cloudera
Ask Bigger Questions

Installing and Using Cloudera Impala

Important Notice

Cloudera, Inc.
1001 Page Mill Road, Building 2
Palo Alto, CA 94304-1008
info@cloudera.com
US: 1-888-789-1488
Intl: 1-650-362-0488
www.cloudera.com

Release Information

Version: 1.1.1
Date: September 5, 2013

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 (Hue Beeswax) 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.

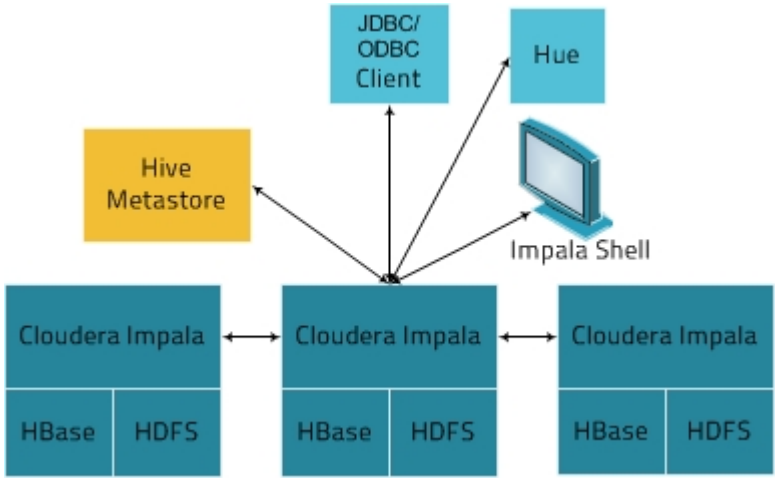
Impala Benefits

Impala provides:

- Familiar SQL interface that data scientists and analysts already know
- Ability to interactively query data on big data in Apache Hadoop
- 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.

- 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 starting with:
  - 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.

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.

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.

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.

Overview of Impala Programming Interfaces

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

- The `impala-shell` command.
- 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 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 gathered by the `ANALYZE TABLE` statement in Hive.

Overview of Impala Metadata and the Metastore

As discussed in How Impala Works with Hive on page 13, 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, any other Impala daemon in the cluster must retrieve the latest metadata, replacing the obsolete cached metadata, before issuing a query against that table. The mechanism to refresh this data is 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 Impala tables. 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 149 for details.

Installing Cloudera Impala

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

Impala is installed separately from other components as an add-on to your environment. Impala is made up of a set of components which can be installed on multiple nodes throughout your cluster.

The Impala package installs three binaries:

- `impalad` - The Impala daemon. Plans and executes queries against HDFS and HBase data. Run one daemon process on each node in the cluster that has a data node.
- `statestored` - Name service that tracks location and status of all `impalad` instances in the cluster. Run one instance of this daemon on a node in your cluster.
- `impala-shell` - Command-line interface for issuing queries to the Impala daemon. You install this on one or more nodes in the cluster, not necessarily data nodes. It can connect remotely to a node running the Impala daemon.

Before doing the installation, ensure that you have all necessary prerequisites. See Cloudera Impala Requirements on page 15 for details.

You can install Impala in one of two ways:

- Using the Cloudera Manager installer, as described in Installing Impala with Cloudera Manager on page 18. This is the recommended technique for doing a reliable and verified Impala installation. Cloudera Manager 4.6 or later can automatically install, configure, manage, and monitor Impala 1.1 and higher.
- Using the manual process described in Installing Impala without Cloudera Manager on page 19. You must do additional verification steps in this case, to check that Impala can interact with other Hadoop components correctly, and that your cluster is configured for efficient Impala execution.

Cloudera Impala Requirements

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

Supported Operating Systems

Supported 64-bit operating systems:

- Red Hat Enterprise Linux (RHEL) 5.7/6.2 or Centos 5.7/6.2.
- SLES 11 with Service Pack 1 or later.
- Ubuntu 10.04/12.04 or Debian 6.03.

Supported Hadoop Distributions

Impala 1.1 is supported under CDH 4.1 or later. For best performance, Cloudera strongly recommends using Impala with CDH 4.2 or higher, ideally the latest release CDH 4.3.

- CDH 4.1 or later for Impala 1.1.
- CDH 4.1 or later for Impala 1.0.

- CDH 4.1 or later for Impala 0.7.
- CDH 4.2 or later for Impala 0.6.
- CDH 4.1 for Impala 0.5 and earlier. This combination is only supported on RHEL/CentOS.

Hive Metastore and Related Configuration

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

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

  - **Note:**

    Installing and configuring a Hive metastore is an Impala requirement. Impala does not work without a Hive metastore. For the process of installing and configuring a Hive metastore, see Installing Cloudera Impala on page 15.

    When installing Impala under CDH 4.1, always configure a **Hive metastore service** rather than connecting directly to the metastore database. The metastore service is required to interoperate between the different levels of Hive used by CDH 4.1 and Impala. The metastore service is set up for you by default if you install through Cloudera Manager 4.5 or later.

    A summary of the Hive installation process is as follows:

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

- **Optional:** Hive. Although only the Hive metastore is required for Impala to function, you might install Hive on some client machines to create and load data into tables that use certain file formats. See How Impala Works with Hadoop File Formats on page 131 for details. Hive does not need to be installed on the same data nodes as Impala; it just needs access to the same metastore database.

Java Dependencies

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

- All Java dependencies are packaged in the `impala-dependencies.jar` file, which is located at `/usr/lib/impala/lib/`. These map to everything that is built under `fe/target/dependency`.

Packages and Repositories

Packages or properly configured repositories. You can install Impala manually using packages, through the Cloudera Impala public repositories, or from your own custom repository. To install using the Cloudera Impala repository, download and install the file to each machine on which you intend to install Impala or Impala Shell. Install the appropriate package or list file as follows:

- Red Hat 5 repo file (http://archive.cloudera.com/impala/redhat/5/x86_64/impala/cloudera-impala.repo) in `/etc/yum.repos.d/`.

- Red Hat 6 repo file (http://archive.cloudera.com/impala/redhat/6/x86_64/impala/cloudera-impala.repo) in `/etc/yum.repos.d/`.
- SUSE repo file (http://archive.cloudera.com/impala/sles/11/x86_64/impala/cloudera-impala.repo) in `/etc/zypp/repos.d/`.
- Ubuntu 10.04 list file (http://archive.cloudera.com/impala/ubuntu/lucid/amd64/impala/cloudera.list) in `/etc/apt/sources.list.d/`.
- Ubuntu 12.04 list file (http://archive.cloudera.com/impala/ubuntu/precise/amd64/impala/cloudera.list) in `/etc/apt/sources.list.d/`.
- Debian list file (http://archive.cloudera.com/impala/debian/squeeze/amd64/impala/cloudera.list) in `/etc/apt/sources.list.d/`.

For example, on a Red Hat 6 system, you might run a sequence of commands like the following:

```
$ cd /etc/yum.repos.d
$ sudo wget
http://archive.cloudera.com/impala/redhat/6/x86_64/impala/cloudera-impala.repo
$ ls
CentOS-Base.repo  CentOS-Debuginfo.repo  CentOS-Media.repo  Cloudera-cdh.repo
cloudera-impala.repo
```

- **Note:**

  You can retrieve files from the `archive.cloudera.com` site through `wget` or `curl`, but not through `rsync`.

Optionally, you can install and manage Impala through the Cloudera Manager product. To install Impala version 1.1.1 using Cloudera Manager, upgrade Cloudera Manager to 4.7 or higher, for full support of all features, including the auditing feature introduced in Impala 1.1.1. When you install through Cloudera Manager, you can install either with the OS-specific "package" mechanism, or the Cloudera Manager "parcels" mechanism, which simplifies upgrades and deployment across an entire cluster.

Networking Configuration Requirements

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

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

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

Hardware Requirements

During join operations all data from both data sets is loaded into memory. Data sets can be very large, so ensure your hardware has sufficient memory to accommodate the joins you anticipate completing.

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

- CPU - Impala uses the SSE4.2 instruction set, which is included in newer processors. Impala can use older processors, but for best performance use:

  - Intel - Nehalem (released 2008) or later processors.
  - AMD - Bulldozer (released 2011) or later processors.
- Memory - 32GB or more. Impala cannot run queries that have a working set greater than the total available RAM. Note that the working set is not the size of the input.
- Storage - DataNodes with 10 or more disks each. I/O speeds are often the limiting factor for disk performance with Impala. Ensure you have sufficient disk space to store the data Impala will be querying.

User Account Requirements

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

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

The Impala security features provide authorization and authentication based on the Linux OS user who connects to the Impala server, and the associated groups for that user. See Impala Security on page 35 for details.

Installing Impala with Cloudera Manager

Before installing Impala through the Cloudera Manager interface, make sure all applicable nodes have the appropriate hardware configuration and levels of operating system and CDH. See Cloudera Impala Requirements on page 15 for details.

> **Note:**
>
> To install Impala version 1.1.1, upgrade Cloudera Manager to 4.7 or higher, for full support of all features, including the auditing feature introduced in Impala 1.1.1.

For information on installing Impala in a Cloudera Manager-managed environment, see Installing Impala with Cloudera Manager in the Cloudera Manager Installation Guide.

Managing your Impala installation through Cloudera Manager has a number of advantages. For example, when you make configuration changes to CDH components using Cloudera Manager, it automatically applies changes to the copies of configuration files, such as `hive-site.xml`, that Impala keeps under `/etc/impala/conf`. It also sets up the Hive metastore service that is required for Impala running under CDH 4.1.

In some cases, depending on the level of Impala, CDH, and Cloudera Manager, you might need to add particular component configuration details in one of the "Safety Valve" fields on the Impala configuration pages within Cloudera Manager.

> **Note:**
>
> Due to a change to the implementation of logging in Impala 1.1.1, currently you should change the default setting for the `logbuflevel` property for the Impala service after installing through Cloudera Manager. In Cloudera Manager, go to the log settings page for the Impala service: **Services** > **Impala** > **Configuration** > **View and Edit** > **Impala Daemon (Default)** > **Logs** . Change the setting **Impala Daemon Log Buffer Level (logbuflevel)** from -1 to 0. You might change this setting to a value higher than 0, if you prefer to reduce the I/O overhead for logging, at the expense of possibly losing some lower-priority log messages in the event of a crash.

Installing Impala without Cloudera Manager

Before installing Impala manually, make sure all applicable nodes have the appropriate hardware configuration, levels of operating system and CDH, and any other software prerequisites. See Cloudera Impala Requirements on page 15 for details.

You can install Impala across many nodes or on one node:

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

To install Impala on a node:

1. Install CDH4 as described in CDH4 Installation in the CDH4 Installation Guide.
2. Install the Hive metastore somewhere in your cluster, as described in Hive Installation in the CDH4 Installation Guide. As part of this process, you configure the Hive metastore to use an external database as a metastore. Impala uses this same database for its own table metadata. You can choose either a MySQL or PostgreSQL database as the metastore. The process for configuring each type of database is described in the CDH4 Installation Guide.

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

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

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

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

4. Confirm that your package management command is aware of the Impala repository settings, as described in Cloudera Impala Requirements on page 15. (This is a different repository than for CDH.) You might need to download a repo or list file into a system directory underneath `/etc`.
5. Use **one** of the following commands to install the Impala package:

For RHEL/CentOS systems:

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

For SUSE systems:

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

For Debian/Ubuntu systems:

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

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

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

   For RHEL/CentOS systems:

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

   For SUSE systems:

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

   For Debian/Ubuntu systems:

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

8. Complete any required or recommended configuration, as described in Post-Installation Configuration for Impala on page 21. Some of these configuration changes are mandatory. (They are applied automatically when you install using Cloudera Manager.)
9. Once installation and configuration are complete, see Starting Impala on page 51 for how to activate the software on the appropriate nodes in your cluster.

Configuring Impala

This section explains how to configure Impala to accept connections from applications that use popular programming APIs:

- Post-Installation Configuration for Impala on page 21
- Configuring Impala to Work with ODBC on page 24
- Configuring Impala to Work with JDBC on page 29

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

You can also configure these other aspects of Impala:

- Impala Security on page 35
- Modifying Impala Startup Options on page 51

Post-Installation Configuration for Impala

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

In some cases, depending on the level of Impala, CDH, and Cloudera Manager, you might need to add particular component configuration details in one of the "Safety Valve" fields on the Impala configuration pages within Cloudera Manager.

- You must enable short-circuit reads, whether or not Impala was installed through Cloudera Manager.
- If you installed Impala in an environment that is not managed by Cloudera Manager, you can optionally enable block location tracking and native checksumming for optimal performance.
- If you deployed Impala using Cloudera Manager see Testing Impala Performance on page 129 to confirm proper configuration.

Mandatory: Short-Circuit Reads

Enabling short-circuit reads allows Impala to read local data directly from the file system. This removes the need to communicate through the DataNodes, improving performance. This setting also minimizes the number of additional copies of data. Short-circuit reads requires `libhadoop.so` (the Hadoop Native Library) to be accessible to both the server and the client. `libhadoop.so` is not available if you have installed from a tarball. You must install from an `.rpm`, `.deb`, or parcel to use short-circuit local reads.

