.
-s *kerberos_service_name* or --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	Enables verbose output.
--quiet	Disables verbose output.
-v or --version	Displays version information.
-c	Continues on query failure.
-r or --refresh_after_connect	Refreshes Impala metadata upon connection. Same as running the REFRESH statement after connecting.

Option	Explanation
-d *default_db* or --database=*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`: enables 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 SSL, but does not validate the server certificate. This is useful for connecting to a known-good Impala that is only running over SSL, when a copy of the certificate is not available (such as when debugging customer installations).
-l	Enables LDAP authentication.
-u	Supplies the user name, when LDAP authentication is enabled by the `-l` option. (Specify the short user name, not the full LDAP distinguished name.) The shell then prompts interactively for the password.
--strict_unicode	Causes the shell to ignore invalid Unicode code points in input strings.

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, or during an `impala-shell` session by issuing a `CONNECT` command. You can connect to any DataNode where an instance of `impalad` is running, and that node coordinates the execution of all queries sent to it.

For simplicity, you might always connect to the same node, perhaps running `impala-shell` on the same node as `impalad` and specifying the host name as `localhost`. Routing all SQL statements to the same node can help to avoid issuing frequent `REFRESH` statements, as is necessary when table data or metadata is updated through a different node.

For load balancing or general flexibility, you might connect to an arbitrary node for each `impala-shell` session. In this case, depending on whether table data or metadata might have been updated through another node, you might issue a `REFRESH` statement to bring the metadata for all tables up to date on this node (for a long-lived session that will query many tables) or issue specific `REFRESH` *`table_name`* statements just for the tables you intend to query.

To connect the Impala shell to any DataNode with an `impalad` daemon:

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) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.4.x (hash) built on date)
[Not connected] >
```

2. 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 host name you have configured for any DataNode running Impala in your environment. The changed prompt indicates a successful connection.

Running Commands in impala-shell

For information on available commands, see impala-shell Command Reference on page 190. You can see the full set of available commands by pressing TAB twice:

```
[impalad-host:21000] >
connect    describe  explain   help      history   insert    quit      refresh   select
    set        shell     show      use       version
[impalad-host:21000] >
```

> **Note:** Commands must be terminated by a semi-colon. A command can span multiple lines.

For example:

```
[impalad-host:21000] > select * from alltypessmall limit 5
Query: select * from alltypessmall limit 5
Query finished, fetching results ...
2009    3       50      true    0       0       0       0       0       0       03/01/09
        0       2009-03-01 00:00:00
2009    3       51      false   1       1       1       10      1.100000023841858
   10.1    03/01/09        1       2009-03-01 00:01:00
2009    3       52      true    2       2       2       20      2.200000047683716
   20.2    03/01/09        2       2009-03-01 00:02:00.100000000
2009    3       53      false   3       3       3       30      3.299999952316284
   30.3    03/01/09        3       2009-03-01 00:03:00.300000000
2009    3       54      true    4       4       4       40      4.400000095367432
   40.4    03/01/09        4       2009-03-01 00:04:00.600000000
Returned 5 row(s) in 0.10s
[impalad-host:21000] >
```

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 79 and ALTER VIEW Statement on page 83 for details.
`compute stats`	Gathers important performance-related information for a table, used by Impala to optimize queries. See COMPUTE STATS Statement on page 84 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

Command	Explanation
	alternate port specified by the `--fe_port` flag, you must provide that alternate port. See Connecting to impalad through impala-shell on page 189 for examples. The `SET` command 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 subsequent `CONNECT` commands to connect to different Impala servers,
`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 96 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 101, DROP VIEW Statement on page 102, DROP DATABASE Statement on page 100, and DROP FUNCTION Statement on page 101 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 103 and Using the EXPLAIN Plan for Performance Tuning on page 224 for details.
`help`	Help provides a list of all available commands and options.
`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 105 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 111 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 226 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 116 for details.
`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 118 for details.
`set`	Manages query options for an `impala-shell` session. The available options are the ones listed in Query Options for the SET Command on page 192. 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` commands in the same session. To modify option values, issue commands with the syntax `set` *`option`*`=`*`value`*. To restore an option to its default,

Command	Explanation
	use the `unset` command. Some options take Boolean values of `true` and `false`. Others take numeric arguments, or quoted string values. The `SET` command 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 subsequent `CONNECT` commands to connect to different Impala servers,
`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.
`show`	Displays metastore data for schema objects created and accessed through Impala, Hive, or both. show can be used to gather information about databases or tables by following the show command with one of those choices. See SHOW Statement on page 135 for details.
`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 225 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 Command on page 192 for the available query options.
`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 138 for details. Not effective with the `-q` option, because that option only allows a single statement in the argument.
`version`	Returns Impala version information.

Query Options for the SET Command

You can specify the following options within an `impala-shell` session, 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.

Note: Currently, there is no way to set query options directly through the JDBC and ODBC interfaces. For JDBC and ODBC applications, you can execute queries that need specific query options by invoking `impala-shell` to run a script that starts with `SET` commands, or by defining query options globally through the `impalad` startup flag `--default_query_options`.

ABORT_ON_DEFAULT_LIMIT_EXCEEDED

Now that the `ORDER BY` clause no longer requires an accompanying `LIMIT` clause in Impala 1.4.0 and higher, this query option is deprecated and has no effect.

Type: Boolean

Default: `false` (shown as 0 in output of `SET` command)

ABORT_ON_ERROR

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 enabled by default to hep you gather maximum diagnostic information when an errors 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`

Default: `false` (shown as 0 in output of `SET` command)

ALLOW_UNSUPPORTED_FORMATS

An obsolete query option from early work on support for file formats. Do not use. Might be removed in the future.

Type: `BOOLEAN`

Default: `false` (shown as 0 in output of `SET` command)

BATCH_SIZE

Number of rows evaluated at a time by SQL operators. Unspecified or a size of 0 uses a predefined default size. Primarily for Cloudera testing.

Default: 0 (meaning 1024)

DEBUG_ACTION

Introduces artificial problem conditions within queries. For internal Cloudera debugging and troubleshooting.

Type: `STRING`

Default: empty string

DEFAULT_ORDER_BY_LIMIT

Now that the `ORDER BY` clause no longer requires an accompanying `LIMIT` clause in Impala 1.4.0 and higher, this query option is deprecated and has no effect.

Prior to Impala 1.4.0, Impala queries that use the `ORDER BY` clause must also include a `LIMIT` clause, to avoid accidentally producing huge result sets that must be sorted. Sorting a huge result set is a memory-intensive operation. In Impala 1.4.0 and higher, Impala uses a temporary disk work area to perform the sort if that operation would otherwise exceed the Impala memory limit on a particular host.

Default: -1 (no default limit)

DISABLE_CODEGEN

This is a debug option, intended for diagnosing and working around issues that cause crashes. 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. Do not otherwise run with this setting turned on, because it results in lower overall performance.

Because the code generation phase adds a small amount of overhead for each query, you might turn on the DISABLE_CODEGEN option to achieve maximum throughput when running many short-lived queries against small tables.

Type: BOOLEAN

Default: false (shown as 0 in output of SET command)

EXPLAIN_LEVEL

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 (might be incorrectly reported as 0 in output of SET command)

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.
- 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 Impala Administration on page 33. See EXPLAIN Statement on page 103 for the syntax of the EXPLAIN statement, and Using the EXPLAIN Plan for Performance Tuning on page 224 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 How Impala Uses Statistics for Query Optimization on page 212 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

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 region server, in conjunction with the HBASE_CACHING query option. See HBASE_CACHING on page 199 for details.

Type: BOOLEAN

Default: false (shown as 0 in output of SET command)

HBASE_CACHING

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 region server, in conjunction with the HBASE_CACHE_BLOCKS query option. See HBASE_CACHE_BLOCKS on page 199 for details.

Type: BOOLEAN

Default: 0

MAX_ERRORS

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.

Default: 0 (meaning 1000 errors)

MAX_IO_BUFFERS

Deprecated query option. Currently has no effect.

Default: 0

MAX_SCAN_RANGE_LENGTH

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, typically several megabytes for most file formats, or 1 GB for Parquet tables.

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.

Default: 0

MEM_LIMIT

When resource management is not enabled, defines the maximum amount of memory a query can allocate on each node. If query processing exceeds the specified memory limit on any node, 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.

When resource management is enabled in CDH 5, the mechanism for this option changes. If set, it overrides the automatic memory estimate from Impala. Impala requests this amount of memory from YARN on each node, and the query does not proceed until that much memory is available. The actual memory used by the query could be lower, since some queries use much less memory than others. With resource management, the `MEM_LIMIT` setting acts both as a hard limit on the amount of memory a query can use on any node (enforced by YARN and a guarantee that that much memory will be available on each node while the query is being executed. When resource management is enabled but no `MEM_LIMIT` setting is specified, Impala estimates the amount of memory needed on each node for each query, requests that much memory from YARN before starting the query, and then internally sets the `MEM_LIMIT` on each node to the requested amount of memory during the query. Thus, if the query takes more memory than was originally estimated, Impala detects that the `MEM_LIMIT` is exceeded and cancels the query itself.

Default: 0

NUM_NODES

Limit the number of nodes that process a query, typically during debugging. Only accepts the values 0 (meaning all nodes) or 1 (meaning all work is done on the coordinator node). 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.

Default: 0

NUM_SCANNER_THREADS

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.

Default: 0

Note: Currently, a known issue (IMPALA-488) could cause excessive memory usage during a `COMPUTE STATS` operation on a Parquet table. As a workaround, issue the command `SET NUM_SCANNER_THREADS=2` in `impala-shell` before issuing the `COMPUTE STATS` statement. Then issue `UNSET NUM_SCANNER_THREADS` before continuing with queries.

PARQUET_COMPRESSION_CODEC

When Impala writes Parquet data files using the `INSERT` statement, the underlying compression is controlled by the `PARQUET_COMPRESSION_CODEC` query option. The allowed values for this query option are `SNAPPY` (the default), `GZIP`, and `NONE`. The option value is not case-sensitive. See Snappy and GZip Compression for Parquet Data Files on page 249 for details and examples.

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.

Default: SNAPPY

Related information:

For information about the Parquet file format, and how compressing the data files affects query performance, see Using the Parquet File Format with Impala Tables on page 246.

PARQUET_FILE_SIZE

Specifies the maximum size of each Parquet data file produced by Impala `INSERT` statements. For small or partitioned tables where the default Parquet block size of 1 GB is much larger than needed for each data file, you can increase parallelism by specifying a smaller size, resulting in more HDFS blocks that can be processed by different nodes. Reducing the file size also reduces the memory required to buffer each block before writing it to disk.

Specify the size in bytes, for example:

