Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

This is a brief tutorial that provides an introduction on how to use Apache Hive HiveQL with Hadoop Distributed File System. This tutorial can be your first step towards becoming a successful Hadoop Developer with Hive.

The term ‘Big Data’ is used for collections of large datasets that include huge volume, high velocity, and a variety of data that is increasing day by day. Using traditional data management systems, it is difficult to process Big Data. Therefore, the Apache Software Foundation introduced a framework called Hadoop to solve Big Data management and processing challenges.

Hadoop

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

* **MapReduce:** It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.
* **HDFS:**Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

* **Sqoop:** It is used to import and export data to and from between HDFS and RDBMS.
* **Pig:** It is a procedural language platform used to develop a script for MapReduce operations.
* **Hive:** It is a platform used to develop SQL type scripts to do MapReduce operations.

**Note:** There are various ways to execute MapReduce operations:

* The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
* The scripting approach for MapReduce to process structured and semi structured data using Pig.
* The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

What is Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

* A relational database
* A design for OnLine Transaction Processing (OLTP)
* A language for real-time queries and row-level updates

Features of Hive

* It stores schema in a database and processed data into HDFS.
* It is designed for OLAP.
* It provides SQL type language for querying called HiveQL or HQL.
* It is familiar, fast, scalable, and extensible.

Architecture of Hive

The following component diagram depicts the architecture of Hive:



This component diagram contains different units. The following table describes each unit:

|  |  |
| --- | --- |
| **Unit Name** | **Operation** |
| User Interface | Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server). |
| Meta Store | Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping. |
| HiveQL Process Engine | HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it. |
| Execution Engine | The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce. |
| HDFS or HBASE | Hadoop distributed file system or HBASE are the data storage techniques to store data into file system. |

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

|  |  |
| --- | --- |
| **Step No.** | **Operation** |
| 1 | **Execute Query**  The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute. |
| 2 | **Get Plan**  The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query. |
| 3 | **Get Metadata**  The compiler sends metadata request to Metastore (any database). |
| 4 | **Send Metadata**  Metastore sends metadata as a response to the compiler. |
| 5 | **Send Plan**  The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete. |
| 6 | **Execute Plan**  The driver sends the execute plan to the execution engine. |
| 7 | **Execute Job**  Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job. |
| 7.1 | **Metadata Ops**  Meanwhile in execution, the execution engine can execute metadata operations with Metastore. |
| 8 | **Fetch Result**  The execution engine receives the results from Data nodes. |
| 9 | **Send Results**  The execution engine sends those resultant values to the driver. |
| 10 | **Send Results**  The driver sends the results to Hive Interfaces. |

All Hadoop sub-projects such as Hive, Pig, and HBase support Linux operating system. Therefore, you need to install any Linux flavored OS. The following simple steps are executed for Hive installation:

## Step 1: Verifying JAVA Installation

Java must be installed on your system before installing Hive. Let us verify java installation using the following command:

$ java –version

If Java is already installed on your system, you get to see the following response:

java version "1.7.0\_71"

Java(TM) SE Runtime Environment (build 1.7.0\_71-b13)

Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)

If java is not installed in your system, then follow the steps given below for installing java.

## Installing Java

### Step I:

Download java (JDK <latest version> - X64.tar.gz) by visiting the following link [http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.](http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html)

Then jdk-7u71-linux-x64.tar.gz will be downloaded onto your system.

### Step II:

Generally you will find the downloaded java file in the Downloads folder. Verify it and extract the jdk-7u71-linux-x64.gz file using the following commands.

$ cd Downloads/

$ ls

jdk-7u71-linux-x64.gz

$ tar zxf jdk-7u71-linux-x64.gz

$ ls

jdk1.7.0\_71 jdk-7u71-linux-x64.gz

### Step III:

To make java available to all the users, you have to move it to the location “/usr/local/”. Open root, and type the following commands.

$ su

password:

# mv jdk1.7.0\_71 /usr/local/

# exit

### Step IV:

For setting up PATH and JAVA\_HOME variables, add the following commands to ~/.bashrc file.

export JAVA\_HOME=/usr/local/jdk1.7.0\_71

export PATH=$PATH:$JAVA\_HOME/bin

Now apply all the changes into the current running system.