> **Note:** Cloudera Manager provides a checkbox for enabling short-circuit reads. With this release, you must still configure short-circuit reads manually, even if you have enabled the short-circuit reads checkbox in Cloudera Manager.

Cloudera strongly recommends using Impala with CDH 4.2 or higher, ideally the latest release, CDH 4.3. Impala does support short-circuit reads with CDH 4.1, but for best performance, upgrade to CDH 4.3. The process of configuring short-circuit reads varies according to which version of CDH you are using. Choose the procedure that is appropriate for your environment.

To configure DataNodes for short-circuit reads with CDH 4.2 or later:

1. On all Impala nodes, configure the following properties in `hdfs-site.xml` as shown:

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

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

<property>
    <name>dfs.client.file-block-storage-locations.timeout</name>
    <value>3000</value>
</property>
```

   - **Note:** The text `_PORT` appears just as shown; you do not need to substitute a number.

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

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

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

To configure DataNodes for short-circuit reads with CDH 4.1:

- **Note:** Cloudera strongly recommends using Impala with CDH 4.2 or higher, ideally the latest release, CDH 4.3. Impala does support short-circuit reads with CDH 4.1, but for best performance, upgrade to CDH 4.3.

1. Enable short-circuit reads by adding settings to the Impala `core-site.xml` file.
   - If you installed Impala using Cloudera Manager, short-circuit reads should be properly configured, but you can review the configuration by checking the contents of the `core-site.xml` file, which is installed at `/etc/impala/conf` by default.
   - If you installed using packages, instead of using Cloudera Manager, create the `core-site.xml` file. This can be easily done by copying the `core-site.xml` client configuration file from another machine that is running hadoop services. This file must be copied to the Impala configuration directory. The Impala configuration directory is set by the `IMPALA_CONF_DIR` environment variable and is by default `/etc/impala/conf`. To confirm the Impala configuration directory, check the `IMPALA_CONF_DIR` environment variable value.

     - **Note:** If the Impala configuration directory does not exist, create it and then add the `core-site.xml` file.

   Add the following to the `core-site.xml` file:

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

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

> **Note:** For an installation managed by Cloudera Manager, specify these settings in the Impala dialogs, in the **HDFS Safety Valve** field.

2. For each DataNode, enable access by adding the following to the `hdfs-site.xml` file:

```
<property>
    <name>dfs.client.use.legacy.blockreader.local</name>
    <value>true</value>
</property>

<property>
    <name>dfs.datanode.data.dir.perm</name>
    <value>750</value>
</property>

<property>
    <name>dfs.block.local-path-access.user</name>
    <value>impala</value>
</property>

<property>
     <name>dfs.client.file-block-storage-locations.timeout</name>
     <value>3000</value>
</property>
```

> **Note:** In the preceding example, the `dfs.block.local-path-access.user` is the user running the `impalad` process. By default, that account is `impala`.

3. Use `usermod` to add users requiring local block access to the appropriate HDFS group. For example, if you assigned `impala` to the `dfs.block.local-path-access.user` property, you would add `impala` to the hadoop HDFS group.

```
$ usermod -a -G hadoop impala
```

> **Note:** The default HDFS group is `hadoop`, but it is possible to have an environment configured to use an alternate group. To find the configured HDFS group name using the Cloudera Manager admin console, click **Services** and click **HDFS**. Click the **Configuration** tab. Under **Service-Wide**, click **Advanced** in the left column. The **Shared Hadoop Group Name** property contains the group name.

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

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

Optional: Block Location Tracking

Enabling block location metadata allows Impala to know which disk data blocks are located on, allowing better utilization of the underlying disks.

To enable block location tracking:

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

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

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

Optional: Native Checksumming

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

To enable native checksumming:

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

Configuring Impala to Work with ODBC

Third-party products can be designed to integrate with Impala using ODBC. For the best experience, ensure any third-party product you intend to use is supported. Verifying support includes checking that the versions of Impala, ODBC, the operating system, and the third-party product have all been approved for use together. Before configuring your systems to use ODBC, download a connector.

> **Note:** You may need to sign in and accept license agreements before accessing the pages required for downloading ODBC connectors.

Version 1.x of the Cloudera ODBC Connectors uses the original HiveServer1 protocol, corresponding to Impala port 21000.

The newer versions 2.5 and 2.0, currently certified for some but not all BI applications, use the HiveServer2 protocol, corresponding to Impala port 21050. Although the 2.x drivers include authentication through either Kerberos tickets or username/password credentials, Impala only supports Kerberos for authentication.

Connector	Download Page	Notes
Cloudera ODBC Driver 2.5 for Impala	Cloudera ODBC Driver for Cloudera Impala	This is the generic ODBC driver you would use to integrate Impala with any ODBC-based application not listed here. This connector is available for both Linux and Windows systems.
Cloudera ODBC Connector 2.0 for Microstrategy	Cloudera ODBC Connector 2.0 for MicroStrategy	This connector is available for both Linux and Windows systems.

Connector	Download Page	Notes
Cloudera ODBC Connector 2.0 for Qlikview	Cloudera ODBC Connector 2.0 for Qlikview	This connector is available for both Linux and Windows systems.
Cloudera ODBC Connector 1.2 for Tableau	Cloudera ODBC Connector 1.2 for Tableau	This connector is available for Windows systems only.
Previous ODBC Connectors	Downloads for Previous ODBC Connectors	This connector is available for both Windows and Linux systems. The latest connector versions have standalone installation guides linked from their respective download pages, while installation for this older connector version is documented in the following section, Configuration Instructions for Cloudera ODBC Connector 1.2 on page 25.

- **Important:** If you are using the Cloudera Connector for Tableau, to connect Impala to your Kerberos-secured CDH clusters, contact your Tableau account representative for an updated Tableau Data-connection Customization (TDC) file. The updated TDC file will override the Tableau connection settings to set specific parameters on the connection string that are required for a secure connection.

Configuration Instructions for Cloudera ODBC Connector 1.2

The instructions in this section apply to the older (1.2) version of the Cloudera ODBC driver. Most applications now use the 2.x drivers, which have separate installation instructions, as listed in Configuring Impala to Work with ODBC on page 24. For the ODBC driver 1.2 configuration instructions, refer to the following sections depending on your platform:

- Configuring MicroStrategy to Use the Cloudera ODBC Driver 1.2 on Linux on page 25
- Configuring MicroStrategy and Qlikview to Use the Cloudera ODBC Driver 1.2 on Windows on page 27

Configuring MicroStrategy to Use the Cloudera ODBC Driver 1.2 on Linux

You must use ODBC driver version 1.2 or later. Using earlier versions is not supported.

After downloading a connector, configure your systems to work with it.

- **Note:**

  The unixODBC driver manager, which is often used to test drivers, requires `odbc.ini` be in the home directory. If you are using the unixODBC driver manager, you must prepare your environment by using a command similar to the following example to copy the `odbc.ini` file:

  ```
  $ cp /etc/odbc.ini ~/odbc.ini
  ```

To configure Microstrategy to use ODBC on Linux:

1. Use `tar` to extract and install the ODBC driver from the archive to a location of your choosing. For example, to install the driver to the standard location of `/usr`, use the following commands:

```
tar -zxf ClouderaHiveODBC-v1.20.tar.gz -C /usr
```

   To install different versions, place the files in alternate locations such as `/opt/`.

2. Create a system DSN for Impala by adding values to `odbc.ini`.

```
[ODBC Data Sources]
IMPALA-SERVER=Hive ODBC Driver

[IMPALA-SERVER]
Driver=ODBC_DRIVER_LIBS_PATH/libhiveodbc.so.1
Description=Hive ODBC Driver
Host=IMPALAD_HOST
Port=IMPALAD_PORT
Database=
FRAMED=0
Trace=Yes
TraceFile=/tmp/odbc.log
Type=Beeswax
```

   - **Note:** In the preceding example, replace values as follows: `ODBC_DRIVER_LIBS_PATH`: The full ODBC driver library path. This is often `/usr/lib`. `IMPALAD_HOST`: The fully qualified hostname of any node running the `impalad` process. `IMPALAD_PORT`: The port number for the `impalad` process. The default is 21000.

3. Implement environment variable setting by adding values to the end of `ODBC.sh`.

```
#
# ODBC Driver for Hive
#
HIVE_CONFIG='ODBC_DRIVER_INSTALL_PATH'
if [ "${HIVE_CONFIG}" != '<HIVE_CONFIG>' ]; then
export HIVE_CONFIG

mstr_append_path LD_LIBRARY_PATH "${HIVE_CONFIG:?}"/lib
export LD_LIBRARY_PATH

mstr_append_path PATH "${HIVE_CONFIG:?}"/bin
export PATH
fi
```

   - **Note:** In the preceding example, replace `ODBC_DRIVER_INSTALL_PATH` with the full path to the parent directory of the driver libraries and include files. This is often `/usr`.

4. Add a section for the ODBC driver to `odbcinst.ini`.

```
[ODBC Drivers]
Hive Driver=Installed

[Hive Driver]
Driver=ODBC_DRIVER_LIBS_PATH/libhiveodbc.so.1
Description=Hive Driver
Setup=ODBC_DRIVER_LIBS_PATH/libhiveodbc.so.1
APILevel=2
ConnectFunctions=YYY
DriverODBCVer=1.0
FileUsage=0
SQLLevel=1
```

   - **Note:** In the preceding example, replace `ODBC_DRIVER_LIBS_PATH` with the full path to the ODBC driver libraries. This is often `/usr/lib`.

Configuring MicroStrategy and Qlikview to Use the Cloudera ODBC Driver 1.2 on Windows

After downloading a connector, configure your systems to work with it.

To configure Microstrategy, Qlikview, or other tools except Tableau to use ODBC on Windows:

1. Run the downloaded `.exe` file to invoke the installation wizard. You will need to accept the license agreement, and accept or modify the installation location. The driver currently comes only in a 32-bit version, but can also run on a 64-bit system.

   > **Note:** You can perform a silent (non-interactive) installation by running the install executable as `<Install_exe>/S /D="<Dir_to_install>"`. For example: `ClouderaHiveODBCSetup_v1_20.exe /S /D="C:\Program Files (x86)\Hive ODBC"` will install the driver and the uninstall script in the `C:\Program Files (x86)\Hive ODBC` directory. The license is accepted automatically in this mode.

2. Open the ODBC Data Source Administrator. You must use the 32-bit version.
3. Click **System DSN** tab, then click **Add**.
4. Select **Cloudera ODBC Driver for Apache Hive** and click **Finish**.
5. Enter values for the following fields:
   - **Data Source Name** – Name for this DSN.
   - **Host** – Fully qualified hostname for a node running the impalad process.
   - **Port** – Port number for the impalad process. The default is 21000.
   - **Type** – Beeswax
6. Click **OK**.

Steps for Tableau on Windows

If you would like to use Tableau with the Cloudera Connector for Tableau, use the same Tableau documentation as Hive. Tableau does not use the system DSN configuration described here, but instead requires configuring the connection within Tableau itself.

Known Issues and Limitations of the Cloudera ODBC Connector Version 1.2

Users of this ODBC driver should take the time to familiarize themselves with Impala and the limitations when using it with the Cloudera ODBC Connector.

Impala is not a relational database, and does not support many of the features that are typically found in relational database systems. Missing features include the following:

- No support for transactions.
- No support for cursors or scrollable cursors.
- No support for row level inserts, updates, or deletes.
- No explicit support for canceling a running query.
- No support for prepared statements.

Users should be aware of the SQL functionality supported by Impala. Users are encouraged to read the Language Reference and check with the application developer to confirm it is supported on Impala before attempting to use this driver.

These limitations affect the driver in two ways:

- Many types of SQL statements can not be executed through the driver because the underlying system does not support them.
- Many functions in the ODBC API can not be supported on top of Hive because Hive does not provide the required infrastructure.

The following ODBC functions are not supported by the driver:

- SQLBindParameter
- SQLBrowseConnect
- SQLBulkOperations
- SQLCancel
- SQLCloseCursor
- SQLColumnPrivileges
- SQLCopyDesc
- SQLDataSources
- SQLDrivers
- SQLEndTrans
- SQLForeignKeys
- SQLGetCursorName
- SQLGetDescField
- SQLGetDescRec
- SQLGetTypeInfo
- SQLMoreResults
- SQLNativeSQL
- SQLParamOptions
- SQLPrepare
- SQLPrimaryKeys
- SQLProcedureColumns
- SQLProcedures
- SQLSetCursorName
- SQLSetDescField
- SQLSetDescRec
- SQLSetPos
- SQLSetScrollOptions
- SQLSpecialColumns
- SQLStatistics
- SQLTablePrivileges
- SQLTransact

The driver provides support for the following ODBC functions, though bugs may be present:

- SQLAllocConnect
- SQLAllocEnv
- SQLAllocHandle
- SQLAllocStmt
- SQLBindCol
- SQLColAttribute
- SQLColAttributes
- SQLColumns
- SQLConnect
- SQLDescribeCol
- SQLDescribeParam
- SQLDisconnect
- SQLDriverConnect
- SQLError
- SQLExecDirect
- SQLExecute

- SQLFetch
- SQLFreeConnect
- SQLFreeEnv
- SQLFreeHandle
- SQLFreeStmt
- SQLGetConnectAttr
- SQLGetConnectOption
- SQLGetConnectionOption
- SQLGetData
- SQLGetDiagField
- SQLGetDiagRec
- SQLGetEnvAttr
- SQLGetFunctions
- SQLGetInfo
- SQLGetStmtAttr
- SQLGetStmtOption
- SQLNumParams
- SQLNumResultCols
- SQLParamData
- SQLPutData
- SQLRowCount (Always returns -1)
- SQLSetConnectAttr
- SQLSetEnvAttr
- SQLSetParam
- SQLSetStmtAttr
- SQLSetStmtOption
- SQLTables

Configuring Impala to Work with JDBC

Impala supports JDBC integration. The JDBC driver allows you to access Impala from a Java program that you write, or a Business Intelligence or similar tool that uses JDBC to communicate with various database products. Setting up a JDBC connection to Impala involves the following steps:

- Specifying an available communication port. See Configuring the JDBC Port on page 29.
- Installing the JDBC driver on every system that runs the JDBC-enabled application. See Enabling Impala JDBC Support on Client Systems on page 30.
- Specifying a connection string for the JDBC application to access one of the servers running the `impalad` daemon, with the appropriate security settings. See Establishing JDBC Connections on page 30.

Configuring the JDBC Port

The default JDBC 2.0 port is 21050; Impala server accepts JDBC connections through this same port 21050 by default. Make sure this port is available for communication with other hosts on your network, for example, that it is not blocked by firewall software. If your JDBC client software connects to a different port, specify that alternative port number with the `--hs2_port` option when starting `impalad`. See Starting Impala on page 51 for details.

Enabling Impala JDBC Support on Client Systems

The Impala JDBC integration is made possible by a client-side JDBC driver, which is contained in JAR files within a zip file. Download this zip file to each client machine that will use JDBC with Impala.

To enable JDBC support for Impala on the system where you run the JDBC application:

1. Download the Impala JDBC zip file to the client machine that you will use to connect to Impala servers.
2. Extract the contents of the zip file to a location of your choosing. For example:
   - On Linux, you might extract this to a location such as `/opt/jars/`.
   - On Windows, you might extract this to a subdirectory of `C:\Program Files`.
3. To successfully load the Impala JDBC driver, client programs must be able to locate the associated JAR files. This often means setting the `CLASSPATH` for the client process to include the JARs. Consult the documentation for your JDBC client for more details on how to install new JDBC drivers, but some examples of how to set `CLASSPATH` variables include:
   - On Linux, if you extracted the JARs to `/opt/jars/`, you might issue the following command to prepend the JAR files path to an existing classpath:

     ```
     export CLASSPATH=/opt/jars/*.jar:$CLASSPATH
     ```

   - On Windows, use the **System Properties** control panel item to modify the **Environment Variables** for your system. Modify the environment variables to include the path to which you extracted the files.

     > **Note:** If the existing `CLASSPATH` on your client machine refers to some older version of the Hive JARs, ensure that the new JARs are the first ones listed. Either put the new JAR files earlier in the listings, or delete the other references to Hive JAR files.

Establishing JDBC Connections

The JDBC driver class is `org.apache.hive.jdbc.HiveDriver`. Once you have configured Impala to work with JDBC, you can establish connections between the two. To do so for a cluster that does not use Kerberos authentication, use a connection string of the form `jdbc:hive2://host:port/;auth=noSasl`. For example, you might use:

```
jdbc:hive2://myhost.example.com:21050/;auth=noSasl
```

To connect to an instance of Impala that requires Kerberos authentication, use a connection string of the form `jdbc:hive2://host:port/;principal=principal_name`. The principal must be the same user principal you used when starting Impala. For example, you might use:

```
jdbc:hive2://myhost.example.com:21050/;principal=impala/myhost.example.com@H2.EXAMPLE.COM
```

Upgrading Impala

Upgrading Cloudera Impala involves stopping Impala services, using your operating system's package management tool to upgrade Impala to the latest version, and then restarting Impala services.

- **Note:**
  - If you upgrade Impala to version 1.1, upgrade Cloudera Manager to 4.6 or higher.
  - You can use either CDH 4.2 or 4.1. If you upgrade from an early Impala beta release to 1.1 or higher, and you are running CDH 4.1, make sure to configure the Hive metastore service rather than connecting directly to the Hive metastore database.
  - Every time you upgrade to a new major or minor Impala release, see Incompatible Changes in the *Release Notes* for any changes needed in your source code, startup scripts, and so on.
  - Also check Known Issues and Workarounds in Impala in the Release Notes for any issues or limitations that require workarounds.
  - For Impala 1.1, the primary differences are:
    - The `REFRESH` statement now requires a table name argument. Thus, in any SQL scripts that call `REFRESH` with no table name, either change to one or more `REFRESH` statements with specific table names, or use the `INVALIDATE METADATA` statement which is now the way to reload all metadata for all tables. See REFRESH Statement on page 96 and INVALIDATE METADATA Statement on page 88 for details; `REFRESH` with a table name argument is the preferred technique where practical, in the common use case where you add new data files to an existing table, while `INVALIDATE METADATA` is required after creating new tables through Hive.
    - 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 10, reusing existing Impala Parquet data files in Hive requires updating the table metadata. See Exchanging Parquet Data Files with Other Hadoop Components on page 141 for the commands to use.
  - Due to a change to the implementation of logging in Impala 1.1.1, currently you should change the default setting for the `logbuflevel` property for the Impala service after installing through Cloudera Manager. In Cloudera Manager, go to the log settings page for the Impala service: **Services** > **Impala** > **Configuration** > **View and Edit** > **Impala Daemon (Default)** > **Logs** . Change the setting **Impala Daemon Log Buffer Level (logbuflevel)** from -1 to 0. You might change this setting to a value higher than 0, if you prefer to reduce the I/O overhead for logging, at the expense of possibly losing some lower-priority log messages in the event of a crash.

To upgrade Impala in a Cloudera Managed environment, using parcels:

1. If you originally installed using packages and now are switching to parcels, remove all the Impala-related packages first.
2. Connect to the Cloudera Manager Admin Console.
3. Go to the **Hosts < Parcels** tab. You should see a parcel with a newer version of Impala that you can upgrade to.
4. Click **Download**, then **Distribute** . (The button changes as each step completes.)
5. Click **Activate**.
6. When prompted, click **Restart** to restart the Impala service.

To upgrade Impala in a Cloudera Managed environment, using packages:

1. Connect to the Cloudera Manager Admin Console.
2. In the **Services** tab, click the **Impala** service.

3. Click **Actions** and click **Stop**.
4. Use **one** of the following sets of commands to update Impala on each Impala node in your cluster:

   For RHEL/CentOS systems:

   ```
   $ sudo yum update impala-server
   $ sudo yum update hadoop-lzo-cdh4 # Optional; if this package is already installed
   ```

   For SUSE systems:

   ```
   $ sudo zypper update impala-server
   $ sudo zypper update hadoop-lzo-cdh4 # Optional; if this package is already
   installed
   ```

   For Debian/Ubuntu systems:

   ```
   $ sudo apt-get install impala-server
   $ sudo apt-get install hadoop-lzo-cdh4 # Optional; if this package is already
   installed
   ```

5. Use **one** of the following sets of commands to update Impala shell on each node on which it is installed:

   For RHEL/CentOS systems:

   ```
   $ sudo yum update impala-shell
   ```

   For SUSE systems:

   ```
   $ sudo zypper update impala-shell
   ```

   For Debian/Ubuntu systems:

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

6. Connect to the Cloudera Manager Admin Console.
7. In the **Services** tab, click the Impala service.
8. Click **Actions** and click **Start**.

To upgrade Impala without Cloudera Manager:

1. Stop Impala services.

   a. Stop `impalad` on each Impala node in your cluster:

   ```
   $ sudo service impala-server stop
   ```

   b. Stop any instances of the state store in your cluster:

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

2. Check if there are new recommended or required configuration settings to put into place in the configuration files, typically under `/etc/impala/conf`. See Post-Installation Configuration for Impala on page 21 for settings related to performance and scalability.
3. Use **one** of the following sets of commands to update Impala on each Impala node in your cluster:

   For RHEL/CentOS systems:

   ```
   $ sudo yum update impala-server
   ```

For SUSE systems:

```
$ sudo zypper update impala-server
```

For Debian/Ubuntu systems:

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

4. Use **one** of the following sets of commands to update Impala shell on each node on which it is installed:

   For RHEL/CentOS systems:

   ```
   $ sudo yum update impala-shell
   ```

   For SUSE systems:

   ```
   $ sudo zypper update impala-shell
   ```

   For Debian/Ubuntu systems:

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

5. Restart Impala services:

   a. Restart the Impala state store service on the desired nodes in your cluster. Expect to see a process named `statestored` if the service started successfully.

   ```
   $ sudo service impala-state-store start
   $ ps ax | grep [s]tatestored
    6819 ?        Sl     0:07 /usr/lib/impala/sbin/statestored
   -log_dir=/var/log/impala -state_store_port=24000
   ```

   Restart the state store service *before* the Impala server service to avoid "Not connected" errors when you run `impala-shell`.

   b. Restart the Impala daemon service on each node in your cluster. Expect to see a process named `impalad` if the service started successfully.

   ```
   $ sudo service impala-server start
   $ ps ax | grep [i]mpalad
    7936 ?        Sl     0:12 /usr/lib/impala/sbin/impalad -log_dir=/var/log/impala
    -state_store_port=24000 -use_statestore
   -state_store_host=127.0.0.1 -be_port=22000
   ```

> **Note:**
>
> If the services did not start successfully (even though the `sudo service` command might display `[OK]`), check for errors in the Impala log file, typically in `/var/log/impala`.

Impala Security

Impala 1.1 adds a fine-grained authorization framework for Hadoop, by integrating the Sentry open source project. Together with the existing Kerberos authentication framework, Impala 1.1 takes Hadoop security to a new level needed for the requirements of highly regulated industries such as healthcare, financial services, and government. Impala 1.1.1 fills in the security feature set even more by adding an auditing capability; Impala generates the audit data, the Cloudera Navigator product consolidates the audit data from all nodes in the cluster, and Cloudera Manager lets you filter, visualize, and produce reports.

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

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

The security features are divided into these broad categories:

authorization

Which users are allowed to access which resources, and what operations are they allowed to perform? Impala uses the OS user ID of the user who runs `impala-shell` or other client program, and associates various privileges with each user. Impala relies on the open source Sentry project for authorization.

authentication

How does Impala verify the identity of the user to confirm that they really are allowed to exercise the privileges assigned to that user? Impala relies on the Kerberos subsystem for authentication.

auditing

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

The following sections lead you through the various security-related features of Impala:

Security Guidelines for Impala

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

- Secure the `root` account. The `root` user can tamper with the `impalad` daemon, read and write the data files in HDFS, log into other user accounts, and access other system services that are beyond the control of Impala.
- Restrict membership in the `sudoers` list (in the `/etc/sudoers` file). The users who can run the `sudo` command can do many of the same things as the `root` user.
- Ensure the Hadoop ownership and permissions for Impala data files are restricted.
- Ensure the Hadoop ownership and permissions for Impala log files are restricted.
- Ensure that the Impala web UI (available by default on port 25000 on each Impala node) is password-protected. See Securing the Impala Web User Interface on page 37 for details.
- Create a policy file that specifies which Impala privileges are available to users in particular Hadoop groups (which by default map to Linux OS groups). Create the associated Linux groups using the `groupadd` command if necessary.
- The Impala authorization feature makes use of the HDFS file ownership and permissions mechanism; for background information, see the CDH HDFS Permissions Guide. Set up users and assign them to groups at the OS level, corresponding to the different categories of users with different access levels for various databases, tables, and HDFS locations (URIs). Create the associated Linux users using the `useradd` command if necessary, and add them to the appropriate groups with the `usermod` command.
- Design your databases, tables, and views with database and table structure to allow policy rules to specify simple, consistent rules. For example, if all tables related to an application are inside a single database, you can assign privileges for that database and use the `*` wildcard for the table name. If you are creating views with different privileges than the underlying base tables, you might put the views in a separate database so that you can use the `*` wildcard for the database containing the base tables, while specifying the precise names of the individual views. (For specifying table or database names, you either specify the exact name or `*` to mean all the databases on a server, or all the tables and views in a database.)
- Enable authorization by running the `impalad` daemons with the `-server_name` and `-authorization_policy_file` options on all nodes. (The authorization feature does not apply to the `statestored` daemon, which has no access to schema objects or data files.)
- Set up authentication using Kerberos, to make sure users really are who they say they are.

Securing Impala Data and Log Files

One aspect of security is to protect files from unauthorized access at the filesystem level. For example, if you store sensitive data in an Impala table, you specify permissions on the associated files and directories in HDFS to restrict read and write permissions to the appropriate users and groups. If you issue queries containing sensitive values in the `WHERE` clause, such as financial account numbers, those values are stored in Impala log files in the Linux filesystem and you must secure those files also.

For the locations of Impala log files, see Using Impala Logging on page 153.

Installation Considerations for Impala Security

Impala 1.1 comes set up with all the software and settings needed to enable security when you run the `impalad` daemon with the new security-related options (`-server_name` and `-authorization_policy_file`). You do

not need to change any environment variables or install any additional JAR files. In a cluster managed by Cloudera Manager, you do not need to change any settings in Cloudera Manager.

Securing the Hive Metastore Database

It is important to secure the Hive metastore, so that users cannot access the names or other information about databases and tables the through the Hive client or by querying the metastore database. Do this by turning on Hive metastore security, using the instructions in the CDH4 Security Guide for securing different Hive components:

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

Securing the Impala Web User Interface

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

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

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

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

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

> **Note:**
>
> If the private key is encrypted using a passphrase, Impala will ask for that passphrase on startup, which is not useful for a large cluster. In that case, remove the passphrase and make the `.pem` file readable only by Impala and administrators.
>
> When you turn on SSL for the Impala web UI, the associated URLs change from `http://` prefixes to `https://`. Adjust any bookmarks or application code that refers to those URLs.

Using Authorization with Impala

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

The object hierarchy covers Server, URI, Database, and Table. Currently, you cannot assign privileges at the partition or column level.

A restricted set of privileges determines what you can do with each object:

SELECT privilege

Lets you read data from a table, for example with the `SELECT` statement, the `INSERT...SELECT` syntax, or `CREATE TABLE...LIKE`. Also required to issue the `DESCRIBE` statement or the `EXPLAIN` statement for a query against a particular table. Only objects for which a user has this privilege are shown in the output for `SHOW DATABASES` and `SHOW TABLES` statements. The `REFRESH` statement and `INVALIDATE METADATA` statements only access metadata for tables for which the user has this privilege.

INSERT privilege

Lets you write data to a table. Applies to the `INSERT` and `LOAD DATA` statements.

ALL privilege

Lets you create or modify the object. Required to run DDL statements such as `CREATE TABLE`, `ALTER TABLE`, or `DROP TABLE` for a table, `CREATE DATABASE` or `DROP DATABASE` for a database, or `CREATE VIEW`, `ALTER VIEW`, or `DROP VIEW` for a view. Also required for the URI of the "location" parameter for the `CREATE EXTERNAL TABLE` and `LOAD DATA` statements.

Privileges can be specified for a table before that table actually exists. If you do not have sufficient privilege to perform an operation, the error message does not disclose if the object exists or not.

Privileges are encoded in a policy file, stored in HDFS. (Currently, there is no `GRANT` or `REVOKE` statement; you cannot adjust privileges through SQL statements.) The location is listed in the `auth-site.xml` configuration file. To minimize overhead, the security information from this file is cached by each `impalad` daemon and refreshed automatically, with a default interval of 5 minutes. After making a substantial change to security policies, restart all Impala daemons to pick up the changes immediately.

See the following sections for details about using the Impala authorization features:

- Setting Up the Policy File for Impala Security on page 38
- Secure Startup for the impalad Daemon on page 39
- Examples of Policy File Rules for Security Scenarios on page 40
- Setting Up Schema Objects for a Secure Impala Deployment on page 44
- Privilege Model and Object Hierarchy on page 44
- Using Multiple Policy Files for Different Databases on page 46

Setting Up the Policy File for Impala Security

The policy file is a file that you put in a designated location in HDFS, and is read during the startup of the `impalad` daemon when you specify the `-server_name` and `-authorization_policy_file` startup options. It controls which objects (databases, tables, and HDFS directory paths) can be accessed by the user who connects to `impalad`, and what operations that user can perform on the objects.

The policy file uses the familiar `.ini` format, divided into the major sections `[groups]` and `[roles]`. There is also an optional `[databases]` section, which allows you to specify a specific policy file for a particular database, as explained in Using Multiple Policy Files for Different Databases on page 46. Another optional section, `[users]`, allows you to override the OS-level mapping of users to groups; that is an advanced technique primarily for testing and debugging, and is beyond the scope of this document.

In the `[groups]` section, you define various categories of users and select which roles are associated with each category. The group and user names correspond to Linux groups and users on the server where the `impalad` daemon runs.

The group and user names in the `[groups]` section correspond to Linux groups and users on the server where the `impalad` daemon runs. When you access Impala through the `impalad` interpreter, for purposes of authorization, the user is the logged-in Linux user and the groups are the Linux groups that user is a member of. When you access Impala through the ODBC or JDBC interfaces, the user and password specified through the

connection string are used as login credentials for the Linux server, and authorization is based on that user name and the associated Linux group membership.

In the `[roles]` section, you a set of roles. For each role, you specify precisely the set of privileges is available. That is, which objects users with that role can access, and what operations they can perform on those objects. This is the lowest-level category of security information; the other sections in the policy file map the privileges to higher-level divisions of groups and users. In the `[groups]` section, you specify which roles are associated with which groups. The group and user names correspond to Linux groups and users on the server where the `impalad` daemon runs. The privileges are specified using patterns like:

```
server=server_name->db=database_name->table=table_name->action=SELECT
server=server_name->db=database_name->table=table_name->action=CREATE
server=server_name->db=database_name->table=table_name->action=ALL
```

For the *server_name* value, substitute the same symbolic name you specify with the `impalad -server_name` option. You can use `*` wildcard characters at each level of the privilege specification to allow access to all such objects. For example:

```
server=impala-host.example.com->db=default->table=t1->action=SELECT
server=impala-host.example.com->db=*->table=*->action=CREATE
server=impala-host.example.com->db=*->table=audit_log->action=SELECT
server=impala-host.example.com->db=default->table=t1->action=*
```

When authorization is enabled, Impala uses the policy file as a *whitelist*, representing every privilege available to any user on any object. That is, only operations specified for the appropriate combination of object, role, group, and user are allowed; all other operations are not allowed. If a group or role is defined multiple times in the policy file, the last definition takes precedence.

To understand the notion of whitelisting, set up a minimal policy file that does not provide any privileges for any object. When you connect to an Impala node where this policy file is in effect, you get no results for `SHOW DATABASES`, and an error when you issue any `SHOW TABLES`, `USE database_name`, `DESCRIBE table_name`, `SELECT`, and or other statements that expect to access databases or tables, even if the corresponding databases and tables exist.

The contents of the policy file are cached, to avoid a performance penalty for each query. The policy file is re-checked by each `impalad` node every 5 minutes. When you make a non-time-sensitive change such as adding new privileges or new users, you can let the change take effect automatically a few minutes later. If you remove or reduce privileges, and want the change to take effect immediately, restart the `impalad` daemon on all nodes, again specifying the `-server_name` and `-authorization_policy_file` options so that the rules from the updated policy file are applied.

Secure Startup for the impalad Daemon

To run the `impalad` daemon with authorization enabled, you add two options to the `IMPALA_SERVER_ARGS` declaration in the `/etc/default/impala` configuration file. The `-authorization_policy_file` option specifies the HDFS path to the policy file that defines the privileges on different schema objects. The rules in the policy file refer to a symbolic server name, and you specify a matching name as the argument to the `-server_name` option of `impalad`.

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

```
IMPALA_SERVER_ARGS=" \
  -authorization_policy_file=/user/hive/warehouse/auth-policy.ini \
  -server_name=server1 \
  ...
```

Then, the rules in the `[roles]` section of the policy file would refer to this same `server1` name. For example, the following rule sets up a role `report_generator` that lets users with that role query any table in a database

named `reporting_db` on a node where the `impalad` daemon was started up with the `-server_name=server1` option:

```
[roles]
report_generator = server=server1->db=reporting_db->table=*->action=SELECT
```

If the `impalad` daemon is not already running, start it as described in Starting Impala on page 51. If it is already running, restart it with the command `sudo /etc/init.d/impala-server restart`. Run the appropriate commands on all the nodes where `impalad` normally runs.

When `impalad` is started with one or both of the `-server_name=server1` and `-authorization_policy_file` options, Impala authorization is enabled. If Impala detects any errors or inconsistencies in the authorization settings or the policy file, the daemon refuses to start.

Examples of Policy File Rules for Security Scenarios

The following examples show rules that might go in the policy file to deal with various authorization-related scenarios. For illustration purposes, this section shows several very small policy files with only a few rules each. In your environment, typically you would define many roles to cover all the scenarios involving your own databases, tables, and applications, and a smaller number of groups, whose members are given the privileges from one or more roles.

A User with No Privileges

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

This is the experience you want a user to have if they somehow log into a system where they are not an authorized Impala user. In a real deployment with a filled-in policy file, a user might have no privileges because they are not a member of any of the relevant groups mentioned in the policy file.

Examples of Privileges for Administrative Users

When an administrative user has broad access to tables or databases, the associated rules in the `[roles]` section typically use wildcards and/or inheritance. For example, in the following sample policy file, `db=*` refers to all databases and `db=*->table=*` refers to all tables in all databases.

Omitting the rightmost portion of a rule means that the privileges apply to all the objects that could be specified there. For example, in the following sample policy file, the `all_databases` role has all privileges for all tables in all databases, while the `one_database` role has all privileges for all tables in one specific database. The `all_databases` role does not grant privileges on URIs, so a group with that role could not issue a `CREATE TABLE` statement with a `LOCATION` clause. The `entire_server` role has all privileges on both databases and URIs within the server.

```
[groups]
supergroup = all_databases

[roles]
read_all_tables = server=server1->db=*->table=*->action=SELECT
all_tables = server=server1->db=*->table=*
all_databases = server=server1->db=*
one_database = server=server1->db=test_db
entire_server = server=server1
```

A User with Privileges for Specific Databases and Tables

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

roles for the user that sets up a schema and other users or applications that perform day-to-day operations on the tables.

The following sample policy file shows some of the syntax that is appropriate as the policy file grows, such as the `#` comment syntax, `\` continuation syntax, and comma separation for roles assigned to groups or privileges assigned to roles.

```
[groups]
cloudera = training_sysadmin, instructor
visitor = student

[roles]
training_sysadmin = server=server1->db=training, \
  server=server1->db=instructor_private, \
  server=server1->db=lesson_development
instructor = server=server1->db=training->table=*->action=*, \
  server=server1->db=instructor_private->table=*->action=*, \
  server=server1->db=lesson_development->table=lesson*
# This particular course is all about queries, so the students can SELECT but not
INSERT or CREATE/DROP.
student = server=server1->db=training->table=lesson_*->action=SELECT
```

Privileges for Working with External Data Files

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

In this sample policy file:

- The `external_table` role lets us insert into and query the Impala table, `external_table.sample`.
- The `staging_dir` role lets us specify the HDFS path `/user/cloudera/external_data` with the `LOAD DATA` statement. Remember, when Impala queries or loads data files, it operates on all the files in that directory, not just a single file, so any Impala `LOCATION` parameters refer to a directory rather than an individual file.
- We included the IP address and port of the Hadoop name node in the HDFS URI of the `staging_dir` rule. We found those details in `/etc/hadoop/conf/core-site.xml`, under the `fs.default.name` element. That is what we use in any roles that specify URIs (that is, the locations of directories in HDFS).
- We start this example after the table `external_table.sample` is already created. In the policy file for the example, we have already taken away the `external_table_admin` role from the `cloudera` group, and replaced it with the lesser-privileged `external_table` role.
- We assign privileges to a subdirectory underneath `/user/cloudera` in HDFS, because such privileges also apply to any subdirectories underneath. If we had assigned privileges to the parent directory `/user/cloudera`, it would be too likely to mess up other files by specifying a wrong location by mistake.
- The `cloudera` under the `[groups]` section refers to the `cloudera` group. (In the demo VM used for this example, there is a `cloudera` user that is a member of a `cloudera` group.)

Policy file:

```
[groups]
cloudera = external_table, staging_dir

[roles]
external_table_admin = server=server1->db=external_table
external_table = server=server1->db=external_table->table=sample->action=*
staging_dir =
server=server1->uri=hdfs://127.0.0.1:8020/user/cloudera/external_data->action=*
```

`impala-shell` session:

```
[localhost:21000] > use external_table;
Query: use external_table
[localhost:21000] > show tables;
Query: show tables
```

```
Query finished, fetching results ...
+--------+
| name   |
+--------+
| sample |
+--------+
Returned 1 row(s) in 0.02s

[localhost:21000] > select * from sample;
Query: select * from sample
Query finished, fetching results ...
+-----+
| x   |
+-----+
| 1   |
| 5   |
| 150 |
+-----+
Returned 3 row(s) in 1.04s

[localhost:21000] > load data inpath '/user/cloudera/external_data' into table
sample;
Query: load data inpath '/user/cloudera/external_data' into table sample
Query finished, fetching results ...
+-----------------------------------------------------------+
| summary                                                   |
+-----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 2 |
+-----------------------------------------------------------+
Returned 1 row(s) in 0.26s
[localhost:21000] > select * from sample;
Query: select * from sample
Query finished, fetching results ...
+-------+
| x     |
+-------+
| 2     |
| 4     |
| 6     |
| 8     |
| 64738 |
| 49152 |
| 1     |
| 5     |
| 150   |
+-------+
Returned 9 row(s) in 0.22s

[localhost:21000] > load data inpath '/user/cloudera/unauthorized_data' into table
 sample;
Query: load data inpath '/user/cloudera/unauthorized_data' into table sample
ERROR: AuthorizationException: User 'cloudera' does not have privileges to access:
 hdfs://127.0.0.1:8020/user/cloudera/unauthorized_data
```

Controlling Access at the Column Level through Views

If a user has `SELECT` privilege for a view, they can query the view, even if they do not have any privileges on the underlying table. To see the details about the underlying table through `EXPLAIN` or `DESCRIBE FORMATTED` statements on the view, the user must also have `SELECT` privilege for the underlying table.

- **Important:**

  The types of data that are considered sensitive and confidential differ depending on the jurisdiction the type of industry, or both. For fine-grained access controls, set up appropriate privileges based on all applicable laws and regulations.

  Be careful using the `ALTER VIEW` statement to point an existing view at a different base table or a new set of columns that includes sensitive or restricted data. Make sure that any users who have `SELECT` privilege on the view do not gain access to any additional information they are not authorized to see.

The following example shows how a system administrator could set up a table containing some columns with sensitive information, then create a view that only exposes the non-confidential columns.