```
set PARQUET_FILE_SIZE=128000000;
INSERT INTO parquet_table SELECT * FROM text_table;
```

Default: 0 (produces files with a maximum size of 1 gigabyte)

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 246.

REQUEST_POOL

The pool or queue name that queries should be submitted to. Only applies when you enable the Impala admission control feature (CDH 4 or CDH 5; see Admission Control and Query Queuing on page 33), or the YARN resource management feature (CDH 5 only; see Using YARN Resource Management with Impala (CDH 5 Only) on page 41). Specifies the name of the pool used by requests from Impala to the resource manager.

Formerly known as `YARN_POOL` during the CDH 5 beta period. Renamed to reflect that it can be used both with YARN and with the lightweight admission control feature introduced in Impala 1.3.

Default: empty (use the user-to-pool mapping defined by an `impalad` startup option in the Impala configuration file)

RESERVATION_REQUEST_TIMEOUT (CDH 5 Only)

Maximum number of milliseconds Impala will wait for a reservation to be completely granted or denied. Used in conjunction with the Impala resource management feature in Impala 1.2 and higher with CDH 5.

Default: 300000 (5 minutes)

SUPPORT_START_OVER

Leave this setting `false`.

Default: false

SYNC_DDL

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.)

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.

Default: false

V_CPU_CORES (CDH 5 Only)

The number of per-host virtual CPU cores to request from YARN. If set, the query option overrides the automatic estimate from Impala. Used in conjunction with the Impala resource management feature in Impala 1.2 and higher and CDH 5.

Default: 0 (use automatic estimates)

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 and software settings from Post-Installation Configuration for Impala.

- Partitioning on page 233. 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 206. 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 Column Statistics on page 213 and Table Statistics on page 212 are also important primarily for join performance.
- Table Statistics on page 212 and Column Statistics on page 213. 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 222. Do some post-setup testing to ensure Impala is using optimal settings for performance, before conducting any benchmark tests.
- Benchmarking Impala Queries on page 217. The configuration and sample data that you use for initial experiments with Impala is often not appropriate for doing performance tests.
- Controlling Resource Usage on page 217. 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.

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 239 for comparisons of all file formats supported by

Impala, and Using the Parquet File Format with Impala Tables on page 246 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 105 for examples of the `INSERT ... SELECT` syntax.

For example, if you have thousands of partitions in a Parquet table, each with less than 1 GB 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 1 GB 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 Date and Time Functions on page 146 for details.

See Partitioning on page 233 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 1 GB block size. Each file is a single block, allowing the whole file to be processed by a single node. 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 files 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 file is processed by a single core on one of the data nodes. In a 100-node cluster of 16-core machines, you could potentially process 1600 data files simultaneously. You want to find a sweet spot between “many time 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 to less than 1 GB, such as 256 MB or 128 MB. (Specify the file size as an absolute number of bytes.) 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 206 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 Command-Line Options on page 187 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 103 and Using the EXPLAIN Plan for Performance Tuning on page 224 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 226 for details.

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` keyword. 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 Hints on page 133 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` keyword. 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. In this case, rewrite the query so that the largest table is on the left, followed by the next largest, and so on until the smallest table is on the right.

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;
```

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.

How Impala Uses Statistics for Query Optimization

Impala can do better optimization for complex or multi-table queries when statistics are available, to better understand the volume of data and how the values are distributed, and use this information to help parallelize and distribute the work for a query. The following sections describe the categories of statistics Impala can work with, and how to produce them and keep them up to date.

Originally, Impala relied on the Hive mechanism for collecting statistics, through the Hive `ANALYZE TABLE` statement which initiates a MapReduce job. For better user-friendliness and reliability, Impala implements its own `COMPUTE STATS` statement in Impala 1.2.2 and higher, along with the `SHOW TABLE STATS` and `SHOW COLUMN STATS` statements.

Table Statistics

The Impala query planner can make use of statistics about entire tables and partitions when that metadata is available in the metastore database. This metadata is used on its own for certain optimizations, and used in combination with column statistics for other optimizations.

To gather table statistics after loading data into a table or partition, use one of the following techniques:

- Issue the statement `COMPUTE STATS` in Impala. This statement, new in Impala 1.2.2, is the preferred method because:
  - It gathers table statistics and statistics for all partitions and columns in a single operation.
  - It does not rely on any special Hive settings, metastore configuration, or separate database to hold the statistics.
  - If you need to adjust statistics incrementally for an existing table, such as after adding a partition or inserting new data, you can use an `ALTER TABLE` statement such as:

    ```
    alter table analysis_data set tblproperties('numRows'='new_value');
    ```

    to update that one property rather than re-processing the whole table.
- Load the data through the `INSERT OVERWRITE` statement in Hive, while the Hive setting **hive.stats.autogather** is enabled.
- Issue an `ANALYZE TABLE` statement in Hive, for the entire table or a specific partition.

  ```
  ANALYZE TABLE tablename [PARTITION(partcol1[=val1], partcol2[=val2], ...)] COMPUTE
   STATISTICS [NOSCAN];
  ```

  For example, to gather statistics for a non-partitioned table:

  ```
  ANALYZE TABLE customer COMPUTE STATISTICS;
  ```

  To gather statistics for a `store` table partitioned by state and city, and both of its partitions:

  ```
  ANALYZE TABLE store PARTITION(s_state, s_county) COMPUTE STATISTICS;
  ```