$ source ~/.bashrc

### Step V:

Use the following commands to configure java alternatives:

# alternatives --install /usr/bin/java java usr/local/java/bin/java 2

# alternatives --install /usr/bin/javac javac usr/local/java/bin/javac 2

# alternatives --install /usr/bin/jar jar usr/local/java/bin/jar 2

# alternatives --set java usr/local/java/bin/java

# alternatives --set javac usr/local/java/bin/javac

# alternatives --set jar usr/local/java/bin/jar

Now verify the installation using the command java -version from the terminal as explained above.

## Step 2: Verifying Hadoop Installation

Hadoop must be installed on your system before installing Hive. Let us verify the Hadoop installation using the following command:

$ hadoop version

If Hadoop is already installed on your system, then you will get the following response:

Hadoop 2.4.1 Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1529768

Compiled by hortonmu on 2013-10-07T06:28Z

Compiled with protoc 2.5.0

From source with checksum 79e53ce7994d1628b240f09af91e1af4

If Hadoop is not installed on your system, then proceed with the following steps:

## Downloading Hadoop

Download and extract Hadoop 2.4.1 from Apache Software Foundation using the following commands.

$ su

password:

# cd /usr/local

# wget http://apache.claz.org/hadoop/common/hadoop-2.4.1/

hadoop-2.4.1.tar.gz

# tar xzf hadoop-2.4.1.tar.gz

# mv hadoop-2.4.1/\* to hadoop/

# exit

## Installing Hadoop in Pseudo Distributed Mode

The following steps are used to install Hadoop 2.4.1 in pseudo distributed mode.

### Step I: Setting up Hadoop

You can set Hadoop environment variables by appending the following commands to **~/.bashrc** file.

export HADOOP\_HOME=/usr/local/hadoop

export HADOOP\_MAPRED\_HOME=$HADOOP\_HOME

export HADOOP\_COMMON\_HOME=$HADOOP\_HOME

export HADOOP\_HDFS\_HOME=$HADOOP\_HOME

export YARN\_HOME=$HADOOP\_HOME

export HADOOP\_COMMON\_LIB\_NATIVE\_DIR=$HADOOP\_HOME/lib/native export

PATH=$PATH:$HADOOP\_HOME/sbin:$HADOOP\_HOME/bin

Now apply all the changes into the current running system.

$ source ~/.bashrc

### Step II: Hadoop Configuration

You can find all the Hadoop configuration files in the location “$HADOOP\_HOME/etc/hadoop”. You need to make suitable changes in those configuration files according to your Hadoop infrastructure.

$ cd $HADOOP\_HOME/etc/hadoop

In order to develop Hadoop programs using java, you have to reset the java environment variables in **hadoop-env.sh** file by replacing **JAVA\_HOME** value with the location of java in your system.

export JAVA\_HOME=/usr/local/jdk1.7.0\_71

Given below are the list of files that you have to edit to configure Hadoop.

**core-site.xml**

The **core-site.xml** file contains information such as the port number used for Hadoop instance, memory allocated for the file system, memory limit for storing the data, and the size of Read/Write buffers.

Open the core-site.xml and add the following properties in between the <configuration> and </configuration> tags.

<configuration>

<property>

<name>fs.default.name</name>

<value>hdfs://localhost:9000</value>

</property>

</configuration>

**hdfs-site.xml**

The **hdfs-site.xml** file contains information such as the value of replication data, the namenode path, and the datanode path of your local file systems. It means the place where you want to store the Hadoop infra.

Let us assume the following data.

dfs.replication (data replication value) = 1

(In the following path /hadoop/ is the user name.

hadoopinfra/hdfs/namenode is the directory created by hdfs file system.)

namenode path = //home/hadoop/hadoopinfra/hdfs/namenode

(hadoopinfra/hdfs/datanode is the directory created by hdfs file system.)

datanode path = //home/hadoop/hadoopinfra/hdfs/datanode

Open this file and add the following properties in between the <configuration>, </configuration> tags in this file.

<configuration>

<property>

<name>dfs.replication</name>

<value>1</value>

</property>

<property>

<name>dfs.name.dir</name>

<value>file:///home/hadoop/hadoopinfra/hdfs/namenode </value>

</property>

<property>

<name>dfs.data.dir</name>

<value>file:///home/hadoop/hadoopinfra/hdfs/datanode </value >

</property>

</configuration>

**Note:** In the above file, all the property values are user-defined and you can make changes according to your Hadoop infrastructure.