```
[localhost:21000] > create table sensitive_info
                  > (
                  >   name string,
                  >   address string,
                  >   credit_card string,
                  >   taxpayer_id string
                  > );
[localhost:21000] > create view name_address_view as select name, address from
sensitive_info;
```

Then the following policy file specifies read-only privilege for that view, without authorizing access to the underlying table:

```
[groups]
cloudera = view_only_privs

[roles]
view_only_privs = server=server1->db=reports->table=name_address_view->action=SELECT
```

Thus, a user with the `view_only_privs` role could access through Impala queries the basic information but not the sensitive information, even if both kinds of information were part of the same data file.

You might define other views to allow users from different groups to query different sets of columns.

The DEFAULT Database in a Secure Deployment

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

Separating Administrator Responsibility from Read and Write Privileges

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

For example, this policy file divides responsibilities between users in 3 different groups:

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

- Members of the `visitor` group have the `student` role and so can query those tables in the `training` database.

```
[groups]
supergroup = training_sysadmin
cloudera = instructor
visitor = student

[roles]
training_sysadmin = server=server1->db=training
instructor = server=server1->db=training->table=*->action=*
student = server=server1->db=training->table=*->action=SELECT
```

Setting Up Schema Objects for a Secure Impala Deployment

Remember that in the `[roles]` section of the policy file, you specify privileges at the level of individual databases and tables, or all databases or all tables within a database. To simplify the structure of these rules, plan ahead of time how to name your schema objects so that data with different authorization requirements is divided into separate databases.

If you are adding security on top of an existing Impala deployment, remember that you can rename tables or even move them between databases using the `ALTER TABLE` statement. In Impala, creating new databases is a relatively inexpensive operation, basically just creating a new directory in HDFS.

You can also plan the security scheme and set up the policy file before the actual schema objects named in the policy file exist. Because the authorization capability is based on whitelisting, a user can only create a new database or table if the required privilege is already in the policy file: either by listing the exact name of the object being created, or a `*` wildcard to match all the applicable objects within the appropriate container.

Privilege Model and Object Hierarchy

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

The kinds of objects in the schema hierarchy are:

```
Server
  URI
  Database
    Table
```

The server name is specified by the `-server_name` option when `impalad` starts. Specify the same name for all `impalad` nodes in the cluster.

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

There are not separate privileges for individual table partitions or columns. To specify read privileges at this level, you create a view that queries specific columns and/or partitions from a base table, and give `SELECT` privilege on the view but not the underlying table. See Views on page 103 for details about views in Impala.

URIs must start with either `hdfs://` or `file://`. If a URI starts with anything else, it will cause an exception and the policy file will be invalid. When defining URIs for HDFS, you must also specify the NameNode. For example:

```
data_read = server=server1->uri=file:///path/to/dir, \
  server=server1->uri=hdfs://namenode:port/path/to/dir
```

- **Warning:**

  Because the NameNode host and port must be specified, Cloudera strongly recommends you use High Availability (HA). This ensures that the URI will remain constant even if the namenode changes.

```
data_read = server=server1->uri=file:///path/to/dir,\
server=server1->uri=hdfs://ha-nn-uri/path/to/dir
```

Table 1: Valid Privilege types and objects they apply to

Privilege	Object
INSERT	TABLE, URI
SELECT	TABLE, VIEW, URI
ALL	SERVER, DB, URI

- **Note:**

  Although this document refers to the `ALL` privilege, currently, you do not use the actual keyword `ALL`. in the policy file. When you code role entries in the policy file:

  - To specify the `ALL` privilege for a server, use a role like `server=server_name`.
  - To specify the `ALL` privilege for a database, use a role like `server=server_name->db=database_name`.
  - To specify the `ALL` privilege for a table, use a role like `server=server_name->db=database_name->table=table_name->action=*`.

Table 2: Privilege table for Impala SQL operations

Impala SQL Operation	Privileges Required	Object on which Privileges Required
EXPLAIN	SELECT	Table
LOAD DATA	INSERT, SELECT	Table (INSERT) and URI (ALL); write privilege is required on the URI because the data files are physically moved from there
CREATE DATABASE	ALL	Database
DROP DATABASE	ALL	Database
DROP TABLE	ALL	Table
DESCRIBE TABLE	SELECT or INSERT	Table
ALTER TABLE	ALL	Table

Impala SQL Operation	Privileges Required	Object on which Privileges Required
SHOW DATABASES	Any privilege	Any object in the database; only databases for which the user has privileges on some object are shown
SHOW TABLES	Any privilege	Any table in the database; only tables for which the user has privileges are shown
CREATE VIEW	ALL, SELECT	You need ALL privilege on the named view, plus SELECT privilege for any tables or views referenced by the view query. Once the view is created or altered by a high-privileged system administrator, it can be queried by a lower-privileged user who does not have full query privileges for the base tables. (This is how you implement column-level security.)
DROP VIEW	ALL	Table
ALTER VIEW	ALL, SELECT	You need ALL privilege on the named view, plus SELECT privilege for any tables or views referenced by the view query. Once the view is created or altered by a high-privileged system administrator, it can be queried by a lower-privileged user who does not have full query privileges for the base tables. (This is how you implement column-level security.)
ALTER TABLE LOCATION	ALL	Table, URI
CREATE TABLE	ALL	Database
CREATE EXTERNAL TABLE	ALL, SELECT	Database (ALL), URI (SELECT)
SELECT	SELECT	Table, View; you can have SELECT privilege for a view without having SELECT privilege for the underlying tables, which allows a system administrator to implement column-level security by creating views that reference particular sets of columns
USE	Any privilege	Any object in the database

Using Multiple Policy Files for Different Databases

For an Impala cluster with many databases being accessed by many users and applications, it might be cumbersome to update the security policy file for each privilege change or each new database, table, or view. You can allow security to be managed separately for individual databases, by setting up a separate policy file for each database:

- Add the optional `[databases]` section to the main policy file.
- Add entries in the `[databases]` section for each database that has its own policy file.
- For each listed database, specify the HDFS path of the appropriate policy file.

For example:

```
[databases]
# Defines the location of the per-DB policy files for the 'customers' and 'sales'
```

```
databases.
customers = hdfs://ha-nn-uri/etc/access/customers.ini
sales = hdfs://ha-nn-uri/etc/access/sales.ini
```

Enabling Kerberos Authentication for Impala

Impala supports Kerberos authentication. For more information on enabling Kerberos authentication, see the topic on Configuring Hadoop Security in CDH4 in the CDH4 Security Guide. Impala currently does not support application data wire encryption. When using Impala in a managed environment, Cloudera Manager automatically completes Kerberos configuration. In an unmanaged environment, create a Kerberos principal for each host running `impalad` or `statestored`. Cloudera recommends using a consistent format, such as `impala/_HOST@Your-Realm`, but you can use any three-part Kerberos server principal.

Start all `impalad` and `statestored` daemons with the `--principal` and `--keytab-file` flags set to the principal and full path name of the `keytab` file containing the credentials for the principal. Impala supports the Cloudera ODBC driver and the Kerberos interface provided. To use Kerberos through the ODBC driver, the host type must be set to `SecImpala`. To enable Kerberos in the Impala shell, start the `impala-shell` command using the `-k` flag.

To enable Impala to work with Kerberos security on your Hadoop cluster, make sure you perform the installation and configuration steps in the topic on Configuring Hadoop Security in CDH4 in the CDH4 Security Guide. Also note that when Kerberos security is enabled in Impala, a web browser that supports Kerberos HTTP SPNEGO is required to access the Impala web console (for example, Firefox, Internet Explorer, or Chrome).

If the NameNode, Secondary NameNode, DataNode, JobTracker, TaskTrackers, ResourceManager, NodeManagers, HttpFS, Oozie, Impala, or Impala statestore services are configured to use Kerberos HTTP SPNEGO authentication, and two or more of these services are running on the same host, then all of the running services must use the same HTTP principal and keytab file used for their HTTP endpoints.

Configuring Impala to Support Kerberos Security

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

- Creating service principals for Impala and the HTTP service. Principal names take the form: `<serviceName>/<fully.qualified.domain.name>@<KERBEROS.REALM>`
- Creating, merging, and distributing key tab files for these principals.
- Editing `/etc/default/impala` to accommodate Kerberos authentication.

To enable Kerberos for Impala:

1. Create an Impala service principal. For example:

```
$ kadmin
kadmin: addprinc -randkey impala_1/impala_host.example.com@TEST.EXAMPLE.COM
```

2. Create an HTTP service principal. For example:

```
kadmin: addprinc -randkey HTTP/impala_host.example.com@TEST.EXAMPLE.COM
```

> **Note:** The `HTTP` component of the service principal must be upper case as shown in the preceding example.

3. Create `keytab` files with both principals. For example:

```
kadmin: xst -k impala.keytab impala/impala_host.example.com
kadmin: xst -k http.keytab HTTP/impala_host.example.com
kadmin: quit
```

4. Use `ktutil` to read the contents of the two keytab files and then write those contents to a new file. For example:

```
$ ktutil
ktutil: rkt impala.keytab
ktutil: rkt http.keytab
ktutil: wkt impala-http.keytab
ktutil: quit
```

5. (Optional) Test that credentials in the merged keytab file are valid. For example:

```
$ klist -e -k -t impala-http.keytab
```

6. Copy the `impala-http.keytab` file to the Impala configuration directory. Change the permissions to be only read for the file owner and change the file owner to the `impala` user. By default, the Impala user and group are both named `impala`. For example:

```
$ cp impala-http.keytab /etc/impala/conf
$ cd /etc/impala/conf
$ chmod 400 impala-http.keytab
$ chown impala:impala impala-http.keytab
```

7. Add Kerberos options to the Impala defaults file, `/etc/default/impala`. Add the options for both the `impalad` and `statestored` daemons, using the `IMPALA_SERVER_ARGS` and `IMPALA_STATE_STORE_ARGS` variables. For example, you might add:

```
-kerberos_reinit_interval=60
-kerberos_ticket_life=36000
-principal=impala_1/impala_host.example.com@TEST.EXAMPLE.COM
-keytab_file=/var/run/cloudera-scm-agent/process/3212-impala-IMPALAD/impala.keytab
```

For more information on changing the Impala defaults specified in `/etc/default/impala`, see Modifying Impala Startup Options.

> **Note:** Restart `impalad` and `statestored` for these configuration changes to take effect.

Using a Web Browser to Access a URL Protected by Kerberos HTTP SPNEGO

Your web browser must support Kerberos HTTP SPNEGO. For example, Chrome, Firefox, or Internet Explorer.

To configure Firefox to access a URL protected by Kerberos HTTP SPNEGO:

1. Open the advanced settings Firefox configuration page by loading the `about:config` page.
2. Use the **Filter** text box to find `network.negotiate-auth.trusted-uris`.
3. Double-click the `network.negotiate-auth.trusted-uris` preference and enter the hostname or the domain of the web server that is protected by Kerberos HTTP SPNEGO. Separate multiple domains and hostnames with a comma.
4. Click **OK**.

Auditing Impala Operations

To monitor how Impala data is being used within your organization, ensure that your Impala authorization and authentication policies are effective, and detect attempts at intrusion or unauthorized access to Impala data, you can use the auditing feature introduced in Impala 1.1.1:

- Enable auditing by including the option `-audit_event_log_dir=`*`directory_path`* in your `impalad` startup options. The path refers to a local directory on the server, not an HDFS directory.
- Decide how many queries will be represented in each log files. By default, Impala starts a new log file every 5000 queries. To specify a different number, include the option `-max_audit_event_log_file_size=`*`number_of_queries`* in the `impalad` startup options. Limiting the size lets you manage disk space by archiving older logs, and reduce the amount of text to process when analyzing activity for a particular period.
- Configure the Cloudera Navigator product to collect and consolidate the audit logs from all the nodes in the cluster.
- Use the Cloudera Manager product to filter, visualize, and produce reports based on the audit data. (The Impala auditing feature works with Cloudera Manager 4.7 or higher.) Check the audit data to ensure that all activity is authorized and/or detect attempts at unauthorized access.

Durability and Performance Considerations for Impala Auditing

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

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

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

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

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

Format of the Audit Log Files

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

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

Impala records more information for a successful operation than for a failed one, because an unauthorized query is stopped immediately, before all the query planning is completed.

The information logged for each query includes:

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

Which Operations Are Audited

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

- Queries that are prevented due to lack of authorization.
- Queries that Impala can analyze and parse to determine that they are authorized. The audit data is recorded immediately after Impala finishes its analysis, before the query is actually executed.

The audit log does not contain entries for queries that could not be parsed and analyzed. For example, a query that fails due to a syntax error is not recorded in the audit log. The audit log also does not contain queries that fail due to a reference to a table that does not exist, if you would be authorized to access the table if it did exist.

Certain statements in the `impala-shell` interpreter, such as `CONNECT`, `PROFILE`, `SET`, and `QUIT`, do not correspond to actual SQL queries, and these statements are not reflected in the audit log.

Reviewing the Audit Logs

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

Starting Impala

To begin using Cloudera Impala, start at least one instance of the Impala state store and then start Impala on one or more DataNodes. If you installed Impala with Cloudera Manager, use Cloudera Manager to start and stop services. To start the Impala state store and Impala using means other than Cloudera Manager, you can either use init scripts or you can start the state store and Impala directly.

Impala is designed to run in distributed mode using the Impala state store. While the Impala state store helps ensure the best performance, in the event the state store becomes unavailable, Impala continues to function. Start the Impala state store and then start `impalad` instances. You can modify the values the init scripts use when starting the state store and Impala by editing `/etc/default/impala`.

Start the state store using a command similar to the following:

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

Start Impala on each data node using a command similar to the following:

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

If the state store or Impala server fail to start, review:

- Reviewing Impala Logs on page 154
- Appendix B - Troubleshooting Impala on page 159

Once Impala is running, you can conduct interactive experiments using the instructions in Impala Tutorial on page 55 and try Using the Impala Shell on page 119.

Modifying Impala Startup Options

The Impala server and Impala state store start up using values provided in a defaults file, `/etc/default/impala`. You can check the current value of all these settings through the Impala web interface, available by default at `http://impala-node-hostname:25000/varz`.

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

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

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

To use alternate values, edit the defaults file, then restart the Impala server and the Impala state store so that the changes take effect. Restart the Impala server using the following commands:

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

Restart the Impala state store using the following commands:

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

Some common settings to change include:

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

  ```
  IMPALA_STATE_STORE_HOST=127.0.0.1
  ```

  to:

  ```
  IMPALA_STATE_STORE_HOST=192.168.0.27
  ```

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

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

  to:

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

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

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

- Core dump enablement. To enable core dumps, change:

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

  to:

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

  > **Note:** The location of core dump files may vary according to your operating system configuration. Other security settings may prevent Impala from writing core dumps even when this option is enabled.

- Authorization using the open source Sentry plugin. Specify the `-server_name` and `-authorization_policy_file` options as part of the `IMPALA_SERVER_ARGS` and `IMPALA_STATE_STORE_ARGS` settings to enable the core Impala support for authentication. See Secure Startup for the impalad Daemon on page 39 for details.
- Auditing for successful or blocked Impala queries, another aspect of security. Specify the `-audit_event_log_dir=`*`directory_path`* option and optionally the `-max_audit_event_log_file_size=`*`number_of_queries`* and `-abort_on_failed_audit_event` options as part of the `IMPALA_SERVER_ARGS` settings, for each Impala node, to enable and customize auditing. See Auditing Impala Operations on page 49 for details.
- Password protection for the Impala web UI, which listens on port 25000 by default. This feature involves adding some or all of the `--webserver_password_file`, `--webserver_authentication_domain`, and `--webserver_certificate_file` options to the `IMPALA_SERVER_ARGS` and `IMPALA_STATE_STORE_ARGS` settings. See Security Guidelines for Impala on page 36 for details.
- Another setting you might add to `IMPALA_SERVER_ARGS` is:

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

  These options control the behavior of queries performed by this `impalad` instance. The option values you specify here override the default values shown by the `SET` command in `impala-shell`.
- During troubleshooting, Cloudera Support might direct you to change other values, particularly for `IMPALA_SERVER_ARGS`, to work around issues or gather debugging information.

> **Note:**
>
> These startup options for the `impalad` daemon are different from the command-line options for the `impala-shell` command. For the `impala-shell` options, see impala-shell Command-Line Options on page 122.

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 Installing Cloudera Impala on page 15. 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.

Choose from these lessons:

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
 /user/cloudera/sample_data/tab3
```

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;
-- TAB1 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 ',';
```

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: show databases
Query finished, fetching results ...
default
Returned 1 row(s) in 0.00s
[impala-host:21000] > show tables
Query: show tables
Query finished, fetching results ...
customer
customer_address
Returned 2 row(s) in 0.00s
[impala-host:21000] > describe customer_address
Query: describe customer_address
Query finished, fetching results ...
+------------------+--------+---------+
| 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;
Query: select count(*) from customer_address
Query finished, fetching results ...
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
Query: select count(*) from customer_address
Query finished, fetching results ...
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
Query: select count(*) from customer_address
Query finished, fetching results ...
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  |
+----+-------+---------+-------+-----+
```

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_partitions.db/logs/year=2013/month=07/day=28/host=host1/3981726974111751120--8907184999369517436_822630111_data.0
$ hdfs dfs -cat
/user/impala/warehouse/external_partitions.db/logs/year=2013/month=07/day=28/host=host1/3981726974111751120--8
 907184999369517436_822630111_data.0
foo,foo,foo
```

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

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

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

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

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

```
use external_partitions;
alter table logs rename to logs_original;
create external table logs (field1 string, field2 string, field3 string)
  partitioned by (year string, month string, day string, host string)
  row format delimited fields terminated by ','
  location '/user/impala/data/logs';
```

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

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

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

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

Impala SQL Language Reference

- 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. 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. The Cloudera Impala 1.0 Release Notes contains information on the current differences.

For full details about the Impala SQL dialect, see Impala SQL Language Elements on page 65. For information about Hive syntax not available in Impala, see Unsupported Language Elements on page 106. For a list of the built-in functions available in Impala queries, see Built-in Function Support on page 107.

Impala SQL Language Elements

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 early Impala beta releases, a semicolon was optional at the end of each statement in the `impala-shell` interpreter. Now that the `impala-shell` interpreter supports multi-line commands, making it easy to copy and paste code from script files, the semicolon at the end of each statement is required.

Here are the major statements, clauses, and other SQL language elements that you work with in Impala:

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.

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:

```
ALTER TABLE table_name 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 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_spec;
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 table data to be in:

```
ALTER TABLE table_name SET FILEFORMAT { PARQUETFILE | RCFILE | SEQUENCEFILE |
TEXTFILE }
```

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. Although Impala can read Avro tables created through Hive, you cannot specify the Avro file format in an Impala `ALTER TABLE` statement.

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. 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 string);
```

```
-- Create an empty partition into which we could copy data files from some other
source.
alter table part_t add partition (month='January');
-- 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='February');
-- Now we no longer need the older data.
alter table part_t drop partition (month='January');
-- If the table was partitioned by month and year, we would issue a statement like:
-- alter table part_t drop partition (year=2003,month='January');
-- which would require 12 ALTER TABLE statements to remove a year's worth of data.
```

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));
```

Whenever you specify partitions in an `ALTER TABLE` statement, you must include all the partitioning columns in the specification.

Most of the preceding 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.

- **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.

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.

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

Examples:

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

AVG Function

An aggregation 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;
```

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

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;
```

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).

Related information: Mathematical Functions on page 108

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.

Related information: Conditional Functions on page 113

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 ...;
```

COUNT Function

An aggregation 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;
```

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.

The syntax for creating a database is as follows:

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

  [LOCATION hdfs_path];
```

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`.

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.

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;
```

CREATE TABLE Statement

The syntax for creating a table is as follows:

```
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'], ...)]
  [
   [ROW FORMAT row_format] [STORED AS file_format]
  ]
  [LOCATION 'hdfs_path']

data_type
  : primitive_type

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

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

file_format:
  : PARQUETFILE
  | SEQUENCEFILE
  | TEXTFILE
  | RCFILE
```

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 Impala leaves the data files in place when you drop the table.

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 125.

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 query an Avro table created by Hive but cannot create an Avro table; 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 131.

By default (when no STORED AS clause is specified), Impala tables are created as Text files with Ctrl-A characters as the delimiter. Specify the ROW FORMAT 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, use '\t' for tab, '\n' for carriage return, and '\r' for linefeed.

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 '\\'.

To create a 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. 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 131 for details.)

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

To see the column definitions and column comments for an existing table, 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*.

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.

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. Currently, to create this metadata, you issue the ANALYZE TABLE statement in Hive to gather this information, after creating the table and loading representative data into it.

- **Note:**

  The Impala CREATE TABLE statement cannot create an HBase table, because it currently does not support the TBLPROPERTIES 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 149.

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.

```
CREATE VIEW view_name [(column_list)]
  AS select_statement
```

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.

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;
```

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, whether or not the data is compressed, 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.

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.

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
parquetfile location '/user/cloudera/sample_data';
Query: create external table t2 (x int, y int, s string) stored as parquetfile
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
```

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.

> **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.

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.

Related information: Mathematical Functions on page 108

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.

The syntax for dropping a database is:

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

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

DROP TABLE Statement

The syntax for dropping a table is:

```
DROP TABLE [IF EXISTS] table_name
```

By default, Impala removes the associated HDFS directory and data files for the table. If the table was created with the `EXTERNAL` clause, Impala leaves all files and directories untouched.

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*.

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;
```

To drop a set of related tables quickly, either because you do not need them anymore or to free up space, consider putting them in the same database and using a single `DROP DATABASE` statement instead of a `DROP TABLE` statement for each table. See CREATE DATABASE Statement on page 72 for examples showing sequences of `CREATE DATABASE`, `CREATE TABLE`, and `DROP DATABASE` statements.

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.

```
DROP VIEW [database_name.]view_name
```

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:

```
[impalad-host:21000] > explain select count(*) from customer_address;
PLAN FRAGMENT 0
  PARTITION: UNPARTITIONED

  3:AGGREGATE
  |  output: SUM(<slot 0>)
  |  group by:
  |  tuple ids: 1
  |
  2:EXCHANGE
     tuple ids: 1

PLAN FRAGMENT 1
  PARTITION: RANDOM

  STREAM DATA SINK
    EXCHANGE ID: 2
    UNPARTITIONED

  1:AGGREGATE
  |  output: COUNT(*)
  |  group by:
  |  tuple ids: 1
  |
  0:SCAN HDFS
     table=default.customer_address #partitions=1 size=5.25MB
     tuple ids: 0
```

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.

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 quite 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 speed up a query so that it executes in seconds rather than minutes or hours.

External Tables

The syntax `CREATE EXTERNAL TABLE` sets up an Impala table that points at existing data files. 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.

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.

Related information: Mathematical Functions on page 108

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.

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.

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.

Currently, all the 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.

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 89.

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.

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.
- 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.)

> **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.

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 PARQUETFILE` 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 PARQUETFILE;
```

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;
Query: 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;
```

```
Query: 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.
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.
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;
```

Concurrency considerations: Each `INSERT` operation creates new data files with unique names, so you can run multiple `INSERT INTO` statements simultaneously with conflicts. If data is inserted into a table by a statement issued to a different `impalad` node, issue a `REFRESH` *`table_name`* statement to make the node you are connected to aware of this new data. 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.

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).

Related information: Mathematical Functions on page 108

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.

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 13 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:**

  The INVALIDATE METADATA statement is new in Impala 1.1, 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 or through a different Impala node, 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 Avro 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;
...
```

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.

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. 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.

- 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.

> **Note:**
>
> Join ordering affects Impala performance. For best results:
>
> - Join the biggest table first.
> - Join subsequent tables according to which table has the most selective filter. Joining the table with the most selective filter results in the fewest number of rows being returned.