To gather statistics for the `store` table and only the partitions for California:

```
ANALYZE TABLE store PARTITION(s_state='CA', s_county) COMPUTE 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 135 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. Cloudera recommends using the Impala `COMPUTE STATS` statement to avoid potential configuration and scalability issues with the statistics-gathering process.

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. Currently, Impala does not create this metadata itself. Use the `ANALYZE TABLE` statement in the Hive shell to gather these statistics. (This statement works from Hive whether you create the table in Impala or in Hive.)

Note:

For column statistics to be effective in Impala, you also need to have table statistics for the applicable tables, as described in Table Statistics on page 212. If 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.

Currently, the `COMPUTE STATS` statement under CDH 4 does not store any statistics for `DECIMAL` columns. When Impala runs under CDH 5, which has better support for `DECIMAL` in the metastore database, `COMPUTE STATS` does collect statistics for `DECIMAL` columns and Impala uses the statistics to optimize query performance.

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 make use of 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 135 and EXPLAIN Statement on page 103 for details.

Setting Statistics Manually through ALTER TABLE

The most crucial piece of data in all the statistics is the number of rows in the table (for an unpartitioned table) or 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 an entire `COMPUTE 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:

```
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');
```

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 1000000 rows, and we add another partition,
-- change the numRows property for the partition and the overall table.
alter table partitioned_data partition(year=2009, month=4) set tblproperties
('numRows'='30000');
alter table partitioned_data set tblproperties ('numRows'='1030000');
```

In practice, the `COMPUTE STATS` statement should be fast enough that this technique is not needed. It is most useful as a workaround for in case of performance issues where you might adjust the `numRows` value higher or lower to produce the ideal join order.

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_precentage   | 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_precentage  | 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 207. 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 PROFILEoutput, 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 Command-Line Options on page 187 for details.

Controlling 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, and works with both CDH 4 and CDH 5. See Admission Control and Query Queuing on page 33 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 Managing Clusters with Cloudera Manager.
- When you use Impala in combination with CDH 5, you can use the YARN resource management framework in combination with the Llama service, as explained in Using YARN Resource Management with Impala (CDH 5 Only) on page 41.

Using HDFS Caching with Impala (CDH 5.1 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 data nodes, 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.

For background information about how to set up and manage HDFS caching for a CDH cluster, see the CDH documentation.

Overview of HDFS Caching for Impala

On CDH 5.1 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.

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. (Any data block is only cached on one host. Once a data block is cached on one host, all requests to process that block are routed to that same host.)
- 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 CDH documentation.

Once HDFS caching is enabled and one or more pools are available, see Enabling HDFS Caching for Impala Tables and Partitions on page 218 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:

```
-- 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';

-- 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 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 90 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. When files are added to a table or partition whose contents are cached, Impala automatically detects those changes and performs a `REFRESH` automatically once the relevant data is cached.

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 data nodes 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 135 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. The `SHOW TABLES` command also now has a `SHOW CACHED TABLES` variant to examine which Impala tables are currently being cached.

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.

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 data node) 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.

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. If you need 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 starting Impala and executing queries can be found in Starting Impala and Using the Impala Shell (impala-shell Command) on page 187. Information on logging can be found in Using Impala Logging on page 275. 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 266 for examples of interpreting `EXPLAIN` plans for queries against HBase tables.

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 on page 194 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 226, 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 normal to verbose (or from 0 to 1) 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 226, 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 statement, 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 224, 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 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.

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.

Here is an example of a query profile, from a relatively straightforward query on a single-node pseudo-distributed cluster to keep the output relatively brief.

```
[localhost:21000] > profile;
Query Runtime Profile:
Query (id=6540a03d4bee0691:4963d6269b210ebd):
  Summary:
    Session ID: ea4a197f1c7bf858:c74e66f72e3a33ba
    Session Type: BEESWAX
    Start Time: 2013-12-02 17:10:30.263067000
    End Time: 2013-12-02 17:10:50.932044000
    Query Type: QUERY
    Query State: FINISHED
    Query Status: OK
    Impala Version: impalad version 1.2.1 RELEASE (build
edb5af1bcad63d410bc5d47cc203df3a880e9324)
    User: cloudera
    Network Address: 127.0.0.1:49161
    Default Db: stats_testing
    Sql Statement: select t1.s, t2.s from t1 join t2 on (t1.id = t2.parent)
    Plan:
----------------
Estimated Per-Host Requirements: Memory=2.09GB VCores=2

PLAN FRAGMENT 0
  PARTITION: UNPARTITIONED

  4:EXCHANGE
     cardinality: unavailable
     per-host memory: unavailable
     tuple ids: 0 1

PLAN FRAGMENT 1
  PARTITION: RANDOM

  STREAM DATA SINK
    EXCHANGE ID: 4
    UNPARTITIONED

  2:HASH JOIN
  |  join op: INNER JOIN (BROADCAST)
  |  hash predicates:
  |    t1.id = t2.parent
  |  cardinality: unavailable
```

```
   |  per-host memory: 2.00GB
   |  tuple ids: 0 1
   |
   |----3:EXCHANGE
   |       cardinality: unavailable
   |       per-host memory: 0B
   |       tuple ids: 1
   |
  0:SCAN HDFS
     table=stats_testing.t1 #partitions=1/1 size=33B
     table stats: unavailable
     column stats: unavailable
     cardinality: unavailable
     per-host memory: 32.00MB
     tuple ids: 0

PLAN FRAGMENT 2
  PARTITION: RANDOM

  STREAM DATA SINK
    EXCHANGE ID: 3
    UNPARTITIONED

  1:SCAN HDFS
     table=stats_testing.t2 #partitions=1/1 size=960.00KB
     table stats: unavailable
     column stats: unavailable
     cardinality: unavailable
     per-host memory: 96.00MB
     tuple ids: 1
----------------
    Query Timeline: 20s670ms
       - Start execution: 2.559ms (2.559ms)
       - Planning finished: 23.587ms (21.27ms)
       - Rows available: 666.199ms (642.612ms)
       - First row fetched: 668.919ms (2.719ms)
       - Unregister query: 20s668ms (20s000ms)
  ImpalaServer:
     - ClientFetchWaitTimer: 19s637ms
     - RowMaterializationTimer: 167.121ms
  Execution Profile 6540a03d4bee0691:4963d6269b210ebd:(Active: 837.815ms, % non-child:
 0.00%)
    Per Node Peak Memory Usage: impala-1.example.com:22000(7.42 MB)
     - FinalizationTimer: 0ns
    Coordinator Fragment:(Active: 195.198ms, % non-child: 0.00%)
      MemoryUsage(500.0ms): 16.00 KB, 7.42 MB, 7.33 MB, 7.10 MB, 6.94 MB, 6.71 MB, 6.56
 MB, 6.40 MB, 6.17 MB, 6.02 MB, 5.79 MB, 5.63 MB, 5.48 MB, 5.25 MB, 5.09 MB, 4.86 MB,
 4.71 MB, 4.47 MB, 4.32 MB, 4.09 MB, 3.93 MB, 3.78 MB, 3.55 MB, 3.39 MB, 3.16 MB, 3.01
 MB, 2.78 MB, 2.62 MB, 2.39 MB, 2.24 MB, 2.08 MB, 1.85 MB, 1.70 MB, 1.54 MB, 1.31 MB,
 1.16 MB, 948.00 KB, 790.00 KB, 553.00 KB, 395.00 KB, 237.00 KB
      ThreadUsage(500.0ms): 1
       - AverageThreadTokens: 1.00
       - PeakMemoryUsage: 7.42 MB
       - PrepareTime: 36.144us
       - RowsProduced: 98.30K (98304)
       - TotalCpuTime: 20s449ms
       - TotalNetworkWaitTime: 191.630ms
       - TotalStorageWaitTime: 0ns
      CodeGen:(Active: 150.679ms, % non-child: 77.19%)
         - CodegenTime: 0ns
         - CompileTime: 139.503ms
         - LoadTime: 10.7ms
         - ModuleFileSize: 95.27 KB
      EXCHANGE_NODE (id=4):(Active: 194.858ms, % non-child: 99.83%)
         - BytesReceived: 2.33 MB
         - ConvertRowBatchTime: 2.732ms
         - DataArrivalWaitTime: 191.118ms
         - DeserializeRowBatchTimer: 14.943ms
         - FirstBatchArrivalWaitTime: 191.117ms
         - PeakMemoryUsage: 7.41 MB
         - RowsReturned: 98.30K (98304)
         - RowsReturnedRate: 504.49 K/sec
         - SendersBlockedTimer: 0ns
```

