Hortonworks Data Platform

Data Services

(March 1, 2016)

docs.cloudera.com

Hortonworks Data Platform: Data Services

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including YARN, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to Contact Us directly to discuss your specific needs.

Table of Contents

List of Figures

List of Tables

1. Using Apache Hive

Hortonworks Data Platform deploys Apache Hive for your Hadoop cluster.

Hive is a data warehouse infrastructure built on top of Hadoop. It provides tools to enable easy data ETL, a mechanism to put structures on the data, and the capability for querying and analysis of large data sets stored in Hadoop files.

Hive defines a simple SQL query language, called QL, that enables users familiar with SQL to query the data. At the same time, this language also allows programmers who are familiar with the MapReduce framework to be able to plug in their custom mappers and reducers to perform more sophisticated analysis that may not be supported by the built-in capabilities of the language.

In this document:

1.1. Hive Documentation

Documentation for Hive can be found in wiki docs and javadocs.

1. Javadocs describe the Hive API.
2. The Hive wiki is organized in four major sections:
   a. General Information about Hive
      - Getting Started
      - Presentations and Papers about Hive
      - Hive Mailing Lists
   b. User Documentation
      - Hive Tutorial
      - SQL Language Manual
      - Hive Operators and Functions

- Hive Client
- Beeline: HiveServer2 Client
- Avro SerDe

c. Administrator Documentation

- Installing Hive
- Configuring Hive
- Setting Up the Metastore
- Setting Up Hive Web Interface
- Setting Up Hive Server
- Hive on Amazon Web Services
- Hive on Amazon Elastic MapReduce

d. Resources for Contributors

- Hive Developer FAQ
- How to Contribute
- Hive Developer Guide
- Plug-in Developer Kit
- Unit Test Parallel Execution
- Hive Architecture Overview
- Hive Design Docs
- Full-Text Search over All Hive Resources
- Project Bylaws

1.2. Features Overview

The following sections provide brief descriptions of Hive features:

- Temporary Tables [3]
- Cost-Based SQL Optimization [3]
- Optimized Row Columnar (ORC) Format [6]
- Streaming Data Ingestion [6]
- Query Vectorization [6]

- Comparing Beeline to the Hive CLI [7]

1.2.1. Temporary Tables

Temporary tables are supported in Hive 0.14 and later. A temporary table is a convenient way for an application to automatically manage intermediate data generated during a complex query. Rather than manually deleting tables needed only as temporary data in a complex query, Hive automatically deletes all temporary tables at the end of the Hive session in which they are created. The data in these tables is stored in the user's scratch directory rather than in the Hive warehouse directory. The scratch directory effectively acts as the data sandbox for a user, located by default in /tmp/hive-<username>.

Tip

See Apache AdminManual Configuration for information on configuring Hive to use a non-default scratch directory.

Hive users create temporary tables using the TEMPORARY keyword:

```
CREATE TEMPORARY TABLE tmp1 (c1 string);
CREATE TEMPORARY TABLE tmp2 AS ...
CREATE TEMPORARY TABLE tmp3 LIKE ...
```

Multiple Hive users can create multiple Hive temporary tables with the same name because each table resides in a separate session.

Temporary tables support most table options, but not all. The following features are not supported:

- Partitioned columns
- Indexes

Note

A temporary table with the same name as a permanent table will cause all references to that table name to resolve to the temporary table. The user cannot access the permanent table during that session without dropping or renaming the temporary table.

1.2.2. Cost-Based SQL Optimization

Cost-based optimization (CBO) of SQL queries is supported in Hive 0.13.0 and later. CBO uses Hive table, table partition, and column statistics to create efficient query execution plans. Efficient query plans better utilize cluster resources and improve query latency. CBO is most useful for complex queries that contain multiple JOIN statements and for queries on very large tables.

Note

Tables are not required to have partitions to generate CBO statistics. Column-level CBO statistics can be generated by both partitioned and unpartitioned tables.

CBO generates the following statistics:

Statistics Granularity	Description
Table-level	- Uncompressed size of table - Number of rows - Number of files
Column-level	- Number of distinct values - Number of NULL values - Minimum value - Maximum value

CBO requires column-level statistics to generate the best query execution plans. Later, when viewing these statistics from the command line, you can choose to also include table-level statistics that are generated by the `hive.stats.autogather` configuration property. However, CBO does not use these table-level statistics to generate query execution plans.

Note

See Statistics in Hive for more information.

Enabling Cost-based SQL Optimization

Hortonworks recommends that administrators always enable CBO. Set and verify the following configuration parameters in hive-site.xml to enable cost-based optimization of SQL queries:

Table 1.1. CBO Configuration Parameters

CBO Configuration Parameter	Description	Default Value
`hive.cbo.enable`	Enables cost-based query optimization.	False
`hive.stats.autogather`	Enables automated gathering of table-level statistics for newly created tables and table partitions, such as tables created with the INSERT OVERWRITE statement. The parameter does not produce column-level statistics, such as those generated by CBO. If disabled, administrators must manually generate these table-level statistics with the ANALYZE TABLE statement.	True

The following configuration properties are not specific to CBO, but setting them to `true` will also improve the performance of queries that generate statistics:

Configuration Parameter	Description	Default Value
`hive.stats.fetch.column.stats`	Instructs Hive to collect column-level statistics.	False
`hive.compute.query.using.stats`	Instructs Hive to use statistics when generating query plans.	False

Generating Statistics

Use the ANALYZE TABLE command to generate statistics for tables and columns. Use the optional NoScan clause to improve query performance by preventing a scan of files on HDFS. This option gathers only the following statistics:

- Number of files
- Size of files in bytes

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

The following example views statistics for all partitions in the employees table. The query also uses the NoScan clause to improve performance:

```
ANALYZE TABLE employees PARTITION (dt) COMPUTE STATISTICS [NoScan];
```

Generating Column-level Statistics:

Use the following syntax to generate statistics for columns in the employee table:

```
ANALYZE TABLE tablename [PARTITION(partcol1[1=val1], partcol2[=val2], ...)]
 COMPUTE STATISTICS FOR COLUMNS [NoScan];
```

The following example generates statistics for all columns in the employees table:

```
ANALYZE TABLE employees PARTITION (dt) COMPUTE STATISTICS FOR COLUMNS;
```

Viewing Statistics

Use the DESCRIBE statement to view statistics generated by CBO. Include the EXTENDED keyword if you want to include statistics gathered when the `hive.stats.fetch.column.stats` and `hive.compute.query.using.stats` properties are enabled.

- Viewing Generated Table Statistics
  - Use the following syntax to generate table statistics:

    ```
    DESCRIBE [EXTENDED] tablename;
    ```

    Note

    The EXTENDED keyword can be used only if the `hive.stats.autogather` property is enabled in the hive-site.xml configuration file.

  - The following example displays all statistics for the employees table:

    ```
    DESCRIBE EXTENDED employees;
    ```

- Viewing Generated Column Statistics
  - Use the following syntax to generate column statistics:

    ```
    DESCRIBE FORMATTED [dbname.]tablename.columnname;
    ```

  - The following example displays statistics for the region column in the employees table:

```
DESCRIBE FORMATTED employees.region;
```

1.2.3. Optimized Row Columnar (ORC) Format

ORC-based tables are supported in Hive 0.14.0 and later. These tables can contain more than 1,000 columns. For more information about how the ORC file format enhances Hive performance, see LanguageManual ORC on the Apache site.

1.2.4. Streaming Data Ingestion

Note

If you have questions regarding this feature, contact Support by logging a case on our Hortonworks Support Portal at http://support.hortonworks.com.

Limitations

Hive 0.13 and 0.14 have the following limitations to ingesting streaming data:

- Only ORC files are supported
- Destination tables must be bucketed
- Apache Flume or Apache Storm may be used as the streaming source

1.2.5. Query Vectorization

Vectorization allows Hive to process a batch of rows together instead of processing one row at a time. Each batch is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage. HIVE-4160 has the design document for vectorization and tracks the implementation of many subtasks.

Enable Vectorization in Hive

To enable vectorization, set this configuration parameter:

```
hive.vectorized.execution.enabled=true
```

When vectorization is enabled, Hive examines the query and the data to determine whether vectorization can be supported. If it cannot be supported, Hive will execute the query with vectorization turned off.

Log Information about Vectorized Execution of Queries

The Hive client will log, at the `info` level, whether a query's execution is being vectorized. More detailed logs are printed at the `debug` level.

The client logs can also be configured to show up on the console.

Supported Functionality

The current implementation supports only single table read-only queries. DDL queries or DML queries are not supported.

The supported operators are **selection**, **filter** and **group by**.

Partitioned tables are supported.

These data types are supported:

- tinyint
- smallint
- int
- bigint
- date
- boolean
- float
- double
- timestamp
- string
- char
- varchar
- binary

These expressions are supported:

- Comparison: >, >=, <, <=, =, !=
- Arithmetic: plus, minus, multiply, divide, modulo
- Logical: AND, OR
- Aggregates: sum, avg, count, min, max

Only the ORC file format is supported in the current implementation.

Unsupported Functionality

All datatypes, file formats, and functionality are currently unsupported.

Two unsupported features of particular interest are the logical expression `NOT`, and the `cast` operator. For example, a query such as `select x,y from T where a = b` will not vectorize if `a` is integer and `b` is double. Although both `int` and `double` are supported, casting of one to another is not supported.

1.2.6. Comparing Beeline to the Hive CLI

HDP supports two Hive clients: the Hive CLI and Beeline. The primary difference between the two involves how the clients connect to Hive.

- The Hive CLI, which connects directly to HDFS and the Hive Metastore, and can be used only on a host with access to those services.
- Beeline, which connects to HiveServer2 and requires access to only one .jar file: `hive-jdbc-<version>-standalone.jar`.

Hortonworks recommends using HiveServer2 and a JDBC client (such as Beeline) as the primary way to access Hive. This approach uses SQL standard-based authorization or Ranger-based authorization. However, some users may wish to access Hive data from other applications, such as Pig. For these use cases, use the Hive CLI and storage-based authorization.

Beeline Operating Modes and HiveServer2 Transport Modes

Beeline supports the following modes of operation:

Table 1.2. Beeline Modes of Operation

Operating Mode	Description
Embedded	The Beeline client and the Hive installation both reside on the same host machine. No TCP connectivity is required.
Remote	Use remote mode to support multiple, concurrent clients executing queries against the same remote Hive installation. Remote transport mode supports authentication with LDAP and Kerberos. It also supports encryption with SSL. TCP connectivity is required.

Administrators may start HiveServer2 in one of the following transport modes:

Table 1.3. HiveServer2 Transport Modes

Transport Mode	Description
TCP	HiveServer2 uses TCP transport for sending and receiving Thrift RPC messages.
HTTP	HiveServer2 uses HTTP transport for sending and receiving Thrift RPC messages.

While running in TCP transport mode, HiveServer2 supports the following authentication schemes:

Table 1.4. Authentication Schemes with TCP Transport Mode

Authentication Scheme	Description
Kerberos	A network authentication protocol which operates that uses the concept of 'tickets' to allow nodes in a network to securely identify themselves. Administrators must specify hive.server2.authentication=kerberos in the hive-site.xml configuration file to use this authentication scheme.
LDAP	The Lightweight Directory Access Protocol, an application-layer protocol that uses the concept of 'directory services' to share information across a network. Administrators must specify hive.server2.authentication=ldap in the hive-site.xml configuration file to use this type of authentication.
PAM	Pluggable Authentication Modules, or PAM, allow administrators to integrate multiple authentication schemes into a single API. Administrators must specify

Authentication Scheme	Description
	hive.server2.authentication=pam in the hive-site.xml configuration file to use this authentication scheme.
Custom	Authentication provided by a custom implementation of the org.apache.hive.service.auth.PasswdAuthenticationProvider interface. The implementing class must be available in the classpath for HiveServer2 and its name provided as the value of the hive.server2.custom.authentication.class property in the hive-site.xml configuration property file.
None	The Beeline client performs no authentication with HiveServer2. Administrators must specify hive.server2.authentication=none in the hive-site.xml configuration file to use this authentication scheme.
NoSASL	While running in TCP transport mode, HiveServer2 uses the Java Simple Authentication and Security Layer (SASL) protocol to establish a security layer between the client and server. However, HiveServer2 also supports connections in TCP transfer mode that do not use the SASL protocol Administrators must specify hive.server2.authentication=nosasl in the hive-site.xml configuration file to use this authentication scheme.