The result set from a join query is filtered by including the corresponding join column names in an `ON` clause, or by using comparison operators referencing columns from both tables 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. For example, if `T1` contains 1000 rows and `T2` contains 1000 rows, a query `SELECT` *columns* `FROM t1 JOIN t2` could return up to 1 million rows (1000 * 1000). 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.

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 84.

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___';
```

LIMIT Clause

The `LIMIT` clause in a `SELECT` query sets a maximum number of rows for the result set. It 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, for a query with no `ORDER BY` clause.
- 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.

Top-N queries are so common, and tables queried by Impala are typically so large, that Impala requires any query including an `ORDER BY` clause to also use a `LIMIT` clause, to cap the size of the overall result set. You can specify the `LIMIT` clause as part of the query, or set a default limit for all queries.

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 ...)]
```

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.

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: select count(*) from t1
Query finished, fetching results ...
+-----+
| _c0 |
+-----+
| 10  |
+-----+
Returned 1 row(s) in 0.62s
```

MAX Function

An aggregation 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 aggregation 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;
```

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.

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.

Because an ORDER BY clause on a query returning a huge result set can require so much capacity to perform the sort, Impala requires any query including an ORDER BY clause to also use a LIMIT clause, to cap the size of the overall result set. You can specify the LIMIT clause as part of the query, or set a default limit for all queries with the command SET DEFAULT_ORDER_BY_LIMIT=... in impala-shell or the option -default_query_options default_order_by_limit=... when starting impalad. See SELECT Statement on page 98 for further examples.

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 13 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, or other table-modifying SQL statement in Impala, while connected to a different Impala node.
- After issuing ALTER TABLE, INSERT, 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 on the same Impala node, or because a previous INVALIDATE METADATA statement caused Impala to reload its entire metadata catalog.

- **Note:**

  The functionality of the REFRESH statement has changed in Impala 1.1. 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 the current Impala node is already aware of, when you create a new table in the Hive shell or while connected to a different Impala node, you must enter INVALIDATE METADATA with no table parameter before you can see the new table in impala-shell on the current node. Once the table is known to the Impala node, 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 by some other entity, meaning another impalad instance in your cluster or 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 - through ALTER, CREATE, DROP or INSERT operations.
- Impalad - through CREATE TABLE, ALTER TABLE, and INSERT operations.

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.

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;
...
```

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, that impala-shell option is not recommended for day-to-day use in a production environment.

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.

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 ^...$ aren't required, but make it clear we're matching 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(c_last_name) regexp
'de.*';
```

RLIKE Operator

Synonym for the REGEXP operator.

SELECT Statement

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.
- DISTINCT clause per query. See DISTINCT Operator on page 79 for details.
- Subqueries in a FROM clause.
- WHERE, GROUP BY, HAVING clauses.
- ORDER BY. Impala requires that queries using this keyword also include a LIMIT clause.

- **Note:**

  ORDER BY queries require limits on results. These limits can be set when you start Impala or they can be set in the Impala shell. Setting query options though ODBC or JDBC is not supported at this time, so in those cases, if you are using either of those connectors, set the limit value when starting Impala. For example, to set this value in the shell, use a command similar to:

  ```
  [impalad-host:21000] > set default_order_by_limit=50000
  ```

  To set the limit when starting Impala, include the -default_query_option startup parameter for the impalad daemon. For example, to start Impala with a result limit for ORDER BY queries, use a command similar to:

  ```
  $ GLOG_v=1 nohup impalad -state_store_host=state_store_hostname
  -hostname=impalad_hostname -default_query_options
  default_order_by_limit=50000
  ```

- JOIN clauses. Left, right, semi, full, and outer joins are supported. See Joins on page 89 for details.
- 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 Function Support on page 107.
- The WITH clause to abstract repeated clauses, such as aggregation functions, that are referenced multiple times in the same query. (The WITH clause actually comes before the SELECT statement in a query.)

SHOW Statement

To display a list of available databases or tables, issue these statements:

- SHOW TABLES [IN *database_name*]
- SHOW DATABASES
- SHOW SCHEMAS - an alias for SHOW DATABASES

For example, 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.

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).

Related information: Mathematical Functions on page 108

STRING Data Type

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

Length: 32,767 bytes. (Strictly speaking, the maximum length corresponds to the C/C++ constant INT_MAX, which is 32,767 for typical Linux systems.) 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. Multi-byte character data can be stored in Impala and retrieved through queries, but is 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 other type. You can use CAST() to convert STRING values to TINYINT, SMALLINT, INT, BIGINT, FLOAT, DOUBLE, or TIMESTAMP. 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.

Related information: Date and Time Functions on page 111

SUM Function

An aggregation 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;
```

TIMESTAMP Data Type

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

Range: Internally, the resolution of the time portion of a TIMESTAMP value is in nanoseconds.

Time zones: Impala does not store timestamps using the local timezone. Timestamps are stored relative to GMT.

Conversions: Impala automatically converts `STRING` literals of the correct format into `TIMESTAMP` values, for example, `'1966-07-30'` or `'1985-09-25 17:45:30.5'` 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).

Related information: Date and Time Functions on page 111

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).

Related information: Mathematical Functions on page 108

USE Statement

By default, when you connect to an Impala instance through the `impala-shell` command, 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.

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.

Examples:

See CREATE DATABASE Statement on page 72 for examples covering `CREATE DATABASE`, `USE`, and `DROP DATABASE`.

VALUES Clause

The `VALUES` clause is a general-purpose way to specify all the columns of a row or multiple rows. You typically use the `VALUES` clause in an `INSERT` statement to specify all the column values for one or more rows as they are added to a table.

- **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.

```
[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 does not support specifying a subset of the columns in the table or specifying the columns in a different order. Use a VALUES clause with all the column values in the same order as the table definition, using NULL values for any columns you want to omit from the INSERT operation.

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.

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 on page 42 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 parquetfile;
-- ...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 75, ALTER VIEW Statement on page 68, and DROP VIEW Statement on page 81. You can specify view names when querying data (SELECT Statement on page 98) and copying data from one table to another (INSERT Statement on page 85). 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 busiest_days as select day, howmany from visitors_by_day order by howmany
 desc;
create view top_10_days as select day, howmany from busiest_days limit 10;
select * from top_10_days;
```

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
```

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.

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;
```

Unsupported Language Elements

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
- `LOAD DATA` to load raw files
- Extensibility mechanisms such as `TRANSFORM`, custom User Defined Functions (UDFs), custom file formats, or custom SerDes
- XML and JSON functions
- Certain aggregation functions from HiveQL: `variance`, `var_pop`, `var_samp`, `stddev_pop`, `stddev_samp`, `covar_pop`, `covar_samp`, `corr`, `percentile`, `percentile_approx`, `histogram_numeric`, `collect_set`; Impala supports these aggregation functions: `MAX()`, `MIN()`, `SUM()`, `AVG()`, and `COUNT()`
- User Defined Aggregate Functions (UDAFs)
- User Defined Table Generating Functions (UDTFs)
- Sampling
- Lateral views
- Authorization features such as roles
- Multiple `DISTINCT` clauses per query

Impala does not currently support these HiveQL statements:

- `ANALYZE TABLE`
- `DESCRIBE COLUMN`
- `DESCRIBE DATABASE`
- `EXPORT TABLE`
- `IMPORT TABLE`

- SHOW PARTITIONS
- SHOW TABLE EXTENDED
- SHOW TBLPROPERTIES
- SHOW FUNCTIONS
- SHOW INDEXES
- SHOW COLUMNS
- SHOW CREATE TABLE

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. See Joins on page 89 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.
- 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.
  - Impala does perform implicit casts among the numeric types or from string to timestamp.
- Impala does not store timestamps using the local timezone. Timestamps are stored relative to GMT.
- Impala does not return column overflows as NULL. Instead, 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.
- Impala does not provide virtual columns.
- Impala does not expose locking.
- Impala does not expose some configuration properties.

Built-in Function Support

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.

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.

The categories of functions supported by Impala are:

- Mathematical Functions on page 108
- Type Conversion Functions on page 111
- Date and Time Functions on page 111
- Conditional Functions on page 113
- String Functions on page 114
- Aggregation functions: MAX(), MIN(), SUM(), AVG(), and COUNT()

Mathematical Functions

Impala supports the following mathematical functions:

abs(double a)

Purpose: Returns the absolute value of the argument.

Return type: double

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

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)

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

Return type: int

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`

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`

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`

negative(int a), negative(double a)

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

Return type: `int` or `double`, 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)

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

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

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`

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`

round(double a), round(double a, int 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 argument; `double` when second argument greater than zero

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. 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.

Impala supports the following data and time functions:

date_add(string startdate, int days)

Purpose: Adds a specified number of days to a date represented as a string.

Return type: string

date_sub(string startdate, int days)

Purpose: Subtracts a specified number of days from a date represented as a string.

Return type: `string`

`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`

`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`

`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`

`minute(string date)`

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

Return type: `int`

`month(string date)`

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

Return type: `int`

`now()`

Purpose: Returns the current date and time (in the UTC time zone) as a `timestamp` value.

Return type: `timestamp`

second(string date)

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

Return type: int

to_date(string timestamp)

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

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

unix_timestamp(), unix_timestamp(string date), unix_timestamp(string date, string pattern)

Purpose: Returns a timestamp representing the current date and time, or converts from a specified date and time value represented as a string.

Return type: bigint

weekofyear(string date)

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

Return type: int

year(string date)

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

Return type: int

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

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

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

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

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. Equivalent to the nvl() function from Oracle Database or ifnull() from MySQL.

Return type: same as the first argument value

nvl(type a, type ifNotNull)

Purpose: Alias for the isnull() function; added in Impala 1.1. 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

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

concat(string a, string b...)

Purpose: Returns a single string representing all the argument values joined together.

Return type: string

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

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

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.

Return type: string

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

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`

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:

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`

Using the Impala Shell

You can use the Impala shell tool (`impala-shell`) to set up databases and tables, insert data, and issue queries.

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 Installing Cloudera Impala on page 15. 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 on page 119.

For reference information about the `impala-shell` commands, see impala-shell Command Reference on page 120. For a list of the `impala-shell` command-line options, see impala-shell Command-Line Options on page 122.

Connecting to impalad

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 v1.0.1 (9ef893a) built on Fri May 31 17:50:30
 PDT 2013)
[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

For information on available commands, see impala-shell Command Reference on page 120. 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 section 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 66 for details.
`connect`	Connects to the specified instance of `impalad`. The default port of 21000 is assumed unless you provide another value. You can connect to any host in your cluster that is running `impalad`. If you connect to an instance of `impalad` that was started with an alternate port specified by the `--fe_port` flag, you must provide that alternate port. See Connecting to impalad on page 119 for examples.
`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 75 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 80 and DROP DATABASE Statement on page 80 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 81 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 ~/.impalahistory
`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 85 for details.
`profile`	Displays low-level information about the most recent query. Used for performance diagnosis and tuning.
`quit`	Exits the shell. Remember to include the final semicolon so that the shell recognizes the end of the command.
`refresh`	Refreshes impalad metadata. It is best to `refresh` metadata after making changes to databases such as adding or removing a table. See REFRESH Statement on page 96 for details.

Command	Explanation
`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 98 for details.
`set`	Manages query options for an `impala-shell` session. The available options are the ones listed in Modifying Impala Startup Options on page 51. 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, use the `unset` command. Some options take Boolean values of `true` and `false`. Others take numeric arguments, or quoted string values.
`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.
`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 99 for details.
`unset`	Removes any user-specified value for a query option and returns the option to its default value.
`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`. Not effective with the `-q` option, because that option only allows a single statement in the argument.
`version`	Returns Impala version information.

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 on page 51.

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	
-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=*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.)

Option	Explanation
-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.
-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.

Tuning Impala 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.

> **Note:**
>
> Before starting any performance tuning or benchmarking, make sure your system is configured with all the settings from Post-Installation Configuration for Impala on page 21.

- Partitioning on page 125. 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.
- Join Queries on page 126. 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 127 and Table Statistics on page 127 are also important primarily for join performance.
- Column Statistics on page 127. Gathering column statistics, using the Hive `ANALYZE TABLE` statement, helps Impala automatically optimize the performance for join queries, without requiring changes to SQL query statements. (Column statistics have a greater influence on join performance than table statistics do, but both kinds of statistics are important.)
- Table Statistics on page 127 Gathering table statistics, using the Hive `ANALYZE TABLE` statement, helps Impala automatically optimize the performance for join queries, without requiring changes to SQL query statements. (Column statistics have a greater influence on join performance than table statistics do, but both kinds of statistics are important.)
- Testing Impala Performance on page 129. Do some post-setup testing to ensure Impala is using optimal settings for performance, before conducting any benchmark tests.
- Benchmarking Impala Queries on page 128. 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 129. 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.

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.

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.

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.

See Attaching an External Partitioned Table to an HDFS Directory Structure on page 62 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.

See Partitioning for Parquet Tables on page 138 for performance considerations for partitioned Parquet tables.

See NULL on page 95 for details about how `NULL` values are represented in partitioned tables.

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 never finishes and has to be cancelled.

The simplest technique for tuning an Impala join query is to specify the tables in the optimal order. Specify the largest table first, then the next largest, and so on; the smallest table is specified last. The terms "largest" and "smallest" refers to the number of rows 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.

When possible, the Impala query planner examines metadata about the size of each table and the number of different values in each row, to help choose the appropriate join strategy. To gather table statistics, issue an `ANALYZE TABLE` statement within Hive. To gather column statistics, issue another `ANALYZE TABLE` statement

within Hive. Analyze the table after loading data into it, so the statistics reflect realistic values. Analyze the table again after replacing the original data with new data with substantially different size or distribution, or after appending a large volume of new data. If this metadata is not available, Impala queries use the broadcast join mechanism by default.

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 84 for details.

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.)

Add settings like the following to the `hive-site.xml` configuration file, in the Hive configuration directory, on every node where you run `ANALYZE TABLE` statements through the `hive` shell. The `hive.stats.ndv.error` setting represents the standard error when estimating the number of distinct values for a column. The value of 5.0 is recommended as a tradeoff between the accuracy of the gathered statistics and the resource usage of the stats-gathering process.