```
          - SendersBlockedTotalTimer(*): 0ns
     Averaged Fragment 1:(Active: 442.360ms, % non-child: 0.00%)
       split sizes:  min: 33.00 B, max: 33.00 B, avg: 33.00 B, stddev: 0.00
       completion times: min:443.720ms  max:443.720ms  mean: 443.720ms  stddev:0ns
       execution rates: min:74.00 B/sec  max:74.00 B/sec  mean:74.00 B/sec  stddev:0.00
  /sec
       num instances: 1
        - AverageThreadTokens: 1.00
        - PeakMemoryUsage: 6.06 MB
        - PrepareTime: 7.291ms
        - RowsProduced: 98.30K (98304)
        - TotalCpuTime: 784.259ms
        - TotalNetworkWaitTime: 388.818ms
        - TotalStorageWaitTime: 3.934ms
       CodeGen:(Active: 312.862ms, % non-child: 70.73%)
          - CodegenTime: 2.669ms
          - CompileTime: 302.467ms
          - LoadTime: 9.231ms
          - ModuleFileSize: 95.27 KB
       DataStreamSender (dst_id=4):(Active: 80.63ms, % non-child: 18.10%)
          - BytesSent: 2.33 MB
          - NetworkThroughput(*): 35.89 MB/sec
          - OverallThroughput: 29.06 MB/sec
          - PeakMemoryUsage: 5.33 KB
          - SerializeBatchTime: 26.487ms
          - ThriftTransmitTime(*): 64.814ms
          - UncompressedRowBatchSize: 6.66 MB
       HASH_JOIN_NODE (id=2):(Active: 362.25ms, % non-child: 3.92%)
          - BuildBuckets: 1.02K (1024)
          - BuildRows: 98.30K (98304)
          - BuildTime: 12.622ms
          - LoadFactor: 0.00
          - PeakMemoryUsage: 6.02 MB
          - ProbeRows: 3
          - ProbeTime: 3.579ms
          - RowsReturned: 98.30K (98304)
          - RowsReturnedRate: 271.54 K/sec
         EXCHANGE_NODE (id=3):(Active: 344.680ms, % non-child: 77.92%)
            - BytesReceived: 1.15 MB
            - ConvertRowBatchTime: 2.792ms
            - DataArrivalWaitTime: 339.936ms
            - DeserializeRowBatchTimer: 9.910ms
            - FirstBatchArrivalWaitTime: 199.474ms
            - PeakMemoryUsage: 156.00 KB
            - RowsReturned: 98.30K (98304)
            - RowsReturnedRate: 285.20 K/sec
            - SendersBlockedTimer: 0ns
            - SendersBlockedTotalTimer(*): 0ns
       HDFS_SCAN_NODE (id=0):(Active: 13.616us, % non-child: 0.00%)
          - AverageHdfsReadThreadConcurrency: 0.00
          - AverageScannerThreadConcurrency: 0.00
          - BytesRead: 33.00 B
          - BytesReadLocal: 33.00 B
          - BytesReadShortCircuit: 33.00 B
          - NumDisksAccessed: 1
          - NumScannerThreadsStarted: 1
          - PeakMemoryUsage: 46.00 KB
          - PerReadThreadRawHdfsThroughput: 287.52 KB/sec
          - RowsRead: 3
          - RowsReturned: 3
          - RowsReturnedRate: 220.33 K/sec
          - ScanRangesComplete: 1
          - ScannerThreadsInvoluntaryContextSwitches: 26
          - ScannerThreadsTotalWallClockTime: 55.199ms
            - DelimiterParseTime: 2.463us
            - MaterializeTupleTime(*): 1.226us
            - ScannerThreadsSysTime: 0ns
            - ScannerThreadsUserTime: 42.993ms
          - ScannerThreadsVoluntaryContextSwitches: 1
          - TotalRawHdfsReadTime(*): 112.86us
          - TotalReadThroughput: 0.00 /sec
     Averaged Fragment 2:(Active: 190.120ms, % non-child: 0.00%)
       split sizes:  min: 960.00 KB, max: 960.00 KB, avg: 960.00 KB, stddev: 0.00
```