The next section describes the connection strings used to connect to HiveServer2 for all possible combinations of these modes, as well as the connection string required to connect to HiveServer2 in a secure cluster.

Connecting to Hive with Beeline

The following examples demonstrate how to use Beeline to connect to Hive for all possible variations of these modes.

Embedded Client

Use the following syntax to connect to Hive from Beeline in embedded mode:

```
!connect jdbc:hive2://
```

Remote Client with HiveServer2 TCP Transport Mode and SASL Authentication

Use the following syntax to connect to HiveServer2 in TCP mode from a remote Beeline client:

```
!connect jdbc:hive2://<host>:<port>/<db>
```

The default port for HiveServer2 in TCP mode is `10000`, and `db` is the name of the database to which you want to connect.

Remote Client with HiveServer2 TCP Transport Mode and NoSASL Authentication

Clients must explicitly specify the authentication mode in their connection string when HiveServer2 runs in NoSASL mode:

```
!connect jdbc:hive2://<host>:<port>/<db>;auth=noSasl hiveuser pass org.apache.
hive.jdbc.HiveDriver
```

If users forget to include `auth=noSasl` in the JDBC connection string, the JDBC client API attempts to make an SASL connection to HiveServer2. This causes an open connection that eventually results in the client crashing with an Out Of Memory error.

Remote Client with HiveServer2 HTTP Transport Mode

Use the following syntax to connect to HiveServer2 in HTTP mode from a remote Beeline client:

```
!connect jdbc:hive2://<host>:<port>/<db>?hive.server2.transport.mode=
http;hive.server2.thrift.http.path=<http_endpoint>
```

Remote Client with HiveServer2 in Secure Cluster

Use the following syntax to connect to HiveServer2 in a secure cluster from a remote Beeline client:

```
!connect jdbc:hive2://<host>:<port>/<db>;principal=
<Server_Principal_of_HiveServer2>
```

Note

The Beeline client must have a valid Kerberos ticket in the ticket cache before attempting to connect.

1.3. Moving Data into Hive

There are multiple methods of moving data into Hive. How you move the data into Hive depends on the source format of the data and the target data format that is required. Generally, ORC is the preferred target data format because of the performance enhancements that it provides.

The following methods are most commonly used:

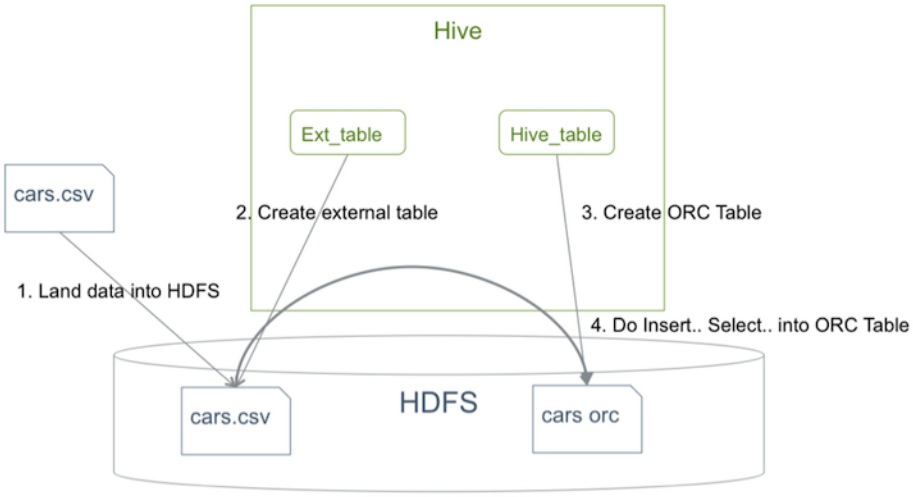
Table 1.5. Most Common Methods to Move Data into Hive

Source of Data	Target Data Format in Hive	Method Description
ETL for legacy systems	ORC file format	1. Move data into HDFS. 2. Use an external table to move data from HDFS to Hive. 3. Then use Hive to convert the data to the ORC file format.
Operational SQL database	ORC file format	1. Use Sqoop to import the data from the SQL database into Hive. 2. Then use Hive to convert the data to the ORC file format.
Streaming source that is "append only"	ORC file format	1. Write directly to the ORC file format using the Hive Streaming feature.

1.3.1. Moving Data from HDFS to Hive Using an External Table

This is the most common way to move data into Hive when the ORC file format is required as the target data format. Then Hive can be used to perform a fast parallel and distributed conversion of your data into ORC. The process is shown in the following diagram:

Figure 1.1. Example: Moving .CSV Data into Hive



Moving .CSV Data into Hive

The following steps describe moving .CSV data into Hive using the method illustrated in the above diagram with command-line operations.

1. **Move .CSV data into HDFS**:

   a. The following is a .CSV file which contains a header line that describes the fields and subsequent lines that contain the data:

   ```
   [<username>@cn105-10 ~]$ head cars.csv
   Name,Miles_per_Gallon,Cylinders,Displacement,Horsepower,Weight_in_lbs,
   Acceleration,Year,Origin
   "chevrolet chevelle malibu",18,8,307,130,3504,12,1970-01-01,A
   "buick skylark 320",15,8,350,165,3693,11.5,1970-01-01,A
   "plymouth satellite",18,8,318,150,3436,11,1970-01-01,A
   "amc rebel sst",16,8,304,150,3433,12,1970-01-01,A
   "ford torino",17,8,302,140,3449,10.5,1970-01-01,A
   ...
   [<username>@cn105-10 ~]$
   ```

   <username> is the user who is performing the operation. To test this example, run with a user from your environment.

   b. First, use the following command to remove the header line from the file because it is not part of the data for the table:

   ```
   [<username>@cn105-10 ~]$ sed -i 1d cars.csv
   ```

   c. Move the data to HDFS:

   ```
   [<username>@cn105-10 ~]$ hdfs dfs -copyFromLocal cars.csv /user/
   <username>/visdata
   [<username>@cn105-10 ~]$ hdfs dfs -ls /user/<username>/visdata
   Found 1 items
   -rwxrwxrwx   3 <username> hdfs      22100 2015-08-12 16:16 /user/
   <username>/visdata/cars.csv
   ```

2. **Create an external table**.

   An *external table* is a table for which Hive does not manage storage. If you delete an external table, only the definition in Hive is deleted. The data remains. An *internal table* is a table that Hive manages. If you delete an internal table, both the definition in Hive *and* the data are deleted.

   The following command creates an external table:

```
CREATE EXTERNAL TABLE IF NOT EXISTS Cars(
        Name STRING,
        Miles_per_Gallon INT,
        Cylinders INT,
        Displacement INT,
        Horsepower INT,
        Weight_in_lbs INT,
        Acceleration DECIMAL,
        Year DATE,
        Origin CHAR(1))
    COMMENT 'Data about cars from a public database'
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    STORED AS TEXTFILE
    location '/user/<username>/visdata';
```

3. **Create the ORC table**.

   Now, create a table that is managed by Hive with the following command:

```
CREATE TABLE IF NOT EXISTS mycars(
        Name STRING,
        Miles_per_Gallon INT,
        Cylinders INT,
        Displacement INT,
        Horsepower INT,
        Weight_in_lbs INT,
        Acceleration DECIMAL,
        Year DATE,
        Origin CHAR(1))
    COMMENT 'Data about cars from a public database'
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    STORED AS ORC;
```

4. **Insert the data from the external table to the Hive ORC table**.

   Now, use an SQL statement to move the data from the external table that you created in Step 2 to the Hive-managed ORC table that you created in Step 3:

```
INSERT OVERWRITE TABLE mycars SELECT * FROM cars;
```

   Note

   Using Hive to convert an external table into an ORC file format is very efficient because the conversion is a parallel and distributed action, and no standalone ORC conversion tool is necessary.

5. **Verify that you imported the data into the ORC-formatted table correctly**:

```
hive> select * from mycars limit 3;
OK
"chevrolet chevelle malibu" 18 8 307 130 3504 12 1970-01-01 A
"buick skylark 320" 15 8 350 165 3693 12 1970-01-01 A
"plymouth satellite" 18 8 318 150 3436 11 1970-01-01 A
Time taken: 0.144 seconds, Fetched: 3 row(s)
```

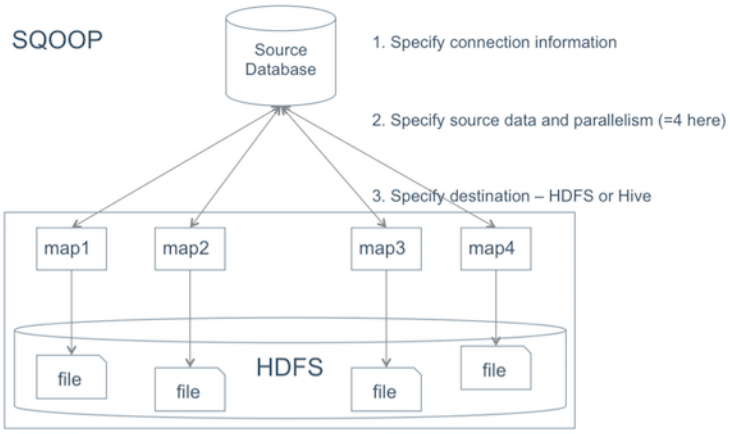
1.3.2. Using Sqoop to Move Data into Hive

Sqoop is a tool that enables you to bulk import and export data from a database. You can use Sqoop to import data into HDFS or directly into Hive. However, Sqoop can only import data into Hive as a text file or as a SequenceFile. To use the ORC file format, you must use a two-phase approach: first use Sqoop to move the data into HDFS, and then use Hive to convert the data into the ORC file format as described in the above Steps 3 and 4 of "Moving Data from HDFS to Hive Using an External Table."

A detailed Sqoop user guide is available on the Apache web site here.

The process for using Sqoop to move data into Hive is shown in the following diagram:

Figure 1.2. Using Sqoop to Move Data into Hive



Moving Data into Hive Using Sqoop

1. **Specify the source connection information**.

   First, you must specify the:

   - database URI (`db.foo.com` in the following example)
   - database name (`bar`)
   - connection protocol (`jdbc:mysql:`)

   For this example, use the following command:

   ```
   sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES
   ```

   If the source database requires credentials, such as a username and password, you can enter the password on the command line or specify a file where the password is stored.

For example:

- Enter the password on the command line:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --
username <username> -P
Enter password: (hidden)
```

- Specify a file where the password is stored:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --
username <username> --password-file ${user.home}/.password
```

More connection options are described in the Sqoop User Guide on the Apache web site.

2. **Specify the data and the parallelism for import**:

   a. Specify the data simply.

   Sqoop provides flexibility to specify exactly the data you want to import from the source system:

   - Import an **entire table**:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES
```

   - Import a **subset of the columns** from a table:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --
columns "employee_id,first_name,last_name,job_title"
```

   - Import only the **latest records** by specifying them with a WHERE clause and then that they be appended to an existing table:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES
 --where "start_date > '2010-01-01'"

sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --
where "id > 100000" --target-dir /incremental_dataset --append
```

   You can also use a free-form SQL statement.

   b. Specify parallelism.

   There are three options for specifying *write parallelism* (number of map tasks):

   - Explicitly set the number of mappers using `--num-mappers`. Sqoop evenly splits the primary key range of the source table:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --
num-mappers 8
```

     In this scenario, the source table must have a primary key.

   - Provide an alternate split key using `--split-by`. This evenly splits the data using the alternate split key instead of a primary key:

```
sqoop import --connect jdbc:mysql://db.foo.com/bar --table EMPLOYEES --
split-by dept_id
```

This method is useful if primary keys are not evenly distributed.

- When there is not split key or primary key, the data import must be sequential. Specify a single mapper by using `--num-mappers 1` or `--autoreset-to-one-mapper`.

c. Specify the data using a query.