```
<property>
 <name>hive.stats.ndv.error</name>
 <value>5.0</value>
</property>
```

5.0 is a relatively low value that devotes substantial computational resources to the statistics-gathering process. To reduce the resource usage, you could increase this value; to make the statistics even more precise, you could lower it.

The syntax for gathering column statistics uses the `ANALYZE TABLE ... COMPUTE STATISTICS` clause, with an additional `FOR COLUMNS` clause. For partitioned tables, you can gather statistics for specific partitions by including a clause `PARTITION (col1=val1,col2=val2, ...)`; but you cannot include the partitioning columns in the `FOR COLUMNS` clause. Also, you cannot use fully qualified table names, so issue a `USE` command first to switch to the appropriate database. For example:

```
USE database_name;
ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS column_list;
ANALYZE TABLE table_name PARTITION (partition_specs) COMPUTE STATISTICS FOR COLUMNS
 column_list;
```

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. Currently, Impala does not create this metadata itself. To gather this metadata for a new or existing 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;
```

Here are the current guidelines and known limitations for the table statistics feature:

- Because table statistics collection is a compute-intensive operation, it can impose significant overhead on the Impala and Hive metastore database. If you work with large tables, as most Impala users do, expect to configure a separate database on a different node than the metastore database to hold the statistics information.
- Table statistics can currently only be stored in a MySQL database.
- Add settings like the following to the `hive-site.xml` configuration file, in the Hive configuration directory, on every node where you run `ANALYZE TABLE` statements through the `hive` shell:

```
<property>
  <name>hive.stats.dbclass</name>
  <value>jdbc:mysql</value>
</property>
<property>
  <name>hive.stats.jdbcdriver</name>
  <value>com.mysql.jdbc.Driver</value>
</property>
<property>
  <name>hive.stats.dbconnectionstring</name>

<value>jdbc:mysql://hostname:port_number/metastore?useUnicode=true&amp;characterEncoding=UTF-8&amp;
user=hiveuser&amp;password=password</value>
</property>
<property>
  <name>hive.stats.autogather</name>
  <value>true</value>
</property>
```

- Specify a `hive.stats.dbconnectionstring` that matches the configuration values for the MySQL database in the Hive metastore configuration settings. Include the appropriate user name and password as parameters in the connection string.

The Impala query planner uses table statistics in combination with column statistics when both kinds of information are available.

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 122 for details.

Controlling Resource Usage

You can restrict the amount of memory Impala reserves during query execution by specifying the `-mem_limits` option for the `impalad` daemon. See Modifying Impala Startup Options on page 51 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.

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 on page 51 and Using the Impala Shell on page 119. Information on logging can be found in Using Impala Logging on page 153. 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.

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 3: 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`, 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`, and query. If LZO compression is used, you must create the table and load data in Hive.
Avro	Structured	Snappy, GZIP, deflate, BZIP2	No, create using Hive.	No, query only. Load data using Hive.
RCFile	Structured	Snappy, GZIP, deflate, BZIP2	Yes.	No, query only. Load data using Hive.
SequenceFile	Structured	Snappy, GZIP, deflate, BZIP2	Yes.	No, query only. Load data using 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.
- GZIP. Recommended when achieving the highest level of compression (and therefore greatest disk-space savings) is desired.
- Deflate.

- BZIP2.
- 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.

See the following sections for details about using text data files with Impala tables:

- Query Performance for Impala Text Tables on page 132
- Creating Text Tables on page 133
- Data Files for Text Tables on page 133
- Loading Data into Impala Text Tables on page 134
- Using LZO-Compressed Text Files on page 134

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. 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.

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, with several clauses at the end; the significant part is the `FIELDS TERMINATED BY` clause. For example:

```
create table csv(id int, s string, n int, t timestamp, b boolean)
  stored as textfile
  fields terminated by ',';

create table tsv(id int, s string, n int, t timestamp, b boolean)
  stored as textfile
  fields terminated by '\t';

create table pipe_separated(id int, s string, n int, t timestamp, b boolean)
  stored as textfile
  fields terminated by '|';
```

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.

- **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.

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.

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 files that employ LZO compression. Because Impala can query LZO-compressed files but currently cannot write them, use Hive to do the initial `CREATE TABLE` and load the data.

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.

1. **Prepare your systems to work with LZO using Cloudera repositories:**

   Download and install the appropriate file to each machine on which you intend to use LZO with Impala. 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 `hadoop-lzo-cdh4` package, and install the `impala-lzo-cdh4` package.

   For RHEL/CentOS systems:

```
$ sudo yum update
$ sudo yum install hadoop-lzo-cdh4
$ sudo yum install impala-lzo
```

For SUSE systems:

```
$ sudo apt-get update
$ sudo zypper install hadoop-lzo-cdh4
$ sudo zypper install impala-lzo
```

For Debian/Ubuntu systems:

```
$ sudo zypper update
$ sudo apt-get install hadoop-lzo-cdh4
$ 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-cdh4` 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>
```

4. Restart the MapReduce 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'
```

Files in an LZO-compressed table must use the `.lzo` extension. After loading data into such a table, index the files so that they can be split. Indexing is done by running the LZO indexer, `com.hadoop.compression.lzo.DistributedLzoIndexer`, which is included in the `hadoop-lzo` package.

Run the indexer using:

```
$ hadoop jar /usr/lib/hadoop/lib/hadoop-lzo-cdh4-0.4.15-gplextras.jar
  com.hadoop.compression.lzo.DistributedLzoIndexer /hdfs_location_of_table/
```

Indexed files have the same name as the file they index, except that index files use the `.index` extension. Impala can read non-indexed files, but such reading is typically done from remote DataNodes, which is very inefficient.

In Hive, when writing LZO compressed text tables, you must include the following specification:

```
hive> SET hive.exec.compress.output=true;
hive> SET mapred.output.compression.codec=com.hadoop.compression.lzo.LzopCodec;
```

Impala does not currently support writing LZO-compressed Text files.

Once you have created tables, you can also take data stored in one format and use Hive to transform existing tables to another format. If you have existing data, you might begin by transforming data from one format into another. Hive provides an easy way to convert tables between formats. To convert tables using Hive, create a new table that uses the desired format, then select all rows from the existing table and insert those into the new table.

> **Note:** The commands for transforming data are unaffected by the input table format.

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.

See the following sections for details about using Parquet data files with Impala tables:

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
PARQUETFILE;
```

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 PARQUETFILE;
```

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 table, perhaps in a different file format or partitioning scheme, you can transfer the to a Parquet table using the `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 138 for some examples showing how to insert data into Parquet tables.

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 1GB 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 1GB 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 140 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 1GB 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 1GB 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.

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 1GB 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 1GB 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.

Partitioning for Parquet Tables

As explained in Partitioning on page 125, 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 141, 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 1GB block size, so when deciding how finely to partition the data, try to find a granularity where each partition contains 1GB 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.xcievers. 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.
- 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.

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 does not compress especially well, 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
```

Example of Uncompressed Parquet Table

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 1GB, 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 1GB 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

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 10, 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:

```
ALTER TABLE table_name SET FILEFORMAT PARQUETFILE;
```

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 1GB. (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 DiscCP 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 1GB, 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 1GB, 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 140 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 1GB 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 1GB, or a multiple of 1GB.

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.

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, but currently cannot create them or insert data into them. For those operations, use Hive, then switch back to Impala to run queries.

See the following sections for details about using Avro data files with Impala tables:

- Using a Hive-Created Avro Table in Impala on page 143
- Creating Avro Tables on page 143

Creating Avro Tables

To create a new table using the Avro file format, use the Hive shell. Impala does not currently support the `CREATE TABLE` clauses required to define the schema for an Avro table. Once the table is created in Hive, switch back to `impala-shell` and issue an `INVALIDATE METADATA` statement. Then you can run queries for that table through `impala-shell`.

The following example demonstrates creating an Avro table in Hive:

```
hive> CREATE TABLE new_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": ["int", "null"]}]}');
OK
Time taken: 6.372 seconds
```

Each field of the record becomes a column of the table. Note that any other information, such as the record name, is ignored.

Using a Hive-Created Avro Table in Impala

Once you have an Avro table created through Hive, you can use it in Impala as long as it contains only Impala-compatible data types. (For example, it cannot contain nested types such as `array`, `map`, or `struct`.) 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.

After 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. If you issue Impala queries through more than one node, issue the `INVALIDATE METADATA` statement on each node, the first time you connect after creating the new table in Hive.

After you load new data into an Avro table, 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`. If you issue Impala queries through more than one node, issue the `REFRESH` statement on each node, the first time you connect after loading new data without using Impala.

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');
```

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.

See the following sections for details about using RCFile data files with Impala tables:

- Creating RCFile Tables on page 146
- Enabling Compression for RCFile Tables on page 146

Creating RCFile Tables

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 131 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.

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 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_rc PARTITION(year) SELECT * FROM tbl;
```

- **Note:**

  The compression type is specified in the following phrase:

  ```
  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.

See the following sections for details about using SequenceFile data files with Impala tables:

- Creating SequenceFile Tables on page 147
- Enabling Compression for SequenceFile Tables on page 147

Creating SequenceFile Tables

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 131 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.

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 phrase:

  ```
  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.

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 the `TBLPROPERTIES` keyword, custom SerDes, and some other syntax needed for these tables. You map these specially created tables to corresponding tables that exist in HBase, with the clause `TBLPROPERTIES("hbase.table.name" = "`*`table_name_in_hbase`*`")`. See Examples of Querying HBase Tables from Impala on page 150 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/hbase.accesscontrol.configuration.html.

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 150 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.

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.

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. 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 on page 51 for details.

Examples of Querying HBase Tables from Impala

You can map an HBase table to a Hive table with or without string row keys. The following examples use HBase with the following table definition:

```
create 'hbasealltypessmall', 'bools', 'ints', 'floats', 'strings'
enable 'hbasealltypessmall'
```

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.)

```
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" = "hbasealltypesagg");
```

Without a String Row Key

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

```
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
```

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 on page 49 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 on page 51.
- 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.
- 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.

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 155 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
```

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 large log files.

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. See Configuring Impala to Work with ODBC for details.
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 upcoming version 2.0 of the Cloudera ODBC driver. See Configuring Impala to Work with JDBC on page 29 for details.
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 Service Port	24000	Internal	Internal use only. State store listens on this port for registration/unregistration requests.

Component	Service	Port	Access Requirement	Comment
Impala StateStore Daemon	StateStore HTTP Server Port	25010	External	StateStore web interface for administrators to monitor and troubleshoot.

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, all data from both sets to be joined is loaded into memory. As data sets can be very large, this can easily exceed the total memory available.	Add more memory to your system or join smaller data sets.
Queries return incorrect results.	Impala metadata may be outdated.	Refresh the Hive metadata as described in REFRESH in the Language Reference.
Queries are slow to return results.	Some `impalad` instances may not have started. Using a browser, connect to the host running the Impala state store. Connect using an address of the form `http://hostname:port/metrics`. **Note:** Replace *hostname* and *port* with the hostname and port of your Impala state store host machine and web server port. The default port is 25010. The number of `impalad` instances listed should match the expected number of `impalad` instances installed in the cluster. There should also be one `impalad` instance installed on each DataNode	Ensure Impala is installed on all DataNodes. Start any `impalad` instances that are not running.

Symptom	Explanation	Recommendation
Queries are slow to return results.	Impala may not be configured to use native checksumming. Native checksumming uses machine-specific instructions to compute checksums over HDFS data very quickly. Review Impala logs. If you find instances of "`INFO util.NativeCodeLoader: Loaded the native-hadoop`" messages, native checksumming is not enabled.	Ensure Impala is configured to use native checksumming as described in Post-Installation Configuration for Impala on page 21.
Queries are slow to return results.	Impala may not be configured to use data locality tracking.	Test Impala for data locality tracking and make configuration changes as necessary. Information on this process can be found in Post-Installation Configuration for Impala on page 21.
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.