```
      completion times: min:191.736ms  max:191.736ms  mean: 191.736ms  stddev:0ns
      execution rates: min:4.89 MB/sec  max:4.89 MB/sec  mean:4.89 MB/sec  stddev:0.00
 /sec
      num instances: 1
       - AverageThreadTokens: 0.00
       - PeakMemoryUsage: 906.33 KB
       - PrepareTime: 3.67ms
       - RowsProduced: 98.30K (98304)
       - TotalCpuTime: 403.351ms
       - TotalNetworkWaitTime: 34.999ms
       - TotalStorageWaitTime: 108.675ms
      CodeGen:(Active: 162.57ms, % non-child: 85.24%)
         - CodegenTime: 3.133ms
         - CompileTime: 148.316ms
         - LoadTime: 12.317ms
         - ModuleFileSize: 95.27 KB
      DataStreamSender (dst_id=3):(Active: 70.620ms, % non-child: 37.14%)
         - BytesSent: 1.15 MB
         - NetworkThroughput(*): 23.30 MB/sec
         - OverallThroughput: 16.23 MB/sec
         - PeakMemoryUsage: 5.33 KB
         - SerializeBatchTime: 22.69ms
         - ThriftTransmitTime(*): 49.178ms
         - UncompressedRowBatchSize: 3.28 MB
      HDFS_SCAN_NODE (id=1):(Active: 118.839ms, % non-child: 62.51%)
         - AverageHdfsReadThreadConcurrency: 0.00
         - AverageScannerThreadConcurrency: 0.00
         - BytesRead: 960.00 KB
         - BytesReadLocal: 960.00 KB
         - BytesReadShortCircuit: 960.00 KB
         - NumDisksAccessed: 1
         - NumScannerThreadsStarted: 1
         - PeakMemoryUsage: 869.00 KB
         - PerReadThreadRawHdfsThroughput: 130.21 MB/sec
         - RowsRead: 98.30K (98304)
         - RowsReturned: 98.30K (98304)
         - RowsReturnedRate: 827.20 K/sec
         - ScanRangesComplete: 15
         - ScannerThreadsInvoluntaryContextSwitches: 34
         - ScannerThreadsTotalWallClockTime: 189.774ms
           - DelimiterParseTime: 15.703ms
           - MaterializeTupleTime(*): 3.419ms
           - ScannerThreadsSysTime: 1.999ms
           - ScannerThreadsUserTime: 44.993ms
         - ScannerThreadsVoluntaryContextSwitches: 118
         - TotalRawHdfsReadTime(*): 7.199ms
         - TotalReadThroughput: 0.00 /sec
    Fragment 1:
      Instance 6540a03d4bee0691:4963d6269b210ebf
(host=impala-1.example.com:22000):(Active: 442.360ms, % non-child: 0.00%)
        Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:1/33.00 B
        MemoryUsage(500.0ms): 69.33 KB
        ThreadUsage(500.0ms): 1
         - AverageThreadTokens: 1.00
         - PeakMemoryUsage: 6.06 MB
         - PrepareTime: 7.291ms
         - RowsProduced: 98.30K (98304)
         - TotalCpuTime: 784.259ms
         - TotalNetworkWaitTime: 388.818ms
         - TotalStorageWaitTime: 3.934ms
        CodeGen:(Active: 312.862ms, % non-child: 70.73%)
           - CodegenTime: 2.669ms
           - CompileTime: 302.467ms
           - LoadTime: 9.231ms
           - ModuleFileSize: 95.27 KB
        DataStreamSender (dst_id=4):(Active: 80.63ms, % non-child: 18.10%)
           - BytesSent: 2.33 MB
           - NetworkThroughput(*): 35.89 MB/sec
           - OverallThroughput: 29.06 MB/sec
           - PeakMemoryUsage: 5.33 KB
           - SerializeBatchTime: 26.487ms
           - ThriftTransmitTime(*): 64.814ms
           - UncompressedRowBatchSize: 6.66 MB
        HASH_JOIN_NODE (id=2):(Active: 362.25ms, % non-child: 3.92%)
          ExecOption: Build Side Codegen Enabled, Probe Side Codegen Enabled, Hash
Table Built Asynchronously
           - BuildBuckets: 1.02K (1024)
           - BuildRows: 98.30K (98304)
           - BuildTime: 12.622ms
           - LoadFactor: 0.00
           - PeakMemoryUsage: 6.02 MB
           - ProbeRows: 3
           - ProbeTime: 3.579ms
           - RowsReturned: 98.30K (98304)
           - RowsReturnedRate: 271.54 K/sec
          EXCHANGE_NODE (id=3):(Active: 344.680ms, % non-child: 77.92%)
             - BytesReceived: 1.15 MB
             - ConvertRowBatchTime: 2.792ms
             - DataArrivalWaitTime: 339.936ms
             - DeserializeRowBatchTimer: 9.910ms
             - FirstBatchArrivalWaitTime: 199.474ms
             - PeakMemoryUsage: 156.00 KB
             - RowsReturned: 98.30K (98304)
             - RowsReturnedRate: 285.20 K/sec
             - SendersBlockedTimer: 0ns
             - SendersBlockedTotalTimer(*): 0ns
        HDFS_SCAN_NODE (id=0):(Active: 13.616us, % non-child: 0.00%)
          Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:1/33.00 B
          Hdfs Read Thread Concurrency Bucket: 0:0% 1:0%
          File Formats: TEXT/NONE:1
          ExecOption: Codegen enabled: 1 out of 1
           - AverageHdfsReadThreadConcurrency: 0.00
           - AverageScannerThreadConcurrency: 0.00
           - BytesRead: 33.00 B
           - BytesReadLocal: 33.00 B
           - BytesReadShortCircuit: 33.00 B
           - NumDisksAccessed: 1
           - NumScannerThreadsStarted: 1
           - PeakMemoryUsage: 46.00 KB
           - PerReadThreadRawHdfsThroughput: 287.52 KB/sec
           - RowsRead: 3
           - RowsReturned: 3
           - RowsReturnedRate: 220.33 K/sec
           - ScanRangesComplete: 1
           - ScannerThreadsInvoluntaryContextSwitches: 26
           - ScannerThreadsTotalWallClockTime: 55.199ms
             - DelimiterParseTime: 2.463us
             - MaterializeTupleTime(*): 1.226us
             - ScannerThreadsSysTime: 0ns
             - ScannerThreadsUserTime: 42.993ms
           - ScannerThreadsVoluntaryContextSwitches: 1
           - TotalRawHdfsReadTime(*): 112.86us
           - TotalReadThroughput: 0.00 /sec
    Fragment 2:
      Instance 6540a03d4bee0691:4963d6269b210ec0
(host=impala-1.example.com:22000):(Active: 190.120ms, % non-child: 0.00%)
        Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:15/960.00 KB
         - AverageThreadTokens: 0.00
         - PeakMemoryUsage: 906.33 KB
         - PrepareTime: 3.67ms
         - RowsProduced: 98.30K (98304)
         - TotalCpuTime: 403.351ms
         - TotalNetworkWaitTime: 34.999ms
         - TotalStorageWaitTime: 108.675ms
        CodeGen:(Active: 162.57ms, % non-child: 85.24%)
           - CodegenTime: 3.133ms
           - CompileTime: 148.316ms
           - LoadTime: 12.317ms
           - ModuleFileSize: 95.27 KB
        DataStreamSender (dst_id=3):(Active: 70.620ms, % non-child: 37.14%)
           - BytesSent: 1.15 MB
           - NetworkThroughput(*): 23.30 MB/sec
           - OverallThroughput: 16.23 MB/sec
           - PeakMemoryUsage: 5.33 KB
           - SerializeBatchTime: 22.69ms
           - ThriftTransmitTime(*): 49.178ms
```

```
            - UncompressedRowBatchSize: 3.28 MB
         HDFS_SCAN_NODE (id=1):(Active: 118.839ms, % non-child: 62.51%)
           Hdfs split stats (<volume id>:<# splits>/<split lengths>): 0:15/960.00 KB
           Hdfs Read Thread Concurrency Bucket: 0:0% 1:0%
           File Formats: TEXT/NONE:15
           ExecOption: Codegen enabled: 15 out of 15
            - AverageHdfsReadThreadConcurrency: 0.00
            - AverageScannerThreadConcurrency: 0.00
            - BytesRead: 960.00 KB
            - BytesReadLocal: 960.00 KB
            - BytesReadShortCircuit: 960.00 KB
            - NumDisksAccessed: 1
            - NumScannerThreadsStarted: 1
            - PeakMemoryUsage: 869.00 KB
            - PerReadThreadRawHdfsThroughput: 130.21 MB/sec
            - RowsRead: 98.30K (98304)
            - RowsReturned: 98.30K (98304)
            - RowsReturnedRate: 827.20 K/sec
            - ScanRangesComplete: 15
            - ScannerThreadsInvoluntaryContextSwitches: 34
            - ScannerThreadsTotalWallClockTime: 189.774ms
              - DelimiterParseTime: 15.703ms
              - MaterializeTupleTime(*): 3.419ms
              - ScannerThreadsSysTime: 1.999ms
              - ScannerThreadsUserTime: 44.993ms
            - ScannerThreadsVoluntaryContextSwitches: 118
            - TotalRawHdfsReadTime(*): 7.199ms
            - TotalReadThroughput: 0.00 /sec
```

Partitioning

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 28 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 248 for performance considerations for partitioned Parquet tables.

See NULL on page 64 for details about how `NULL` values are represented in partitioned tables.

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.
- `ALTER TABLE`: you can add or drop partitions, to work with different portions of a huge data set. With data partitioned by date values, you might "age out" data that is no longer relevant.
- `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 234 for details.
- In Impala 1.4 and higher, there is a `SHOW PARTITIONS` statement that displays information about each partition in a table. See SHOW Statement on page 135 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.

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.

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 |
+------------------------------------------------------------------+
```