Instead of specifying a particular table or columns, you can specify the date with a query. You can use one of the following options:

- Explicitly specify a *split-by column* using `--split-by` and put `$ CONDITIONS` that Sqoop replaces with range conditions based on the split-by key. This method requires a target directory:

```
sqoop import --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id)
WHERE $CONDITIONS' --split-by a.id --target-dir /user/foo/joinresults
```

- Use sequential import if you cannot specify a split-by column:

```
sqoop import --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id)
WHERE $CONDITIONS' -m 1 --target-dir /user/foo/joinresults
```

To try a sample query without importing data, use the `eval` option to print the results to the command prompt:

```
sqoop eval --connect jdbc:mysql://db.foo.com/bar --query "SELECT * FROM
 employees LIMIT 10"
```

3. **Specify the destination for the data: HDFS or Hive**.

Here is an example of specifying the HDFS target directory:

```
sqoop import --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id)
WHERE $CONDITIONS' --split-by a.id --target-dir /user/foo/joinresults
```

If you can add text data into your Hive table, you can specify that the data be directly added to Hive. Using `--hive-import` is the primary method to add text data directly to Hive:

```
sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES --
hive-import
```

This method creates a metastore schema after storing the text data in HDFS.

If you have already moved data into HDFS and want to add a schema, use the `create-hive-table` Sqoop command:

```
sqoop create-hive-table (generic-args) (create-hive-table-args)
```

Additional options for importing data into Hive with Sqoop:

Table 1.6. Sqoop Command Options for Importing Data into Hive

Sqoop Command Option	Description
`--hive-home <directory>`	Overrides `$HIVE_HOME`.
`--hive-import`	Imports tables into Hive using Hive's default delimiters if none are explicitly set.
`--hive-overwrite`	Overwrites existing data in the Hive table.
`--create-hive-table`	Creates a hive table during the operation. If this option is set and the Hive table already exists, the job will fail. Set to `false` by default.
`--hive-table <table_name>`	Specifies the table name to use when importing data into Hive.
`--hive-drop-import-delims`	Drops the delimiters `\n`, `\r`, and `\01` from string fields when importing data into Hive.
`--hive-delims-replacement`	Replaces the delimiters `\n`, `\r`, and `\01` from strings fields with a user-defined string when importing data into Hive.
`--hive-partition-key`	Specifies the name of the Hive field on which a sharded database is partitioned.
`--hive-partition-value <value>`	A string value that specifies the partition key for data imported into Hive.
`--map-column-hive <map>`	Overrides the default mapping from SQL type to Hive type for configured columns.

1.3.3. Incrementally Updating a Hive Table Using Sqoop and an External Table

It is common to perform a one-time ingestion of data from an operational database to Hive and then require incremental updates periodically. Currently, Hive does not support SQL Merge for bulk merges from operational systems. Instead, you must perform periodic updates as described in this section.

Note

This procedure requires change data capture from the operational database that has a primary key and modified date field where you pulled the records from since the last update.

Overview

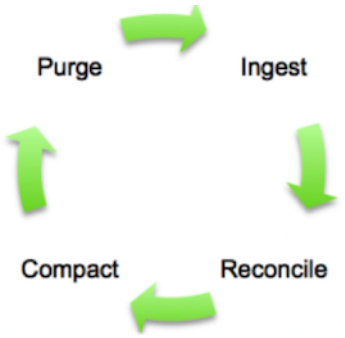
This procedure combines the techniques that are described in the sections "Moving Data from HDFS to Hive Using an External Table" and "Using Sqoop to Move Data into Hive."

Use the following steps to incrementally update Hive tables from operational database systems:

1. **Ingest:** Complete data movement from the operational database (base_table) followed by change or update of changed records only (incremental_table).
2. **Reconcile:** Create a single view of the base table and change records (reconcile_view) to reflect the latest record set.
3. **Compact:** Create a reporting table (reporting_table) from the reconciled view.

4. **Purge:** Replace the base table with the reporting table contents and delete any previously processed change records before the next data ingestion cycle, which is shown in the following diagram.

Figure 1.3. Data Ingestion Lifecycle



The base table is a Hive internal table, which was created during the first data ingestion. The incremental table is a Hive external table, which likely is created from .CSV data in HDFS. This external table contains the changes (INSERTs and UPDATEs) from the operational database since the last data ingestion.

Note

Generally, the table is partitioned and only the latest partition is updated, making this process more efficient.

Incrementally Updating Data in Hive

1. **Ingest the data.**

   a. Store the base table in the ORC format in Hive.

   The first time that data is ingested, you must import the entire table from the source database. You can use Sqoop. The following example shows importing data from Teradata:

```
sqoop import --connect jdbc:teradata://{host name}/Database=retail
--connection-manager org.apache.sqoop.teradata.TeradataConnManager --
username dbc
--password dbc --table SOURCE_TBL --target-dir /user/hive/base_table -m 1
```

   b. Store this data into an ORC-formatted table using the Steps 2 - 5 shown in "Moving Data from HDFS to Hive Using an External Table."

   The base table definition after moving it from the external table to a Hive-managed table looks like the below example:

```
CREATE TABLE base_table (
        id STRING,
        field1 STRING,
        modified_date DATE)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    STORED AS ORC;
```

c. Store the incremental table as an external table in Hive.

It is more common to be importing incremental changes since the last time data was updated and then merging it. See the section "Using Sqoop to Move Data into Hive" for examples of importing data with Sqoop.

In the following example, `--check-column` is used to fetch records newer than `last_import_date`, which is the date of the last incremental data update:

```
sqoop import --connect jdbc:teradata://{host name}/Database=retail
--connection-manager org.apache.sqoop.teradata.TeradataConnManager
--username dbc --password dbc --table SOURCE_TBL --target-dir /user/hive/
incremental_table -m 1
--check-column modified_date --incremental lastmodified --last-value
 {last_import_date}
```

You can also use `--query` to perform the same operation:

```
sqoop import --connect jdbc:teradata://{host name}/Database=retail
--connection-manager org.apache.sqoop.teradata.TeradataConnManager --
username dbc
--password dbc --target-dir /user/hive/incremental_table -m 1
--query 'select * from SOURCE_TBL where modified_date >
 {last_import_date} AND $CONDITIONS'
```

d. After the incremental table data is moved into HDFS using Sqoop, you can define an external Hive table over it with the following command:

```
CREATE EXTERNAL TABLE incremental_table (
        id STRING,
        field1 STRING,
        modified_date DATE)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    STORED AS TEXTFILE
    location '/user/hive/incremental_table';
```

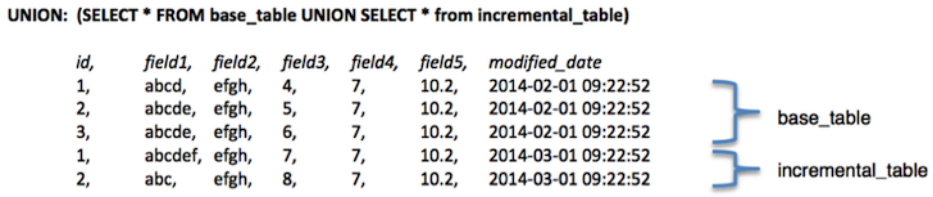
2. **Reconcile or merge the data.**

Create a view that uses UNION ALL to merge the data and reconcile the base table records with the new records:

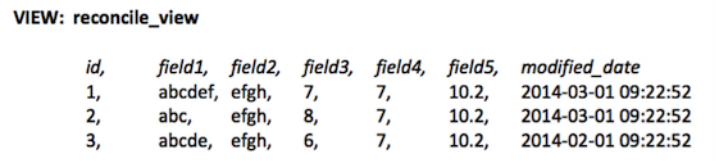
```
CREATE VIEW reconcile_view AS
SELECT t1.* FROM
    (SELECT * FROM base_table
     UNION ALL
     SELECT * from incremental_table) t1
JOIN
    (SELECT id, max(modified_date) max_modified FROM
        (SELECT * FROM base_table
         UNION ALL
         SELECT * from incremental_table)
     GROUP BY id) t2
ON t1.id = t2.id AND t1.modified_date = t2.max_modified;
```

EXAMPLES:

- **Figure 1.4. Dataset after the UNION ALL Command Is Run**



- **Figure 1.5. Dataset in the View**



Note

In the `reconcile_view` only one record exists per primary key, which is shown in the `id` column. The values displayed in the `id` column correspond to the latest modification date that is displayed in the `modified_date` column.

3. **Compact the data.**

   The view changes as soon as new data is introduced into the incremental table in HDFS (`/user/hive/incremental_table`, so create and store a copy of the view as a snapshot in time:

```
DROP TABLE reporting_table;
CREATE TABLE reporting_table AS
SELECT * FROM reconcile_view;
```

4. **Purge data.**

   a. After you have created a reporting table, clean up the incremental updates to ensure that the same data is not read twice:

```
hadoop fs –rm –r /user/hive/incremental_table/*
```

   b. Move the data into the ORC format as the base table. Frequently, this involves a partition rather than the entire table:

```
DROP TABLE base_table;
CREATE TABLE base_table (
        id STRING,
        field1 STRING,
        modified_date DATE)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    STORED AS ORC;

INSERT OVERWRITE TABLE base_table SELECT * FROM reporting_table;
```

Handling Deletes

Deletes can be handled by adding a DELETE_DATE field in the tables:

```
CREATE VIEW reconcile_view AS
    SELECT t1.* FROM
        (SELECT * FROM base_table
         UNION
         SELECT * FROM incremental_table) t1
     JOIN
         (SELECT id, max(modified_date) max_modified FROM
             (SELECT * FROM base_table
              UNION
              SELECT * FROM incremental_table)
      GROUP BY id) t2
   ON t1.id = t2.id AND t1.modified_date = t2.max_modified
   AND t1.delete_date IS NULL;
```

Tip

You can automate the steps to incrementally update data in Hive by using Oozie. See "Using HDP for Workflow and Scheduling (Oozie)."

1.4. Hive JDBC and ODBC Drivers

Hortonworks provides Hive JDBC and ODBC drivers that let you connect to popular Business Intelligence (BI) tools to query, analyze and visualize data stored within the Hortonworks Data Platform.

Hive ODBC drivers for Linux and Mac operating systems are installed when you install Hive. For information about using the Hive ODBC drivers, see the Hive ODBC Driver User Guide.

JDBC URLs have the following format:

```
jdbc:hive2://<host>:<port>/<dbName>;<sessionConfs>?<hiveConfs>#<hiveVars>
```

JDBC Parameter	Description
`host`	The cluster node hosting HiveServer2.
`port`	The port number to which HiveServer2 listens.
`dbName`	The name of the Hive database to run the query against.
`sessionConfs`	Optional configuration parameters for the JDBC/ODBC driver in the following format: `<key1>=<value1>;<key2>=<key2>...;`
`hiveConfs`	Optional configuration parameters for Hive on the server in the following format: `<key1>=<value1>;<key2>=<key2>; ...` The configurations last for the duration of the user session.
`hiveVars`	Optional configuration parameters for Hive variables in the following format: `<key1>=<value1>;<key2>=<key2>; ...` The configurations last for the duration of the user session.

The specific JDBC connection URL for a HiveServer2 client depends on several factors:

- How is HiveServer2 deployed on the cluster?
- What type of transport does HiveServer2 use?
- Does HiveServer2 use transport-layer security?
- Is HiveServer2 configured to authenticate users?

The rest of this section describes how to use session configuration variables to format the JDBC connection URLs for all these scenarios.

Note

Some HiveServer2 clients may need to run on a host outside of the Hadoop cluster. These clients require access to the following .jar files to successfully use the Hive JDBC driver in both HTTP and HTTPS modes: `hive-jdbc-<version>-standalone.jar`, `hadoop-common.jar`, and `hadoop-auth.jar`.

Embedded and Remote Modes

In embedded mode, HiveServer2 runs within the Hive client rather than in a separate process. No host or port number is necessary for the JDBC connection. In remote mode, HiveServer2 runs as a separate daemon on a specified host and port, and the JDBC client and HiveServer2 interact using remote procedure calls with the Thrift protocol.

Embedded Mode

```
jdbc:hive2://
```

Remote Mode

```
jdbc:hive2://<host>:<port>/<dbName>;<sessionConfs>?<hiveConfs>#<hiveVars>
```

Note

The rest of the example JDBC connection URLs in this topic are valid only for HiveServer2 configured in remote mode.