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 higher. 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              |
+----------------------------------------------------------------------+
```

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.

If a view applies to a partitioned table, any partition pruning is determined by the clauses in the original query. Impala does not prune additional columns if the query on the view includes extra `WHERE` clauses referencing the partition key columns.

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.
- 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 1 GB. 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

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 239.) 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.

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 Pig or MapReduce, 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.

Impala can query files encoded with most of the popular file formats and compression codecs used in Hadoop. Impala can create and insert data into tables that use some file formats but not others; for file formats that Impala cannot write to, create the table in Hive, issue the `INVALIDATE METADATA` statement in `impala-shell`, and query the table through Impala. File formats can be structured, in which case they may include metadata and built-in compression. Supported formats include:

Table 2: File Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Parquet	Structured	Snappy, GZIP; currently Snappy by default	Yes.	Yes: `CREATE TABLE`, `INSERT`, `LOAD DATA`, and query.
Text	Unstructured	LZO	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: `CREATE TABLE`, `INSERT`, `LOAD DATA`, and query. If LZO compression is used, you must create the table and load data in Hive.
Avro	Structured	Snappy, GZIP, deflate, BZIP2	Yes, in Impala 1.4.0 and higher. Before that, create the table using Hive.	No. Load data through `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive.
RCFile	Structured	Snappy, GZIP, deflate, BZIP2	Yes.	No. Load data through `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive.
SequenceFile	Structured	Snappy, GZIP, deflate, BZIP2	Yes.	No. Load data through `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive.

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. Not supported for text files.

- GZIP. Recommended when achieving the highest level of compression (and therefore greatest disk-space savings) is desired. Not supported for text files.
- Deflate. Not supported for text files.
- BZIP2. Not supported for text files.
- LZO, for Text files only. Impala can query LZO-compressed Text tables, but currently cannot create them or insert data into them; perform these operations in Hive.

Choosing the File Format for a Table

Different file formats and compression codecs work better for different data sets. While Impala typically provides performance gains regardless of file format, choosing the proper format for your data can yield further 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 where practical. 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 copying the data to the new table using the `INSERT` statement. Depending on the file format, you might run the `INSERT` statement in `impala-shell` or in Hive.
- 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. When performance and resource usage are the primary considerations, use one of the other file formats and consider using 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 copy the data into a table using a different, more compact file format.
- If your architecture involves storing data to be queried in memory, do not compress the data. There is no I/O savings since the data does not need to be moved from disk, but there is a CPU cost to decompress the data.

Using Text Data Files with Impala Tables

Cloudera 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 3: Text Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Text	Unstructured	LZO	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: `CREATE TABLE`, `INSERT`, and query. If LZO compression is used, you must create the table and load data in Hive.

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 243 for details.

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.

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. Impala ignores any hidden files, that is, files whose names start with a dot. Otherwise, the file names are not significant.

Filenames for data produced through Impala `INSERT` statements are given unique names to avoid filename 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. 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")`.)

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 64.

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

Cloudera 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 but not GZip-compressed text. LZO-compressed files are "splittable", meaning that different portions of a file can be uncompressed and processed independently by different nodes. GZip-compressed files are not splittable, making them unsuitable for Impala-style distributed queries.

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 using Cloudera repositories:**

   On systems managed by Cloudera Manager, using parcels:

   See the setup instructions for the LZO parcel in the Cloudera Manager documentation for Cloudera Manager 4 Cloudera Manager 5.

   On systems managed by Cloudera Manager, using packages, or not managed by Cloudera Manager:

Download and install the appropriate file to each machine on which you intend to use LZO with Impala. These files all come from the Cloudera GPL extras download site. Install the:

- Red Hat 5 repo file to `/etc/yum.repos.d/`.
- Red Hat 6 repo file to `/etc/yum.repos.d/`.
- SUSE repo file to `/etc/zypp/repos.d/`.
- Ubuntu 10.04 list file to `/etc/apt/sources.list.d/`.
- Ubuntu 12.04 list file to `/etc/apt/sources.list.d/`.
- Debian list file to `/etc/apt/sources.list.d/`.

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.

   Note: The name of the Hadoop LZO package changes between CDH 4 and CDH 5. In CDH 4, the package name is `hadoop-lzo-cdh4`. In CDH 5, the package name is `hadoop-lzo`. Use the appropriate package name depending on the level of CDH in your cluster.

   For RHEL/CentOS systems:

```
$ sudo yum update
$ sudo yum install hadoop-lzo-cdh4 # For clusters running CDH 4.
$ sudo yum install hadoop-lzo      # For clusters running CDH 5 or higher.
$ sudo yum install impala-lzo
```

   For SUSE systems:

```
$ sudo apt-get update
$ sudo zypper install hadoop-lzo-cdh4 # For clusters running CDH 4.
$ sudo zypper install hadoop-lzo      # For clusters running CDH 5 or higher.
$ sudo zypper install impala-lzo
```

   For Debian/Ubuntu systems:

```
$ sudo zypper update
$ sudo apt-get install hadoop-lzo-cdh4 # For clusters running CDH 4.
$ sudo apt-get install hadoop-lzo      # For clusters running CDH 5 or higher.
$ sudo apt-get install impala-lzo
```

   Note:

   The level of the `impala-lzo-cdh4` 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-cdh4-0.4.15-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 the Parquet File Format with Impala Tables

Impala helps 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 data file contains the values for a set of rows (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.

Table 4: Parquet Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Parquet	Structured	Snappy, GZIP; currently Snappy by default	Yes.	Yes: `CREATE TABLE`, `INSERT`, and query.

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'
  LOCATION '/user/etl/destination'
  STORED AS PARQUET;
```

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;
```

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 90 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 Snappy and GZip Compression for Parquet Data Files on page 249 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:

- These hints are available in Impala 1.2.2 and higher.
- You would only use these hints if an `INSERT` into a partitioned Parquet table was failing due to capacity limits, or if such an `INSERT` was succeeding but with less-than-optimal performance.
- To use these hints, put the hint keyword `[SHUFFLE]` or `[NOSHUFFLE]` (including the square brackets) after the `PARTITION` clause, immediately before the `SELECT` keyword.
- `[SHUFFLE]` selects an execution plan that minimizes the number of files being written simultaneously to HDFS, and the number of 1 GB memory buffers holding data for individual partitions. Thus it reduces overall resource usage for the `INSERT` operation, allowing some `INSERT` 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 constructed on the same node.
- `[NOSHUFFLE]` selects an execution plan that might be faster overall, but might also produce a larger number of small data files or exceed capacity limits, causing the `INSERT` operation to fail. Use `[SHUFFLE]` in cases where an `INSERT` statement fails or runs inefficiently due to all nodes attempting to construct data for all partitions.
- Impala automatically uses the `[SHUFFLE]` method if any partition key column in the source table, mentioned in the `INSERT ... SELECT` query, does not have column statistics. In this case, only the `[NOSHUFFLE]` hint would have any effect.
- If column statistics are available for all partition key columns in the source table mentioned in the `INSERT ... SELECT` 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 `INSERT` operation. In this case, you might need the `[SHUFFLE]` or the `[NOSHUFFLE]` hint to override the execution plan selected by Impala.

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 1 GB 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 1 GB block size of the Parquet data files by using the `hdfs distcp -pb` command rather than a `-put` or `-cp` operation on the Parquet files. See Example of Copying Parquet Data Files on page 251 for an example of this kind of operation.

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 1 GB 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 1 GB 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 105.

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 1 GB data files ("row groups"), 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 1 GB 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 84 for details.

Note: Currently, a known issue (IMPALA-488) could cause excessive memory usage during a `COMPUTE STATS` operation on a Parquet table. As a workaround, issue the command `SET NUM_SCANNER_THREADS=2` in `impala-shell` before issuing the `COMPUTE STATS` statement. Then issue `UNSET NUM_SCANNER_THREADS` before continuing with queries.

Partitioning for Parquet Tables

As explained in Partitioning on page 233, 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 252, 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 1 GB block size, so when deciding how finely to partition the data, try to find a granularity where each partition contains 1 GB 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.

Note: Currently, a known issue (IMPALA-488) could cause excessive memory usage during a `COMPUTE STATS` operation on a Parquet table. As a workaround, issue the command `SET NUM_SCANNER_THREADS=2` in `impala-shell` before issuing the `COMPUTE STATS` statement. Then issue `UNSET NUM_SCANNER_THREADS` before continuing with queries.

Snappy and GZip Compression for Parquet Data Files

When Impala writes Parquet data files using the `INSERT` statement, the underlying compression is controlled by the `PARQUET_COMPRESSION_CODEC` query option. The allowed values for this query option are snappy (the default), gzip, 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 PARQUET_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 PARQUET_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 PARQUET_COMPRESSION_CODEC query option to gzip before inserting the data:

```
[localhost:21000] > create table parquet_gzip like raw_text_data;
[localhost:21000] > set PARQUET_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 PARQUET_COMPRESSION_CODEC query option to none before inserting the data:

```
[localhost:21000] > create table parquet_none like raw_text_data;
[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 sized at about 1 GB, each directory will have a different number of data files and the row groups will be arranged differently.

At the same time, the less agressive 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 `PARQUET_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 `hdfs distcp -pb` to ensure that the special 1 GB block size of the Parquet data files is preserved.