TCP and HTTP Transport

The JDBC client and HiveServer2 can use either HTTP or TCP-based transport to exchange RPC messages. Specify the transport used by HiveServer2 with the `transportMode` and `httpPath` session configuration variables. The default transport is TCP.

transportMode Variable Value	Description
`http`	Connect to HiveServer2 using HTTP transport.
`binary`	Connect to HiveServer2 using TCP transport.

HTTP Transport

```
jdbc:hive2://<host>:<port>/<dbName>;transportMode=http;httpPath=
<http_endpoint>;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

Note

The JDBC driver assumes a value of `cliservice` if the `httpPath` configuration variable is not specified.

TCP Transport

```
jdbc:hive2://<host>:<port>/<dbName>;<otherSessionConfs>?
<hiveConfs>#<hiveVars>
```

Because the default transport is TCP, there is no need to specify `transportMode=binary` if TCP transport is desired.

User Authentication

HiveServer2 supports Kerberos, LDAP, Pluggable Authentication Modules (PAM), and custom plugins for authenticating the JDBC user connecting to HiveServer2. The format of the JDBC connection URL for authentication with Kerberos differs from the format for other authentication models.

User Authentication Variable	Description
`principal`	A string that uniquely identifies a Kerberos user.
`saslQop`	Quality of protection for the SASL framework. The level of quality is negotiated between the client and server during authentication. Used by Kerberos authentication with TCP transport.
`user`	Username for non-Kerberos authentication model.
`password`	Password for non-Kerberos authentication model.

Kerberos Authentication

```
jdbc:hive2://<host>:<port>/<dbName>;principal=
<HiveServer2_kerberos_principal>;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

Kerberos Authentication with Sasl QOP

```
jdbc:hive2://<host>:<port>/<dbName>;principal=
<HiveServer2_kerberos_principal>;saslQop=<qop_value>;<otherSessionConfs>?
<hiveConfs>#<hiveVars>
```

Non-Kerberos Authentication

```
jdbc:hive2://<host>:<port>/<dbName>;user=<username>;password=
<password>;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

Transport Layer Security

HiveServer2 supports SSL and Sasl QOP for transport-layer security. The format of the JDBC connection URL for SSL differs from the format used by Sasl QOP.

SSL Variable	Description
`ssl`	Specifies whether to use SSL
`sslTrustStore`	The path to the SSL TrustStore.
`trustStorePassword`	The password to the SSL TrustStore.

```
jdbc:hive2://<host>:<port>/<dbName>;ssl=true;sslTrustStore=
<ssl_truststore_path>;trustStorePassword=
<truststore_password>;<otherSessionConfs>?<hiveConfs>#<hiveVars>
```

When using TCP for transport and Kerberos for security, HiveServer2 uses Sasl QOP for encryption rather than SSL.

Sasl QOP Variable	Description
`principal`	A string that uniquely identifies a Kerberos user.
`saslQop`	The level of protection desired. For authentication, checksum, and encryption, specify `auth-conf`. The other valid values do not provide encryption.

```
jdbc:hive2://<host>:<port>/<dbName>;principal=
<HiveServer2_kerberos_principal>;saslQop=auth-conf;<otherSessionConfs>?
<hiveConfs>#<hiveVars>
```

1.5. Configuring HiveServer2 for Transactions (ACID Support)

Hive supports transactions that adhere to traditional relational database ACID characteristics: atomicity, consistency, isolation, and durability. See the article about ACID characteristics on Wikipedia for more information.

Limitations

Currently, ACID support in Hive has the following limitations:

- `BEGIN`, `COMMIT`, and `ROLLBACK` are not yet supported.
- Only the ORC file format is supported.
- Transactions are configured to be off by default.
- Tables that use transactions, must be bucketed. For a discussion of bucketed tables, see the Apache site.
- Hive ACID only supports Snapshot Isolation. Transactions only support auto-commit mode and may include exactly one SQL statement.
- ZooKeeper and in-memory lock managers are not compatible with transactions. See the Apache site for a discussion of how locks are stored for transactions.
- Schema changes made by using ALTER TABLE are not supported. HIVE-11421 is tracking this issue.

To configure HiveServer2 for transactions:

Important

- Ensure that the `hive.txn.timeout` property is set to the same value in the `hive-site.xml` file for HiveServer2 that you configure in Step 1 below and the `hive-site.xml` file for the standalone Hive metastore that you configure in Step 2.
- The following listed properties are the minimum that are required to enable transaction support on HiveServer2. For additional information about

configuring this feature and for information about additional configuration parameters, see Hive Transactions on the Apache web site.

1. Set the following parameters in the `hive-site.xml` file:

```
<property>
     <name>hive.support.concurrency</name>
     <value>true</value>
</property>

<property>
     <name>hive.txn.manager</name>
     <value>org.apache.hadoop.hive.ql.lockmgr.DbTxnManager</value>
</property>

<property>
     <name>name>hive.enforce.bucketing</name>
     <value>true</value>
</property>

<property>
     <name>hive.exec.dynamic.partition.mode</name>
     <value>nostrict</value>
</property>
```

2. Ensure that a standalone Hive metastore is running with the following parameters set in its `hive-site.xml` file:

```
<property>
     <name>hive.compactor.initiator.on</name>
     <value>true</value>
</property>

<property>
     <name>hive.compactor.worker.threads</name>
     <value><positive_number></value>
</property>
```

Important

These are the minimum properties required to enable transactions in the standalone Hive metastore. See Hive Transactions on the Apache web site for information about configuring Hive for transactions and additional configuration parameters.

Even though HiveServer2 runs with an embedded metastore, a standalone Hive metastore is required for ACID support to function properly. If you are not using ACID support with HiveServer2, you do not need a standalone metastore.

The default value for `hive.compactor.worker.threads` is `0`. Set this to a positive number to enable Hive transactions. Worker threads spawn MapReduce jobs to perform compactions, but they do not perform the compactions themselves. Increasing the number of worker threads decreases the time that it takes tables or partitions to be compacted. However, increasing the number of worker threads also increases the background load on the Hadoop cluster because they cause more MapReduce jobs to run in the background.

1.6. Configuring HiveServer2 for LDAP and for LDAP over SSL

HiveServer2 supports authentication with LDAP and LDAP over SSL (LDAPS):

To configure HiveServer2 to use LDAP:

1. Add the following properties to the `hive-site.xml` file to set the server authentication mode to LDAP:

```
<property>
     <name>hive.server2.authentication</name>
     <value>LDAP</value>
</property>

<property>
     <name>hive.server2.authentication.ldap.url</name>
     <value>LDAP_URL</value>
</property>
```

   Where LDAP_URL is the access URL for your LDAP server. For example, `ldap://ldap_host_name@xyz.com:389`.

2. Depending on whether or not you use Microsoft Active Directory as your directory service, add the following additional properties to the `hive-site.xml` file:

   - **Other LDAP service types including OpenLDAP**:

```
<property>
     <name>hive.server2.authentication.ldap.baseDN</name>
     <value>LDAP_BaseDN</value>
</property>
```

     Where `LDAP_BaseDN` is the base LDAP distinguished name for your LDAP server. For example, `ou=dev, dc=xyz, dc=com`.

   - **Active Directory** (AD):

```
<property>
     <name>hive.server2.authentication.ldap.Domain</name>
     <value>AD_Domain</value>
</property>
```

     Where `AD_Domain` is the domain name of the AD server. For example, `corp.domain.com`.

3. Test the LDAP authentication. For example, if you are using the Beeline client, type the following commands at the Beeline prompt:

```
beeline>!connect
jdbc:hive2://node1:<port>/default:user=<LDAP_USERID>;password=
<LDAP_PASSWORD>
```

The Beeline client prompts for the user ID and password again. Enter those values to run the command.

To configure HiveServer2 to use LDAP over SSL (LDAPS):

To enable Hive and the Beeline client to use LDAPS, perform the following actions.

Note

Two types of certificates can be used for LDAP over SSL with HiveServer2:

- *CA Certificates*, which are digital certificates that are signed by a Certificate Authority (CA).
- Self-signed certificates.

1. Add the following properties to the `hive-site.xml` file to set the server authentication mode to LDAP:

```
<property>
     <name>hive.server2.authentication</name>
     <value>LDAP</value>
</property>

<property>
     <name>hive.server2.authentication.ldap.url</name>
     <value>LDAP_URL</value>
</property>
```

   Where LDAP_URL is the access URL for your LDAP server. For example, `ldap://ldap_host_name@xyz.com:389`.

2. Depending on whether or not you use Microsoft Active Directory as your directory service, add the following additional properties to the `hive-site.xml` file:

   - **Other LDAP service types including OpenLDAP**:

```
<property>
     <name>hive.server2.authentication.ldap.baseDN</name>
     <value>LDAP_BaseDN</value>
</property>
```

     Where LDAP_BaseDN is the base LDAP distinguished name for your LDAP server. For example, `ou=dev, dc=xyz, dc=com`.

   - **Active Directory** (AD):

```
<property>
     <name>hive.server2.authentication.ldap.Domain</name>
     <value>AD_Domain</value>
</property>
```

     Where AD_Domain is the domain name of the AD server. For example, `corp.domain.com`.

3. Depending on which type of certificate you are using, perform one of the following actions:

- **CA certificate**:

  If you are using a certificate that is signed by a CA, the certificate is already included in the default Java trustStore located at `${JAVA_HOME}/jre/lib/security/cacerts` on all of your nodes. If the CA certificate is not present, you must import the certificate to your Java `cacert` trustStore using the following command:

  ```
  keytool -import -trustcacerts -alias <MyHiveLdaps>
  -storepass <password> -noprompt -file <myCert>.pem -keystore ${JAVA_HOME}/
  jre/lib/security/cacerts
  ```

  If you want to import the CA certificate into another trustStore location, replace `${JAVA_HOME}/jre/lib/security/cacerts` with the `cacert` location that you want to use.

- **Self-signed certificate**:

  If you are using a self-signed digital certificate, you must import it into your Java `cacert` trustStore. For example, if you want to import the certificate to a Java `cacert` location of `/etc/pki/java/cacerts`, use the following command to import your self-signed certificate:

  ```
  keytool -import -trustcacerts -alias <MyHiveLdaps>
  -storepass <password> -noprompt -file <myCert>.pem -keystore /etc/pki/
  java/cacerts
  ```

4. If your trustStore is not `${JAVA_HOME}/jre/lib/security/cacerts`, you must set the HADOOP_OPTS environment variable to point to your CA certificate so that the certificate loads when the HDP platform loads.

   Note

   There is no need to modify the `hadoop-env` template if you use the default Java trustStore of `${JAVA_HOME}/jre/lib/security/cacerts`.

   To set this in Ambari:

   a. In the list of services on the left, click **HDFS**.

   b. Select the **Configs** tab.

   c. On the Configs tab page, select the **Advanced** tab.

   d. Scroll down, and expand the **Advanced hadoop-env** section.

   e. Add the following configuration information to the **hadoop-env template** text box:

   ```
   export HADOOP_OPTS="-Djava_net_preferIPv4Stack=true
   -Djavax.net.ssl.trustStore=/etc/pki/java/cacerts
   -Djavax.net.ssl.trustStorePassword=changeit ${HADOOP_OPTS}"
   ```

   f. Click **Save**.

5. Restart the HDFS and Hive services.

To restart these services in Ambari:

a. Click the service name on the left margin of the page.

b. On the service page, click **Service Actions**.

c. Choose **Restart All**.

For more information about restarting components in Ambari, see "Managing Services" in the *Ambari User's Guide*.

6. Test the LDAPS authentication. For example, if you are using the Beeline client, type the following commands at the Beeline prompt:

```
beeline>!connect jdbc:hive2://node1:10000/default
```

The Beeline client prompts for the user ID and password again. Enter those values to run the command.

Note

- Components such as Apache Knox and Apache Ranger do not use the `hadoop-env.sh.template`. The configuration files for these components must be set for LDAPS and manually restarted.
- Ambari Hive View does not work with LDAP or LDAPS.

1.7. Troubleshooting Hive

- **Error Related to Character Set Used for MySQL: "Specified key was too long; max key length is 767 bytes"**

  MySQL is the default database used by the Hive metastore. Depending on several factors, such as the version and configuration of MySQL, Hive developers may encounter an error message similar to the following:

```
An exception was thrown while adding/validating classes) : Specified key was
 too long; max key length is 767 bytes
```

  Administrators can resolve this issue by altering the Hive metastore database to use the Latin1 character set, as shown in the following example:

```
mysql> ALTER DATABASE <metastore_database_name> character set latin1;
```

- **Limitations When Using the `timestamp.formats` SerDe Parameter**

  The `timestamp.formats` SerDe parameter, introduced in HDP 2.3, produces the following behaviors:

  - Displays only 3 decimal digits when it returns values, but it accepts more decimal digits.

    For example, if you run the following commands:

```
drop table if exists src_hbase_ts;
```

```
create table src_hbase_ts( rowkey string, ts1 string, ts2 string, ts3
 string, ts4 string )
STORED BY 'org.apache.hadoop.hive. hbase. HBaseStorageHandler' WITH
 SERDEPROPERTIES
('hbase.columns.mapping' = 'm:ts1,m:ts2,m:ts3,m:ts4') TBLPROPERTIES
 ('hbase.table.name' = 'hbase_ts');

insert into src_hbase_ts values ('1','2011-01-01T01:01: 01.111111111',
 '2011-01-01T01:01: 01.123456111',
'2011-01-01T01:01: 01.111111111', '2011-01-01T01:01: 01.134567890');

drop table if exists hbase_ts_1;

create external table hbase_ts_1( rowkey string, ts1 timestamp, ts2
 timestamp, ts3 timestamp, ts4 timestamp )
STORED BY 'org.apache.hadoop.hive. hbase. HBaseStorageHandler' WITH
 SERDEPROPERTIES
( 'hbase.columns.mapping' = 'm:ts1,m:ts2,m:ts3,m:ts4', 'timestamp.formats'
 = "yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS")
TBLPROPERTIES ('hbase.table.name' = 'hbase_ts');

select * from hbase_ts_1;
```

The `timestamp.formats` parameter displays:

```
1 2011-01-01 01:01:01.111 2011-01-01 01:01:01.123 2011-01-01 01:01:01.111
 2011-01-01 01:01:01.134
```

When the expected output is:

```
1 2011-01-01 01:01:01.111111111 2011-01-01 01:01:01.123456111 2011-01-01
 01:01:01.111111111 2011-0
```

- The `yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS` format accepts any timestamp data up to `.SSSSSSSSS` decimal digits (9 places to the left of the decimal) instead of only reading data with `.SSSSSSSSS` decimal digits (9 places to the left of the decimal).

  For example, if you run the following commands:

```
drop table if exists src_hbase_ts; create table src_hbase_ts( rowkey
 string, ts1 string, ts2 string, ts3 string, ts4 string )
STORED BY 'org.apache.hadoop. hive. hbase.HBaseStorageHandler' WITH
 SERDEPROPERTIES
('hbase.columns.mapping' = 'm:ts1,m:ts2,m:ts3,m:ts4') TBLPROPERTIES
 ('hbase.table.name' = 'hbase_ts');

insert into src_hbase_ts values ('1','2011-01-01T01:01: 01.111111111',
 '2011-01-01T01:01: 01.111',
'2011-01-01T01:01: 01.11', '2011-01-01T01:01:01.1');

drop table if exists hbase_ts_1;

create external table hbase_ts_1( rowkey string, ts1 timestamp, ts2
 timestamp, ts3 timestamp, ts4 timestamp )
STORED BY 'org.apache.hadoop. hive. hbase.HBaseStorageHandler' WITH
 SERDEPROPERTIES
( 'hbase.columns.mapping' = 'm:ts1,m:ts2,m:ts3,m:ts4', 'timestamp.formats'
 = "yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSS")
TBLPROPERTIES ('hbase.table.name' = 'hbase_ts');
```

```
select * from hbase_ts_1;
```

The actual output is:

```
1 2011-01-01 01:01:01.111 2011-01-01 01:01:01.111 2011-01-01 01:01:01.11
 2011-01-01 01:01:01.1
```

When the expected output is:

```
1 2011-01-01 01:01:01.111 NULL NULL NULL
```

1.8. Hive JIRAs

Issue tracking for Hive bugs and improvements can be found on the Apache Hive site.

2. SQL Compliance

This section discusses the ongoing implementation of standard SQL syntax in Hive. Although SQL in Hive does not yet entirely support the SQL-2011 standard, versions 0.13 and 0.14 provide significant improvements to the parity between SQL as used in Hive and SQL as used in traditional relational databases.

2.1. INSERT ... VALUES, UPDATE, and DELETE SQL Statements

INSERT ... VALUES, UPDATE, and DELETE SQL statements are supported in Apache Hive 0.14 and later. The INSERT ... VALUES statement enable users to write data to Apache Hive from values provided in SQL statements. The UPDATE and DELETE statements enable users to modify and delete values already written to Hive. All three statements support auto-commit, which means that each statement is a separate transaction that is automatically committed after the SQL statement is executed.

The INSERT ... VALUES, UPDATE, and DELETE statements require the following property values in the hive-site.xml configuration file:

Configuration Property	Required Value
`hive.enforce.bucketing`	`true`
`hive.exec.dynamic.partition.mode`	`nonstrict`

Note

Administrators must use a transaction manager that supports ACID and the ORC file format to use transactions. See Hive Transactions for information about configuring other properties related to use ACID-based transactions.

INSERT ... VALUES Statement

The INSERT ... VALUES statement is revised to support adding multiple values into table columns directly from SQL statements. A valid INSERT ... VALUES statement must provide values for each column in the table. However, users may assign null values to columns

for which they do not want to assign a value. In addition, the PARTITION clause must be included in the DML.

```
INSERT INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]
 VALUES values_row [, values_row...]
```

In this syntax, `values_row` is `(value [, value])` and where `value` is either `NULL` or any SQL literal.

The following example SQL statements demonstrate several usage variations of this statement:

```
CREATE TABLE students (name VARCHAR(64), age INT, gpa DECIMAL(3,2)) CLUSTERED
 BY (age) INTO 2 BUCKETS STORED AS ORC;
```

```
INSERT INTO TABLE students VALUES ('fred flintstone', 35, 1.28), ('barney
 rubble', 32, 2.32);
```

```
CREATE TABLE pageviews (userid VARCHAR(64), link STRING, from STRING)
 PARTITIONED BY (datestamp STRING) CLUSTERED BY (userid) INTO 256 BUCKETS
 STORED AS ORC;
```

```
INSERT INTO TABLE pageviews PARTITION (datestamp = '2014-09-23') VALUES
 ('jsmith', 'mail.com', 'sports.com'), ('jdoe', 'mail.com', null);
```

```
INSERT INTO TABLE pageviews PARTITION (datestamp) VALUES ('tjohnson',
 'sports.com', 'finance.com', '2014-09-23'), ('tlee', 'finance.com', null,
 '2014-09-21');
```

UPDATE Statement

Use the UPDATE statement to modify data already written to Apache Hive. Depending on the condition specified in the optional `WHERE` clause, an UPDATE statement may affect every row in a table. You must have both the SELECT and UPDATE privileges to use this statement.

```
UPDATE tablename SET column = value [, column = value ...] [WHERE
 expression];
```

The UPDATE statement has the following limitations:

- The expression in the WHERE clause must be an expression supported by a Hive SELECT clause.
- Partition and bucket columns cannot be updated.
- Query vectorization is automatically disabled for UPDATE statements. However, updated tables can still be queried using vectorization.
- Subqueries are not allowed on the right side of the SET statement.

The following example demonstrates the correct usage of this statement:

```
UPDATE students SET name = null WHERE gpa <= 1.0;
```

DELETE Statement

Use the `DELETE` statement to delete data already written to Apache Hive.

```
DELETE FROM tablename [WHERE expression];
```

The DELETE statement has the following limitation: query vectorization is automatically disabled for the DELETE operation. However, tables with deleted data can still be queried using vectorization.

The following example demonstrates the correct usage of this statement:

```
DELETE FROM students WHERE gpa <= 1,0;
```

2.2. SQL Standard-based Authorization with GRANT And REVOKE SQL Statements

Secure SQL standard-based authorization using the GRANT and REVOKE SQL statements is supported in Hive 0.13 and later. Hive provides three authorization models: SQL standard-based authorization, storage-based authorization, and default Hive authorization.
In addition, Ranger provides centralized management of authorization for all HDP components. Use the following procedure to manually enable standard SQL authorization:

Note

This procedure is unnecessary if your Hive administrator installed Hive using Ambari.

1. Set the following configuration parameters in `hive-site.xml`:

Table 2.1. Configuration Parameters for Standard SQL Authorization

Configuration Parameter	Required Value
`hive.server2.enable.doAs`	`false`
`hive.users.in.admin.role`	Comma-separated list of users granted the administrator role.

2. Start HiveServer2 with the following command-line options:

Table 2.2. HiveServer2 Command-Line Options

Command-Line Option	Required Value
`-hiveconf hive.security.authorization.manager`	`org.apache.hadoop.hive.ql.security.authorization.MetaStoreAuthzAPIAuthorizerEmbedOnly`
`-hiveconf hive.security.authorization.enabled`	`true`
`-hiveconf hive.security.authenticator.manager`	`org.apache.hadoop.hive.ql.security.SessionStateUserAuthenticator`
`-hiveconf hive.metastore.uris`	`''`(a space inside single quotation marks)

Note

Administrators must also specify a storage-based authorization manager for Hadoop clusters that also use storage-based authorization. The `hive.security.authorization.manager` configuration property allows multiple authorization managers in comma-delimited format, so the correct value in this case is:

```
hive.security.authorization.manager=org.apache.hadoop.hive.ql.
security.authorization.StorageBasedAuthorizationProvider,