```
$ hdfs distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_snappy \
  /user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
$ hdfs distcp -pb /user/hive/warehouse/parquet_compression.db/parquet_gzip  \
  /user/hive/warehouse/parquet_compression.db/parquet_everything
...MapReduce output...
$ hdfs 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
```

Exchanging Parquet Data Files with Other Hadoop Components

Starting in CDH 4.5, you can read and write Parquet data files from Hive, Pig, and MapReduce. See the CDH 4 Installation Guide for details.

Previously, it was not possible to create Parquet data through Impala and reuse that table within Hive. Now that Parquet support is available for Hive in CDH 4.5, 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. If any column of a table uses such an unsupported data type, Impala cannot access that table.

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 1 GB. (The `hadoop distcp` operation typically leaves some directories behind, with names matching `_distcp_logs_*`, that you can delete from the destination directory afterward.) See the Hadoop DistCP Guide for details.

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 an HDFS block size and a maximum data file size of 1 GB, to ensure that I/O and network transfer requests apply to large batches of data.

Within that gigabyte of space, 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:

The Parquet data files have an HDFS block size of 1 GB, the same as the maximum Parquet 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 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 251 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 where the values for that column are stored consecutively. 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 1 GB 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 1 GB, or a multiple of 1 GB.

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 1 GB 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 (1 GB by default). Impala estimates on the conservative side when figuring out how much data to write to each Parquet file. Typically, 1 GB of uncompressed data in memory is reduced down to much less than 1 GB on disk by the compression and encoding techniques in the Parquet file format. Impala reserves 1 GB of memory to buffer the data before writing, but the actual data file might be smaller, in the hundreds of megabytes. The final data file size varies depending on the compressibility of the data. Therefore, it is not an indication of a problem if 1 GB of text data is turned into 2 Parquet data files, each less than 1 GB.
- 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.

Using the Avro File Format with Impala Tables

Cloudera Impala supports using tables whose data files use the Avro file format. Impala can query Avro tables, and in Impala 1.4.0 and higher can create them, but currently cannot create them or insert data into them. For insert operations, use Hive, then switch back to Impala to run queries.

Table 5: Avro Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
Avro	Structured	Snappy, GZIP, deflate, BZIP2	Yes, in Impala 1.4.0 and higher. Before that, create the table using Hive.	No. Load data through LOAD DATA on data files already in the right format, or use INSERT in Hive.

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.

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 impala_avro_table
                  > (name 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. Use `DECIMAL` in Avro tables only under CDH 5. The infrastructure and components under CDH 4 do not have reliable `DECIMAL` support.
- 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`.

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 49

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

If you already have data files in Avro format, you can also issue `LOAD DATA` in either Impala or Hive.

To copy data from another table, issue any `INSERT` statements through Hive. For information about loading data into Avro tables through Hive, see Avro page on the Hive wiki.

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;
-- Query: select * from avro_table
-- Query finished, fetching results ...
-- +---+------+
-- | 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;
-- Query: refresh avro_table
-- Query finished, fetching results ...

-- Returned 0 row(s) in 0.23s
-- [localhost:21000] > select * from avro_table;
-- Query: select * from avro_table
-- Query finished, fetching results ...
-- +---+------+----+
-- | a | b    | c  |
-- +---+------+----+
-- | 1 | avro | 10 |
-- +---+------+----+
-- Returned 1 row(s) in 0.14s
```

Using the RCFile File Format with Impala Tables

Cloudera Impala supports using RCFile data files.

Table 6: RCFile Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
RCFile	Structured	Snappy, GZIP, deflate, BZIP2	Yes.	No. Load data through `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive.

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 239 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.

- **Important:** See Known Issues in the Current Production Release (Impala 1.4.x) for potential compatibility issues with RCFile tables created in Hive 0.12, due to a change in the default RCFile SerDe for Hive.

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
```

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.

Using the SequenceFile File Format with Impala Tables

Cloudera Impala supports using SequenceFile data files.

Table 7: SequenceFile Format Support in Impala

File Type	Format	Compression Codecs	Impala Can CREATE?	Impala Can INSERT?
SequenceFile	Structured	Snappy, GZIP, deflate, BZIP2	Yes.	No. Load data through `LOAD DATA` on data files already in the right format, or use `INSERT` in Hive.

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 239 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
```

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.

Using Impala to Query HBase Tables

You can use Impala to query HBase tables. This capability allows convenient access to a storage system that is tuned for different kinds of workloads than the default with Impala. The default Impala tables use data files stored on HDFS, which are ideal for bulk loads and queries using full-table scans. In contrast, HBase can do efficient queries for data organized for OLTP-style workloads, with lookups of individual rows or ranges of values.

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 snapshot of the Apache HBase site (including documentation) for the level of HBase that comes with CDH 4 or CDH 5. To install HBase on a CDH cluster, see the installation instructions for CDH 4 or CDH 5

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 272 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` clauses 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 http://hbase.apache.org/book/ch08s04.html#hbase.accesscontrol.configuration.

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, Cloudera recommends setting timeout values as follows 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 272 for examples.

HBase works as a kind of "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` 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 conditional tests (through operators such as =, <, `BETWEEN`, and `IN`) 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 range of rows, not queries that scan the entire table. (If a query has no `WHERE` clause, that is a strong indicator that it is an inefficient query for an HBase table.)
- If you have join queries that do aggregation operations on large fact tables and join the results against small dimension tables, consider using Impala for the fact tables and HBase for the dimension tables. (Because Impala does a full scan on the HBase table in this case, rather than doing single-row HBase lookups based on the join column, only use this technique where the HBase table is small enough that doing a full table scan does not cause a performance bottleneck for the query.)

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 region server. 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:** Because these options are not currently settable through the JDBC or ODBC interfaces, to use them with JDBC or ODBC applications, choose values appropriate for all Impala applications in the cluster, and specify those values through the `impalad` startup options as previously described.

Use Cases for Querying HBase through Impala

The following are popular use cases for using Impala to query HBase tables:

- Keeping large fact tables in Impala, and smaller dimension tables in HBase. The fact tables use Parquet or other binary file format optimized for scan operations. Join queries scan through the large Impala fact tables, and cross-reference the dimension tables using efficient single-row lookups in HBase.
- 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.

  Or the HBase table could be joined with a larger Impala-managed table. For example, analyze the large Impala table representing web traffic for a site and pick out 50 users who view the most pages. Join that result with the wide user table in HBase to look up attributes of those users. The HBase side of the join would result in 50 efficient single-row lookups in HBase, rather than scanning the entire user table.

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.

Examples of Querying HBase Tables from Impala

The following examples use HBase with the following table definition. Note that 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
...
create 'hbasealltypessmall', 'bools', 'ints', 'floats', 'strings'
quit
```

With a String Row Key

Issue the following `CREATE TABLE` statement in the Hive shell. (The Impala `CREATE TABLE` statement currently does not support all the required clauses, 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 an external table so that when dropped by Impala or Hive, the original HBase table is not touched at all. The `STORED BY` clause is the clause not currently supported by Impala that requires using the Hive shell for the `CREATE TABLE`. 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 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> 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,bools:bool_col,ints:tinyint_col,ints:smallint_col,ints:int_col,ints:\
  bigint_col,floats:float_col,floats:double_col,strings:date_string_col,\
  strings:string_col,strings:timestamp_col"
)
TBLPROPERTIES("hbase.table.name" = "hbasealltypessmall");
```

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.)

Without a String Row Key

This example defines the lookup key column as `INT` instead of `STRING`.

Note: Although this table definition works, Cloudera strongly recommends using a string value as the row key for HBase tables, because the key lookups are much faster when the key column is defined as a string.

Again, issue the following `CREATE TABLE` statement through Hive, then switch back to Impala and the `impala-shell` interpreter to issue the queries.

```
$ hive
...
CREATE EXTERNAL TABLE hbasealltypessmall (
  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,
  timestamp_col timestamp)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
  "hbase.columns.mapping" =

":key,bools:bool_col,ints:tinyint_col,ints:smallint_col,ints:int_col,ints:bigint_col,floats\


:float_col,floats:double_col,strings:date_string_col,strings:string_col,strings:timestamp_col"
)
TBLPROPERTIES("hbase.table.name" = "hbasealltypessmall");
```

Example Queries

Once you have established the mapping to an HBase table, you can issue queries.

For example:

```
# if the row key is mapped as a string col, range predicates are applied to the scan
select * from hbasestringids where id = '5';

# predicate on row key doesn't get transformed into scan parameter, because
# it's mapped as an int (but stored in ASCII and ordered lexicographically)
select * from hbasealltypessmall where id < 5;
```

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 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 Cloudera Impala 1.1.1 produces a separate set of audit log files when enabled. See Auditing Impala Operations for details.