org.apache.hadoop.hive.ql.security.authorization.
MetaStoreAuthzAPIAuthorizerEmbedOnly
```

2.3. Transactions

Support for transactions in Hive 0.13 and later enables SQL atomicity of operations at the row level rather than at the level of a table or partition. This allows a Hive client to read from a partition at the same time that another Hive client is adding rows to the same partition. In addition, transactions provide a mechanism for streaming clients to rapidly update Hive tables and partitions. Hive transactions differ from RDBMS transactions in that each transaction has an identifier, and multiple transactions are grouped into a single transaction batch. A streaming client requests a set of transaction IDs after connecting to Hive and subsequently uses these transaction IDs one at a time during the initialization of new transaction batches. Clients write one or more records for each transaction and either commit or abort a transaction before moving to the next transaction.

Important

When Hive is configured to use an Oracle database and transactions are enabled in Hive, queries might fail with the error `org.apache.hadoop.hive.ql.lockmgr.LockException: No record of lock could be found, may have timed out`. This can be caused by a bug in the BoneCP connection pooling library. In this case, Hortonworks recommends that you set the `datanucleus.connectionPoolingType` property to `dbcp` so the DBCP library is used.

ACID is an acronym for four required traits of database transactions: atomicity, consistency, isolation, and durability.

Transaction Attribute	Description
Atomicity	An operation either succeeds completely or fails; it does not leave partial data.
Consistency	Once an application performs an operation, the results of that operation are visible to the application in every subsequent operation.
Isolation	Operations by one user do not cause unexpected side effects for other users.
Durability	Once an operation is complete, it is preserved in case of machine or system failure.

By default, transactions are disabled in Hive. To use ACID-based transactions, administrators must use a transaction manager that supports ACID and the ORC file format. See Configuring the Hive Transaction Manager [36] later in this section for instructions on configuring a transaction manager for Hive.

Note

See the Hive wiki for more information about Hive's support of ACID semantics for transactions.

Understanding Compactions

Hive stores data in base files that cannot be updated by HDFS. Instead, Hive creates a set of delta files for each transaction that alters a table or partition and stores them in a separate delta directory. Occasionally, Hive *compacts*, or merges, the base and delta files. Hive performs all compactions in the background without affecting concurrent reads and writes of Hive clients. There are two types of compactions:

Table 2.3. Hive Compaction Types

Compaction Type	Description
Minor	Rewrites a set of delta files to a single delta file for a bucket.
Major	Rewrites one or more delta files and the base file as a new base file for a bucket.

By default, Hive automatically compacts delta and base files at regular intervals. However, Hadoop administrators can configure automatic compactions, as well as perform manual compactions of base and delta files using the following configuration parameters in `hive-site.xml`.

Table 2.4. Hive Transaction Configuration Parameters

Configuration Parameter	Description
`hive.txn.manager`	Specifies the class name of the transaction manager used by Hive. Set this property to `org.apache.hadoop.hive.ql.lockmgr.DbTxnManager` to enable transactions. The default value is `org.apache.hadoop.hive.ql.lockmgr.DummyTxnManager`, which disables transactions.
`hive.compactor.initiator.on`	Specifies whether to run the initiator and cleaner threads on this Metastore instance. The default value is false. Must be set to true for exactly one instance of the Hive metastore service.
`hive.compactor.worker.threads`	Specifies the number of of worker threads to run on this Metastore instance. The default value is 0, which must be set to greater than 0 to enable compactions. Worker threads initialize MapReduce jobs to do compactions. Increasing the number of worker threads decreases the time required to compact tables after they cross a threshold that triggers compactions. However, increasing the number of worker threads also increases the background load on a Hadoop cluster.
`hive.compactor.worker.timeout`	Specifies the time period, in seconds, after which a compaction job is failed and re-queued. The default value is 86400 seconds, or 24 hours.
`hive.compactor.check.interval`	Specifies the time period, in seconds, between checks to see if any partitions require compacting. The default value is 300 seconds. Decreasing this value reduces the time required to start a compaction for a table or partition. However, it also increases the background load on the NameNode since each check requires several calls to the NameNode.
`hive.compactor.delta.num.threshold`	Specifies the number of delta directories in a partition that triggers an automatic minor compaction. The default value is 10.
`hive.compactor.delta.pct.threshold`	Specifies the percentage size of delta files relative to the corresponding base files that triggers an automatic major compaction. The default value is.1, which is 10 percent.

Configuration Parameter	Description
hive.compactor.abortedtxn.threshold	Specifies the number of aborted transactions on a single partition that trigger an automatic major compaction.

Configuring the Hive Transaction Manager

Configure the following Hive properties to enable transactions:

- hive.txn.manager
- hive.compactor.initiator.on
- hive.compactor.worker.threads

Tip

To disable automatic compactions for individual tables, set the NO_AUTO_COMPACTION table property for those tables. This overrides the configuration settings in hive-site.xml. However, this property does not prevent manual compactions.

If you experience problems while enabling Hive transactions, check the Hive log file at /tmp/hive/hive.log.

Performing Manual Compactions

Hive administrators use the ALTER TABLE DDL command to queue requests that compact base and delta files for a table or partition:

```
ALTER TABLE tablename [PARTITION (partition_key='partition_value' [,...])]
 COMPACT 'compaction_type'
```

Use the SHOW COMPACTIONS command to monitor the progress of the compactions:

```
SHOW COMPACTIONS
```

Note

ALTER TABLE will compact tables even if the NO_AUTO_COMPACTION table property is set.

The SHOW COMPACTIONS command provides the following output for each compaction:

- Database name
- Table name
- Partition name
- Major or minor compaction
- Compaction state:
  - Initiated - waiting in queue
  - Working - currently compacting

    - Ready for cleaning - compaction completed and old files scheduled for removal
- Thread ID
- Start time of compaction

Hive administrators can also view a list of currently open and aborted transactions with the the `SHOW TRANSACTIONS` command. This command provides the following output for each transaction:

- Transaction ID
- Transaction state
- Hive user who initiated the transaction
- Host machine where transaction was initiated

Lock Manager

`DbLockManager`, introduced in Hive 0.13, stores all transaction and related lock information in the Hive Metastore. Heartbeats are sent regularly from lock holders and transaction initiators to the Hive metastore to prevent stale locks and transactions. The lock or transaction is aborted if the metastore does not receive a heartbeat within the amount of time specified by the `hive.txn.timeout` configuration property. Hive administrators use the `SHOW LOCKS` DDL command to view information about locks associated with transactions.

This command provides the following output for each lock:

- Database name
- Table name
- Partition, if the table is partitioned
- Lock state:
    - Acquired - transaction initiator hold the lock
    - Waiting - transaction initiator is waiting for the lock
    - Aborted - the lock has timed out but has not yet been cleaned
- Lock type:
    - Exclusive - the lock may not be shared
    - Shared_read - the lock may be shared with any number of other shared_read locks
    - Shared_write - the lock may be shared by any number of other shared_read locks but not with other shared_write locks
- Transaction ID associated with the lock, if one exists

- Last time lock holder sent a heartbeat
- Time the lock was acquired, if it has been acquired
- Hive user who requested the lock
- Host machine on which the Hive user is running a Hive client

Note

The output of the command reverts to behavior prior to Hive 0.13 if administrators use ZooKeeper or in-memory lock managers.

Transaction Limitations

HDP currently has the following limitations for ACID-based transactions in Hive:

- The BEGIN, COMMIT, and ROLLBACK SQL statements are not yet supported. All operations are automatically committed as transactions.
- The user initiating the Hive session must have write permission for the destination partition or table.
- ZooKeeper and in-memory locks are not compatible with transactions.
- Only ORC files are supported.
- Destination tables must be bucketed and not sorted.
- The only supported isolation level is Snapshot.

2.4. Subqueries

Hive supports subqueries in FROM clauses and in WHERE clauses of SQL statements. A subquery is a SQL expression that is evaluated and returns a result set. Then that result set is used to evaluate the parent query. The parent query is the outer query that contains the child subquery. Subqueries in WHERE clauses are supported in Hive 0.13 and later. The following example shows a subquery inserted into a WHERE clause:

```
SELECT state, net_payments
FROM transfer_payments
WHERE transfer_payments.year IN (SELECT year FROM us_census);
```

No configuration is required to enable execution of subqueries in Hive. The feature is available by default. However, several restrictions exist for the use of subqueries in WHERE clauses.

Understanding Subqueries in SQL

SQL adheres to syntax rules like any programming language. The syntax governing the use of subqueries in WHERE clauses in SQL depends on the following concepts:

- **Query Predicates and Predicate Operators**

A *predicate* in SQL is a condition that evaluates to a Boolean value. For example, the predicate in the preceeding example returns true for a row of the `transfer_payments` table if at least one row exists in the `us_census` table with the same year as the `transfer_payments` row. The predicate starts with the first WHERE keyword.

```
... WHERE transfer_payments.year IN (SELECT year FROM us_census);
```

A SQL predicate in a subquery must also contain a predicate operator. *Predicate operators* specify the relationship tested in a predicate query. For example, the predicate operator in the above example is the IN keyword.

- **Aggregrated and Correlated Queries**

Aggregated queries combine one or more aggregate functions, such as AVG, SUM, and MAX, with the GROUP BY statement to group query results by one or more table columns. In the following example, the AVG aggregate function returns the average salary of all employees in the engineering department grouped by year:

```
SELECT year, AVG(salary)
FROM Employees
WHERE department = 'engineering' GROUP BY year
```

Note

The GROUP BY statement may be either explicit or implicit.

Correlated queries contain a query predicate with the equals (=) operator. One side of the operator must reference at least one column from the parent query and the other side must reference at least one column from the subquery. The following query is a revised and correlated version of the example query that is shown at the beginning of this section. It is a correlated query because one side of the equals predicate operator in the subquery references the `state` column in the `transfer_payments` table in the parent query and the other side of the operator references the `state` column in the `us_census` table.

```
SELECT state, net_payments
FROM transfer_payments
WHERE EXISTS
   (SELECT year
   FROM us_census
   WHERE transfer_payments.state = us_census.state);
```

In contrast, an *uncorrelated query* does not reference any columns in the parent query.

- **Conjuncts and Disjuncts**

A *conjunct* is equivalent to the AND condition, while a *disjunct* is the equivalent of the OR condition. The following subquery contains a conjunct:

```
... WHERE transfer_payments.year = "2010" AND us_census.state = "california"
```

The following subquery contains a disjunct:

```
... WHERE transfer_payments.year = "2010" OR us_census.state = "california"
```

Restrictions on Subqueries in WHERE Clauses

Subqueries in WHERE clauses have the following limitations:

- Subqueries must appear on the right hand side of an expression.
- Nested subqueries are not supported.
- Only one subquery expression is allowed for a single query.
- Subquery predicates must appear as top level conjuncts.
- Subqueries support four logical operators in query predicates: IN, NOT IN, EXISTS, and NOT EXISTS.
- The IN and NOT IN logical operators may select only one column in a WHERE clause subquery.
- The EXISTS and NOT EXISTS operators must have at least one correlated predicate.
- The left side of a subquery must qualify all references to table columns.
- References to columns in the parent query are allowed only in the WHERE clause of the subquery.
- Subquery predicates that reference a column in a parent query must use the equals (=) predicate operator.
- Subquery predicates may not refer only to columns in the parent query.
- Correlated subqueries with an implied GROUP BY statement may return only one row.
- All unqualified references to columns in a subquery must resolve to tables in the subquery.
- Correlated subqueries cannot contain windowing clauses.

2.5. Common Table Expressions

A common table expression (CTE) is a set of query results obtained from a simple query specified within a WITH clause and which immediately preceeds a SELECT or INSERT keyword. A CTE exists only within the scope of a single SQL statement. One or more CTEs can be used with the following SQL statements:

- SELECT
- INSERT
- CREATE TABLE AS SELECT
- CREATE VIEW AS SELECT

The following example demonstrates the use of `q1` as a CTE in a SELECT statement:

```
WITH q1 AS (SELECT key from src where key = '5')
```

```
   SELECT * from q1;
```

The following example demonstrates the use of `q1` as a CTE in an INSERT statement:

```
CREATE TABLE s1 LIKE src;
   WITH q1 AS (SELECT key, value FROM src WHERE key = '5')
   FROM q1 INSERT OVERWRITE TABLE s1 SELECT *;
```

The following example demonstrates the use of `ql` as a CTE in a CREATE TABLE AS SELECT clause:

```
CREATE TABLE s2 AS WITH q1 AS (SELECT key FROM src WHERE key = '4')
   SELECT * FROM q1;
```

The following example demonstrates the use of `q1` as a CTE in a CREATE TABLE AS VIEW clause:

```
CREATE VIEW v1 AS WITH q1 AS (SELECT key FROM src WHERE key='5')
   SELECT * from q1;
```

CTEs are available by default in Hive 0.13. Hive administrators do not need to perform any configuration to enable them.

Limitations of Common Table Expressions

- Recursive queries are not supported.
- The WITH clause is not supported within subquery blocks.

2.6. Quoted Identifiers in Column Names

Quoted identifiers in the names of table columns are supported in Hive 0.13 and later. An *identifier* in SQL is a sequence of alphanumeric and underscore (_) characters surrounded by backtick (`) characters. Quoted identifiers in Hive are case-insensitive. In the following example, `` `x+y` `` and `` `a?b` `` are valid column names for a new table.

```
CREATE TABLE test (`x+y` String, `a?b` String);
```

Quoted identifiers can be used anywhere a column name is expected, including table partitions and buckets:

```
CREATE TABLE partition_date-1 (key string, value string)
PARTITIONED BY (`dt+x` date, region int);

CREATE TABLE bucket_test(`key?1` string, value string)
CLUSTERED BY (`key?1`) into 5 buckets;
```

Note

Use a backtick character to escape a backtick character (``).

Enabling Quoted Identifiers

Set the `hive.support.quoted.identifiers` configuration parameter to `column` in hive-site.xml to enable quoted identifiers in SQL column names. For Hive 0.13, the valid values are `none` and `column`.

```
hive.support.quoted.identifiers = column
```

2.7. CHAR Data Type Support

Hive 0.13 supports The CHAR data type is supported in Hive 0.13 and later. This data type simplifies the process of migrating data from other databases. Hive ignores trailing whitespace characters for the CHAR data type. However, there is no consensus among database vendors on the handling of trailing whitespaces. Before you perform a data migration to Hive, consult the following table to avoid unexpected behavior with values for CHAR, VARCHAR, and STRING data types.

The following table describes how several types of databases treat trailing whitespaces for the CHAR, VARCHAR, and STRING data types:

Table 2.5. Trailing Whitespace Characters on Various Databases

Data Type	Hive	Oracle	SQL Server	MySQL	Teradata
CHAR	Ignore	Ignore	Ignore	Ignore	Ignore
VARCHAR	Compare	Compare	Configurable	Ignore	Ignore
STRING	Compare	N/A	N/A	N/A	N/A

3. Running Pig with the Tez Execution Engine

By default, Apache Pig runs against Apache MapReduce, but administrators and scripters can configure Pig to run against the Apache Tez execution engine to take advantage of more efficient execution and fewer reads of HDFS. Pig supports Tez in all of the following ways:

Command Line	Use the -x command-line option: `pig -x tez`
Pig Properties	Set the following configuration property in the conf/pig.properties file: `exectype=tez`
Java Option	Set the following Java Option for Pig: `PIG_OPTS="-D exectype=tez"`

Users and administrators can use the same methods to configure Pig to run against the default MapReduce execution engine.

Command Line	Use the `-x` command-line option: `pig -x mr`
Pig Properties	Set the following configuration property in the conf/pig.properties file: `exectype=tez`
Java Option	Set the following Java Option for Pig: `PIG_OPTS="-D exectype=tez"`
Pig Script	Use the `set` command: `set exectype=mr;`

There are some limitations to running Pig with the Tez execution engine:

- Queries that include the ORDER BY clause may run slower than if run against the MapReduce execution engine.
- There is currently no user interface that allows users to view the execution plan for Pig jobs running with Tez. To diagnose a failing Pig job, users must read the Application Master and container logs.

Note

Users should configure parallelism before running Pig with Tez. If parallelism is too low, Pig jobs will run slowly. To tune parallelism, add the PARALLEL clause to your PIG statements.

Running a Pig-on-Tez Job with Oozie

To run a Pig job on Tez using Oozie, perform the following configurations:

- Add the following property and value to the `job.properties` file for the Pig-on-Tez Oozie job:

```
<property>
     <name>oozie.action.sharelib.for.pig</name>
     <value>pig, hive</value>
</property>
```

- Create the `$OOZIE_HOME/conf/action-conf/pig` directory and copy the `tez-site.xml` file into it.

4. Using HDP for Metadata Services (HCatalog)

Hortonworks Data Platform (HDP) deploys Apache HCatalog to manage the metadata services for your Hadoop cluster.

Apache HCatalog is a table and storage management service for data created using Apache Hadoop. This includes:

- Providing a shared schema and data type mechanism.
- Providing a table abstraction so that users need not be concerned with where or how their data is stored.
- Providing interoperability across data processing tools such as Pig, MapReduce, and Hive.

Start the HCatalog CLI with the following command:

```
<hadoop-install-dir>\hcatalog-0.5.0\bin\hcat.cmd
```

Note

HCatalog 0.5.0 was the final version released from the Apache Incubator. In March 2013, HCatalog graduated from the Apache Incubator and became part of the Apache Hive project. New releases of Hive include HCatalog, starting with Hive 0.11.0.

HCatalog includes two documentation sets:

1. General information about HCatalog

   This documentation covers installation and user features. The next section, Using HCatalog, provides links to individual documents in the HCatalog documentation set.

2. WebHCat information

   WebHCat is a web API for HCatalog and related Hadoop components. The section Using WebHCat provides links to user and reference documents, and includes a technical update about standard WebHCat parameters.

For more details on the Apache Hive project, including HCatalog and WebHCat, see "Using Apache Hive" and the following resources:

- Hive project home page
- Hive wiki home page
- Hive mailing lists

4.1. Using HCatalog

For details about HCatalog, see the following resources in the HCatalog documentation set:

- HCatalog Overview
- Installation From Tarball
- HCatalog Configuration Properties
- Load and Store Interfaces
- Input and Output Interfaces
- Reader and Writer Interfaces
- Command Line Interface
- Storage Formats
- Dynamic Partitioning
- Notification
- Storage Based Authorization

4.2. Using WebHCat

WebHCat provides a REST API for HCatalog and related Hadoop components.

Note

WebHCat was originally named Templeton, and both terms may still be used interchangeably. For backward compatibility the Templeton name still appears in URLs and log file names.

For details about WebHCat (Templeton), see the following resources:

- Overview
- Installation
- Configuration
- Reference
  - Resource List
  - GET :version
  - GET status
  - GET version
  - DDL Resources: Summary and Commands
  - POST mapreduce/streaming
  - POST mapreduce/jar

- POST pig
- POST hive
- GET queue/:jobid
- DELETE queue/:jobid

4.3. Security for WebHCat

WebHCat currently supports two types of security:

- Default security (without additional authentication)
- Authentication by using Kerberos

Example: HTTP GET :table

The following example demonstrates how to specify the `user.name` parameter in an HTTP GET request:

```
% curl -s 'http://localhost:50111/templeton/v1/ddl/database/default/table/
my_table?user.name=ctdean'
```

Example: HTTP POST :table

The following example demonstrates how to specify the `user.name` parameter in an HTTP POST request

```
% curl -s -d user.name=ctdean \
       -d rename=test_table_2 \
       'http://localhost:50111/templeton/v1/ddl/database/default/table/
          test_table'
```

Security Error

If the `user.name` parameter is not supplied when required, the following security error is returned:

```
{
   "error": "No user found. Missing user.name parameter."
}
```

5. Using Apache HBase and Apache Phoenix

Hortonworks Data Platform (HDP) deploys Apache HBase as a NoSQL database for your Hadoop cluster. HBase scales linearly to handle very large (petabyte scale), column-oriented data sets. The data store is predicated on a key-value model that supports low latency reads, writes, and updates in a distributed environment.

As a natively non-relational database, HBase can combine data sources that use a wide variety of different structures and schemas. It is natively integrated with HDFS for resilient data storage and is designed for hosting very large tables with sparse data.

HDP support also includes Apache Phoenix, a SQL abstraction layer for interacting with HBase. Phoenix lets you create and interact with tables in the form of typical DDL/DML statements via its standard JDBC API. For more information, see the Apache Phoenix website.

Supported JDBC client drivers can be obtained from the `/usr/hdp/current/phoenix-client/phoenix-client.jar` file on one of your cluster's edge nodes or in the Hortonworks Phoenix server-client repository . If you use the repository, download the JAR file corresponding to your installed HDP version.

5.1. HBase Installation and Setup

You can install and configure HBase for your HDP cluster by either of the following methods:

- Ambari Install Wizard: The wizard is the part of the Ambari web-based platform that guides HDP installation, including deploying the various Hadoop components such as HBase for the needs of your cluster. See the Ambari Install Guide.
- Manual Installation: You can fetch one of the repositories bundled with HBase and install it on the command line. See the Non-Ambari Installation Guide.

5.2. Enabling Phoenix

To enable Phoenix:

1. Open Ambari.
2. Select *Services* tab > *HBase* > *Configs* tab.
3. Scroll down to the Phoenix SQL settings.
4. (Optional) Reset the Phoenix Query Timeout.
5. Click the *Enable Phoenix* slider button.

5.3. Cell-level Access Control Lists (ACLs)

Cell-level access control lists for HBase tables are supported in HBase 0.98 and later.

Note

This feature is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact support by logging a case on the Hortonworks Support Portal.

5.4. Column Family Encryption

Column family encryption is supported in HBase 0.98 and later.

Note

This feature is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact support by logging a case on the Hortonworks Support Portal.

5.5. Tuning RegionServers

To tune garbage collection (GC) in HBase RegionServers for stability, make the following configuration changes:

1. Specify the following configurations in the HBASE_REGIONSERVER_OPTS configuration option in the `/conf/hbase-env.sh` file:

```
-XX:+UseConcMarkSweepGC
-Xmn2500m (depends on MAX HEAP SIZE, but should not be less than 1g and more
 than 4g)
-XX:PermSize=128m
-XX:MaxPermSize=128m
-XX:SurvivorRatio=4
-XX:CMSInitiatingOccupancyFraction=50
-XX:+UseCMSInitiatingOccupancyOnly
-XX:ErrorFile=/var/log/hbase/hs_err_pid%p.log
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
```

2. Make sure that the block cache size and the memstore size combined do not significantly exceed `0.5*MAX_HEAP`, which is defined in the HBASE_HEAP_SIZE configuration option of the `/conf/hbase-env.sh` file.

6. Using HDP for Workflow and Scheduling (Oozie)

Hortonworks Data Platform deploys Apache Oozie for your Hadoop cluster.

Oozie is a server-based workflow engine specialized in running workflow jobs with actions that execute Hadoop jobs, such as MapReduce, Pig, Hive, Sqoop, HDFS operations, and sub-workflows. Oozie supports coordinator jobs, which are sequences of workflow jobs that are created at a given frequency and start when all of the required input data is available.

A command-line client and a browser interface allow you to manage and administer Oozie jobs locally or remotely.

After installing an HDP 2.x cluster by using Ambari, access the Oozie web UI at the following URL:

```
http://{your.ambari.server.hostname}:11000/oozie
```

For additional Oozie documentation, use the following resources:

- Quick Start Guide
- Developer Documentation
  - Oozie Workflow Overview
  - Running the Examples
  - Workflow Functional Specification
  - Coordinator Functional Specification
  - Bundle Functional Specification
  - EL Expression Language Quick Reference
  - Command Line Tool
  - Workflow Rerun
  - Email Action
  - Writing a Custom Action Executor
  - Oozie Client Javadocs
  - Oozie Core Javadocs
  - Oozie Web Services API
- Administrator Documentation
  - Oozie Installation and Configuration

- Oozie Monitoring
- Command Line Tool

7. Using Apache Sqoop

Hortonworks Data Platform deploys Apache Sqoop for your Hadoop cluster. Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle into the Hadoop Distributed File System (HDFS), transform the data in Hadoop MapReduce, and then export the data back into an RDBMS. Sqoop automates most of this process, relying on the database to describe the schema for the data to be imported. Sqoop uses MapReduce to import and export the data, which provides parallel operation as well as fault tolerance.

For additional information see the Sqoop documentation, including these sections in the Sqoop User Guide:

- Basic Usage
- Sqoop Tools
- Troubleshooting

7.1. Apache Sqoop Connectors

Sqoop uses a connector-based architecture which supports plugins that provide connectivity to external systems. Using specialized connectors, Sqoop can connect with external systems that have optimized import and export facilities, or do not support native JDBC. Connectors are plugin components based on Sqoop's extension framework and can be added to any existing Sqoop installation.

Hortonworks provides the following connectors for Sqoop in the HDP 2 distribution:

- **MySQL connector:** Instructions for using this connector are available here.
- **Netezza connector:** See here and below for more information.
- **Oracle JDBC connector:** Instructions for using this connector are available here.
- **PostgreSQL connector:** Instructions for using this connector are here.
- **Microsoft SQL Server connector:** Instructions for using this connector are here.

A Sqoop connector for Teradata is available from the Hortonworks Add-ons page:

- **Teradata connector:** The connector and its documentation can be downloaded from here.

7.2. Sqoop Import Table Commands

When connecting to an Oracle database, the Sqoop import command requires case-sensitive table names and usernames (typically uppercase). Otherwise the import fails with error message "Attempted to generate class with no columns!"

Prior to the resolution of the issue SQOOP-741, import-all-tables would fail for an Oracle database. See the JIRA for more information.

The import-all-tables command has additional restrictions. See Chapter 8 in the Sqoop User Guide.

7.3. Netezza Connector

Netezza connector for Sqoop is an implementation of the Sqoop connector interfaces for accessing a Netezza data warehouse appliance, so that data can be exported and imported to a Hadoop environment from Netezza data warehousing environments.

The HDP 2 Sqoop distribution includes Netezza connector software. To deploy it, the only requirement is that you acquire the JDBC jar file (named `nzjdbc.jar`) from IBM and copy it to the `/usr/local/nz/lib` directory.

Extra Arguments

The following table describes extra arguments supported by the Netezza connector.

Note

All non-Sqoop arguments must be preceded by double dashes (–) to work correctly.

Argument	Description
`--partitioned-access`	Whether each mapper acts on a subset of data slices of a table or all.
`-max-errors`	Applicable only in direct mode. This option specifies the error threshold per mapper while transferring data. If the number of errors encountered exceeds this threshold, the job fails.
`--log-dir`	Applicable only in direct mode. Specifies the directory where Netezza external table operation logs are stored

Direct Mode

Netezza connector supports an optimized data transfer facility using the Netezza external tables feature. Each map task of Netezza connector's import job works on a subset of the Netezza partitions and transparently creates and uses an external table to transport data.

Similarly, export jobs use the external table to push data quickly onto the NZ system. Direct mode does not support staging tables and upsert options.

Direct mode is specified by the `--direct` Sqoop option.

Here is an example of a complete command line for import using the Netezza external table feature:

```
$ sqoop import \
 --direct \
 --connect jdbc:netezza://nzhost:5480/sqoop \
 --table nztable \
 --username nzuser \
 --password nzpass \
```

```
--target-dir hdfsdir \
-- --log-dir /tmp
```

Here is an example of a complete command line for export with tab (\t) as the field terminator character:

```
$ sqoop export \
 --direct \
 --connect jdbc:netezza://nzhost:5480/sqoop \
 --table nztable \
 --username nzuser \
 --password nzpass \
 --export-dir hdfsdir \
 --input-fields-terminated-by "\t"
```

Null String Handling

In direct mode the Netezza connector supports the null-string features of Sqoop. Null string values are converted to appropriate external table options during export and import operations.

Argument	Description
`--input-null-non-string <null-string>`	The string to be interpreted as null for non-string columns.
`--input-null-non-string <null-string>`	The string to be interpreted as null for non-string columns.

In direct mode, both the arguments must either be left to the default values or explicitly set to the same value. The null string value is restricted to 0-4 UTF-8 characters.

On export, for non-string columns, if the chosen null value is a valid representation in the column domain, then the column might not be loaded as null. For example, if the null string value is specified as "1", then on export, any occurrence of "1" in the input file will be loaded as value 1 instead of NULL for int columns.

For performance and consistency, specify the null value as an empty string.

Supported Import Control Arguments

Argument	Description
`--null-string <null-string>`	The string to be interpreted as null for string columns
`--null-non-string <null-string>`	The string to be interpreted as null for non-string columns.

In direct mode, both the arguments must either be left to the default values or explicitly set to the same value. The null string value is restricted to 0-4 UTF-8 characters.

On import, for non-string columns in the current implementation, the chosen null value representation is ignored for non-character columns. For example, if the null string value is specified as "\N", then on import, any occurrence of NULL for non-char columns in the table will be imported as an empty string instead of \N, the chosen null string representation.

For performance and consistency, specify the null value as an empty string.

7.4. Sqoop-HCatalog Integration

This section describes the interaction between HCatalog with Sqoop.

HCatalog is a table and storage management service for Hadoop that enables users with different data processing tools – Pig, MapReduce, and Hive – to more easily read and write data on the grid. HCatalog's table abstraction presents users with a relational view of data in the Hadoop distributed file system (HDFS) and ensures that users need not worry about where or in what format their data is stored: RCFile format, text files, or SequenceFiles.

HCatalog supports reading and writing files in any format for which a Hive SerDe (serializer-deserializer) has been written. By default, HCatalog supports RCFile, CSV, JSON, and SequenceFile formats. To use a custom format, you must provide the InputFormat and OutputFormat as well as the SerDe.

The ability of HCatalog to abstract various storage formats is used in providing RCFile (and future file types) support to Sqoop.

Exposing HCatalog Tables to Sqoop

HCatalog interaction with Sqoop is patterned on an existing feature set that supports Avro and Hive tables. This section introduces five command line options. Some command line options defined for Hive are reused.

Relevant Command-Line Options

Command-line Option	Description
`--hcatalog-database`	Specifies the database name for the HCatalog table. If not specified, the default database name 'default' is used. Providing the –hcatalog-database option without –hcatalog-table is an error. This is not a required option.
`-hcatalog-table`	The argument value for this option is the HCatalog tablename. The presence of the –hcatalog-table option signifies that the import or export job is done using HCatalog tables, and it is a required option for HCatalog jobs.
`--hcatalog-home`	The home directory for the HCatalog installation. The directory is expected to have a lib subdirectory and a share/hcatalog subdirectory with necessary HCatalog libraries. If not specified, the system environment variable HCAT_HOME will be checked and failing that, a system property hcatalog.home will be checked. If none of these are set, the default value will be used and currently the default is set to /usr/lib/hcatalog. This is not a required option.
`--create-hcatalog-table`	This option specifies whether an HCatalog table should be created automatically when importing data. By default, HCatalog tables are assumed to exist. The table name will

Command-line Option	Description
	be the same as the database table name translated to lower case. Further described in Automatic Table Creation.
`--hcatalog-storage- stanza`	This option specifies the storage stanza to be appended to the table. Further described in Automatic Table Creation.

Supported Sqoop Hive Options

The following Sqoop options are also used along with the –hcatalog-table option to provide additional input to the HCatalog jobs. Some of the existing Hive import job options are reused with HCatalog jobs instead of creating HCatalog-specific options for the same purpose.

Command-line Option	Description
`--map-column-hive`	This option maps a database column to HCatalog with a specific HCatalog type.
`--hive-home`	The Hive home location.
`--hive-partition-key`	Used for static partitioning filter. The partitioning key should be of type STRING. There can be only one static partitioning key.
`--hive-partition-value`	The value associated with the partition.

Direct Mode Support

HCatalog integration in Sqoop has been enhanced to support direct mode connectors. Direct mode connectors are high performance connectors specific to a database. The Netezza direct mode connector is enhanced to use this feature for HCatalog jobs.

Important

Only the Netezza direct mode connector is currently enabled to work with HCatalog.

Unsupported Sqoop Hive Import Options

Sqoop Hive options that are not supported with HCatalog jobs:

- `--hive-import`
- `--hive-overwrite`

In addition, the following Sqoop export and import options are not supported with HCatalog jobs:

- `--direct`
- `--export-dir`
- `--target-dir`
- `--warehouse-dir`
- `--append`
- `--as-sequencefile`

- `--as-avrofile`

Ignored Sqoop Options

All input delimiter options are ignored.

Output delimiters are generally ignored unless either `--hive-drop-import-delims` or `--hive-delims-replacement` is used. When the `--hive-drop-import-delims` or `--hive-delims-replacement` option is specified, all database columns of type CHAR are post-processed to either remove or replace the delimiters, respectively. (See Delimited Text Formats and Field and Line Delimiter Characters.) This is only needed if the HCatalog table uses text format.

7.5. Controlling Transaction Isolation

Sqoop uses read-committed transaction isolation in its mappers to import data. However, this may not be ideal for all ETL workflows, and you might want to reduce the isolation guarantees. Use the `--relaxed-isolation` option to instruct Sqoop to use read-uncommitted isolation level.

The read-uncommitted transaction isolation level is not supported on all databases, such as Oracle. Specifying the `--relaxed-isolation` may also not be supported on all databases.

Note

There is no guarantee that two identical and subsequent uncommitted reads will return the same data.

7.6. Automatic Table Creation

One of the key features of Sqoop is to manage and create the table metadata when importing into Hadoop. HCatalog import jobs also provide for this feature with the option `--create-hcatalog-table`. Furthermore, one of the important benefits of the HCatalog integration is to provide storage agnosticism to Sqoop data movement jobs. To provide for that feature, HCatalog import jobs provide an option that lets a user specifiy the storage format for the created table.

The option `--create-hcatalog-table` is used as an indicator that a table has to be created as part of the HCatalog import job.

The option `--hcatalog-storage-stanza` can be used to specify the storage format of the newly created table. The default value for this option is "stored as rcfile". The value specified for this option is assumed to be a valid Hive storage format expression. It will be appended to the CREATE TABLE command generated by the HCatalog import job as part of automatic table creation. Any error in the storage stanza will cause the table creation to fail and the import job will be aborted.

Any additional resources needed to support the storage format referenced in the option `--hcatalog-storage-stanza` should be provided to the job either by placing them in `$HIVE_HOME/lib` or by providing them in `HADOOP_CLASSPATH` and `LIBJAR` files.

If the option `--hive-partition-key` is specified, then the value of this option is used as the partitioning key for the newly created table. Only one partitioning key can be specified with this option.

Object names are mapped to the lowercase equivalents as specified below when mapped to an HCatalog table. This includes the table name (which is the same as the external store table name converted to lower case) and field names.

7.7. Delimited Text Formats and Field and Line Delimiter Characters

HCatalog supports delimited text format as one of the table storage formats. But when delimited text is used and the imported data has fields that contain those delimiters, then the data may be parsed into a different number of fields and records by Hive, thereby losing data fidelity.

For this case, one of these existing Sqoop import options can be used:

- `--hive-delims-replacement`
- `--hive-drop-import-delims`

If either of these options is provided on input, then any column of type `STRING` will be formatted with the Hive delimiter processing and then written to the HCatalog table.

7.8. HCatalog Table Requirements

The HCatalog table should be created before using it as part of a Sqoop job if the default table creation options (with optional storage stanza) are not sufficient. All storage formats supported by HCatalog can be used with the creation of the HCatalog tables. This makes this feature readily adopt new storage formats that come into the Hive project, such as ORC files.

7.9. Support for Partitioning

The Sqoop HCatalog feature supports the following table types:

- Unpartitioned tables
- Partitioned tables with a static partitioning key specified
- Partitioned tables with dynamic partition keys from the database result set
- Partitioned tables with a combination of a static key and additional dynamic partitioning keys

7.10. Schema Mapping

Sqoop currently does not support column name mapping. However, the user is allowed to override the type mapping. Type mapping loosely follows the Hive type mapping already

present in Sqoop except that the SQL types FLOAT and REAL are mapped to the HCatalog type "float." In the Sqoop type mapping for Hive, these two SQL types are mapped to "double." Type mapping is primarily used for checking the column definition correctness only and can be overridden with the `--map-column-hive` option.

All types except binary are assignable to a string type.

Any field of number type (int, shortint, tinyint, bigint and bigdecimal, float and double) is assignable to another field of any number type during exports and imports. Depending on the precision and scale of the target type of assignment, truncations can occur.

Furthermore, date/time/timestamps are mapped to string (the full date/time/timestamp representation) or bigint (the number of milliseconds since epoch) during imports and exports.

BLOBs and CLOBs are only supported for imports. The BLOB/CLOB objects when imported are stored in a Sqoop-specific format and knowledge of this format is needed for processing these objects in a Pig/Hive job or another Map Reduce job.

Database column names are mapped to their lowercase equivalents when mapped to the HCatalog fields. Currently, case-sensitive database object names are not supported.

Projection of a set of columns from a table to an HCatalog table or loading to a column projection is allowed (subject to table constraints). The dynamic partitioning columns, if any, must be part of the projection when importing data into HCatalog tables.

Dynamic partitioning fields should be mapped to database columns that are defined with the NOT NULL attribute (although this is not validated). A null value during import for a dynamic partitioning column will abort the Sqoop job.

7.11. Support for HCatalog Data Types

All the primitive HCatalog types are supported. Currently all the complex HCatalog types are unsupported.

BLOB/CLOB database types are only supported for imports.

7.12. Providing Hive and HCatalog Libraries for the Sqoop Job

With the support for HCatalog added to Sqoop, any HCatalog job depends on a set of jar files being available both on the Sqoop client host and where the Map/Reduce tasks run. To run HCatalog jobs, the environment variable HADOOP_CLASSPATH must be set up as shown below before launching the Sqoop HCatalog jobs:

```
HADOOP_CLASSPATH=$(hcat -classpath)
export HADOOP_CLASSPATH
```

The necessary HCatalog dependencies will be copied to the distributed cache automatically by the Sqoop job.

7.13. Examples

Create an HCatalog table, such as:

```
hcat -e "create table txn(txn_date string, cust_id string, amount
float, store_id int) partitioned by (cust_id string) stored as
rcfile;"
```

Then use Sqoop to import and export the "txn" HCatalog table as follows:

Import

```
$SQOOP_HOME/bin/sqoop import --connect <jdbc-url> -table <table-
name> --hcatalog-table txn
```

Export

```
$SQOOP_HOME/bin/sqoop export --connect <jdbc-url> -table <table-
name> --hcatalog-table txn
```