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 Services Monitoring and Searching Logs in the Cloudera Manager Monitoring and Diagnostics Guide.

If you are using Impala in an environment not managed by Cloudera Manager, review Impala log files on each node:

- By default, the log files are under the directory `/var/log/impala`. To change log file locations, modify the defaults file described in Starting Impala.
- 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/logs/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/logs/impalad/impala-state-store.log`, with all the same information as the corresponding `.INFO`, `.WARNING`, and `.ERROR` files.

Using Impala Logging

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 277 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.

The main administrative tasks involved with Impala logs are:

Reviewing Impala Logs

By default, the Impala log is stored at `/var/logs/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 How To Use Google Logging Library (glog).

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.

Appendix A - 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.

Component	Service	Port	Access Requirement	Comment
Impala Daemon	Impala Daemon Frontend Port	21000	External	Used to transmit commands and receive results by `impala-shell`, Beeswax, and version 1.2 of the Cloudera ODBC driver.
Impala Daemon	Impala Daemon Frontend Port	21050	External	Used to transmit commands and receive results by applications, such as Business Intelligence tools, using JDBC and the version 2.0 or higher of the Cloudera ODBC driver.
Impala Daemon	Impala Daemon Backend Port	22000	Internal	Internal use only. Impala daemons use to communicate with each other.
Impala Daemon	StateStoreSubscriber Service Port	23000	Internal	Internal use only. Impala daemons listen on this port for updates from the state store.
Impala Daemon	Impala Daemon HTTP Server Port	25000	External	Impala web interface for administrators to monitor and troubleshoot.
Impala StateStore Daemon	StateStore HTTP Server Port	25010	External	StateStore web interface for administrators to monitor and troubleshoot.
Impala Catalog Daemon	Catalog HTTP Server Port	25020	External	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	Internal use only. State store listens on this port for registration/unregistration requests.
Impala Catalog Daemon	StateStore Service Port	26000	Internal	Internal use only. The catalog service uses this port to communicate with the Impala daemons. New in Impala 1.2 and higher.

Component	Service	Port	Access Requirement	Comment
Impala Daemon	Llama Callback Port	28000	Internal	Internal use only. Impala daemons use to communicate with Llama. New in CDH 5.0.0 and higher.
Impala Llama ApplicationMaster	Llama Thrift Admin Port	15002	Internal	Internal use only. New in CDH 5.0.0 and higher.
Impala Llama ApplicationMaster	Llama Thrift Port	15000	Internal	Internal use only. New in CDH 5.0.0 and higher.
Impala Llama ApplicationMaster	Llama HTTP Port	15001	External	Llama service web interface for administrators to monitor and troubleshoot. New in CDH 5.0.0 and higher.

Appendix B - Troubleshooting Impala

Use the following steps to diagnose and debug problems with any aspect of Impala.

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 environments.
  - Review the Language Reference to ensure your query is valid.
  - 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.

The following table lists common problems and potential solutions.

Symptom	Explanation	Recommendation
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 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 116.
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`	Ensure Impala is installed on all DataNodes. Start any `impalad` instances that are not running.

Symptom	Explanation	Recommendation
	instances installed in the cluster. There should also be one `impalad` instance installed on each DataNode	
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.
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.
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 to errors connecting to the statestore service and attempts to re-register.	A large number of databases, tables, partitions, and so on can require metadata synchronization on startup that takes longer than the default timeout for the statestore service.	Increase the statestore timeout value above its default of 10 seconds. For instructions, see Increasing the Statestore Timeout on page 44.

Impala Web User Interface for Debugging

The web user interface is primarily for problem diagnosis and troubleshooting. The items listed and their format is subject to change. To monitor Impala health, particularly across the entire cluster at once, use the Cloudera Manager interface.

Debug Web UI for impalad

To debug and troubleshoot the `impalad` daemon 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, choose a specific node that you are curious about or suspect is having problems.

> **Note:** To get a convenient picture of the health of all Impala nodes in a cluster, use the Cloudera Manager interface, which collects the low-level operational information from all Impala nodes, and presents a unified view of the entire cluster.

Main Page

By default, the main page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/` (secure cluster).

This page lists the version of the `impalad` daemon, plus basic hardware and software information about the corresponding host, such as information about the CPU, memory, disks, and operating system version.

Backends Page

By default, the **backends** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/backends` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/backends` (secure cluster).

This page lists the host and port 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. Links take you to the corresponding debug web pages for any of the other nodes in the cluster.

Catalog Page

By default, the **catalog** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/catalog` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/catalog` (secure cluster).

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, and so on.

Logs Page

By default, the **logs** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/logs` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/logs` (secure cluster).

This page shows the last portion of the `impalad.INFO` log file, the most detailed of the info, warning, and error logs for the `impalad` daemon. You can refer here to see the details of the most recent operations, whether the operations succeeded or encountered errors. This central page can be more convenient than looking around the filesystem for the log files, which could be in different locations on clusters that use Cloudera Manager or not.

Memz Page

By default, the **memz** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/memz` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/memz` (secure cluster).

This page displays summary and detailed information about memory usage by the `impalad` daemon. You can see the memory limit in effect for the node, and how much of that memory Impala is currently using.

Metrics Page

By default, the **metrics** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/metrics` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/metrics` (secure cluster).

This page displays the current set of metrics: counters and flags representing various aspects of `impalad` internal operation. For the meanings of these metrics, see Impala Metrics in the Cloudera Manager documentation.

Queries Page

By default, the **queries** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/queries` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/queries` (secure cluster).

This page lists all currently running queries, plus any 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`.)

On this page, you can see at a glance how many SQL statements are failing (`State` value of `EXCEPTION`), how large the result sets are (`# rows fetched`), and how long each statement took (`Start Time` and `End Time`).

Each query has an associated link that displays the detailed query profile, which you can examine to understand the performance characteristics of that query. See Using the Query Profile for Performance Tuning on page 226 for details.

Sessions Page

By default, the **sessions** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/sessions` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/sessions` (secure cluster).

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

By default, the **threadz** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/threadz` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/threadz` (secure cluster).

This page displays information about the threads used by this instance of `impalad`, and shows which categories they are grouped into. Making use of this information requires substantial knowledge about Impala internals.

Varz Page

By default, the **varz** page of the debug web UI is at `http://`*`impala-server-hostname`*`:25000/varz` (non-secure cluster) or `https://`*`impala-server-hostname`*`:25000/varz` (secure cluster).

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; Impala might not actually make use of all settings.

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.

Appendix C - Impala Reserved Words

The following are the reserved words for the current release of Cloudera Impala. A reserved word is one that cannot be used directly as an identifier; you must quote it with backticks. For example, a statement `CREATE TABLE select (x INT)` fails, while ``CREATE TABLE `select` (x INT)`` succeeds. Impala does not reserve the names of aggregate or scalar built-in functions. (Formerly, Impala did reserve the names of some aggregate functions.)

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.

```
add
aggregate
all
alter
and
api_version
as
asc
avro
between
bigint
binary
boolean
by
cached
case
cast
change
char
class
close_fn
column
columns
comment
compute
create
cross
data
database
databases
date
datetime
decimal
delimited
desc
describe
distinct
div
double
drop
else
end
escaped
exists
explain
external
false
fields
fileformat
finalize_fn
first
float
format
formatted
from
```

```
full
function
functions
group
having
if
in
init_fn
inner
inpath
insert
int
integer
intermediate
interval
into
invalidate
is
join
last
left
like
limit
lines
load
location
merge_fn
metadata
not
null
nulls
offset
on
or
order
outer
overwrite
parquet
parquetfile
partition
partitioned
partitions
prepare_fn
produced
rcfile
real
refresh
regexp
rename
replace
returns
right
rlike
row
schema
schemas
select
semi
sequencefile
serdeproperties
serialize_fn
set
show
smallint
stats
stored
straight_join
string
symbol
table
tables
tblproperties
terminated
```

```
textfile
then
timestamp
tinyint
to
true
uncached
union
update_fn
use
using
values
view
when
where
with
```