**yarn-site.xml**

This file is used to configure yarn into Hadoop. Open the yarn-site.xml file and add the following properties in between the <configuration>, </configuration> tags in this file.

<configuration>

<property>

<name>yarn.nodemanager.aux-services</name>

<value>mapreduce\_shuffle</value>

</property>

</configuration>

**mapred-site.xml**

This file is used to specify which MapReduce framework we are using. By default, Hadoop contains a template of yarn-site.xml. First of all, you need to copy the file from mapred-site,xml.template to mapred-site.xml file using the following command.

$ cp mapred-site.xml.template mapred-site.xml

Open **mapred-site.xml** file and add the following properties in between the <configuration>, </configuration> tags in this file.

<configuration>

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

</configuration>

## Verifying Hadoop Installation

The following steps are used to verify the Hadoop installation.

### Step I: Name Node Setup

Set up the namenode using the command “hdfs namenode -format” as follows.

$ cd ~

$ hdfs namenode -format

The expected result is as follows.

10/24/14 21:30:55 INFO namenode.NameNode: STARTUP\_MSG:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

STARTUP\_MSG: Starting NameNode

STARTUP\_MSG: host = localhost/192.168.1.11

STARTUP\_MSG: args = [-format]

STARTUP\_MSG: version = 2.4.1

...

...

10/24/14 21:30:56 INFO common.Storage: Storage directory

/home/hadoop/hadoopinfra/hdfs/namenode has been successfully formatted.

10/24/14 21:30:56 INFO namenode.NNStorageRetentionManager: Going to

retain 1 images with txid >= 0

10/24/14 21:30:56 INFO util.ExitUtil: Exiting with status 0

10/24/14 21:30:56 INFO namenode.NameNode: SHUTDOWN\_MSG:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SHUTDOWN\_MSG: Shutting down NameNode at localhost/192.168.1.11

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

### Step II: Verifying Hadoop dfs

The following command is used to start dfs. Executing this command will start your Hadoop file system.

$ start-dfs.sh

The expected output is as follows:

10/24/14 21:37:56

Starting namenodes on [localhost]

localhost: starting namenode, logging to /home/hadoop/hadoop-2.4.1/logs/hadoop-hadoop-namenode-localhost.out

localhost: starting datanode, logging to /home/hadoop/hadoop-2.4.1/logs/hadoop-hadoop-datanode-localhost.out

Starting secondary namenodes [0.0.0.0]

### Step III: Verifying Yarn Script

The following command is used to start the yarn script. Executing this command will start your yarn daemons.

$ start-yarn.sh

The expected output is as follows:

starting yarn daemons

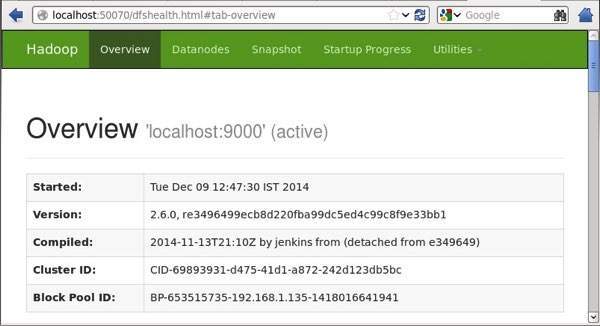
starting resourcemanager, logging to /home/hadoop/hadoop-2.4.1/logs/yarn-hadoop-resourcemanager-localhost.out

localhost: starting nodemanager, logging to /home/hadoop/hadoop-2.4.1/logs/yarn-hadoop-nodemanager-localhost.out

### Step IV: Accessing Hadoop on Browser

The default port number to access Hadoop is 50070. Use the following url to get Hadoop services on your browser.

http://localhost:50070/



### Step V: Verify all applications for cluster

The default port number to access all applications of cluster is 8088. Use the following url to visit this service.

http://localhost:8088/



### Step 3: Downloading Hive

We use hive-0.14.0 in this tutorial. You can download it by visiting the following link [http://apache.petsads.us/hive/hive-0.14.0/.](http://apache.petsads.us/hive/hive-0.14.0/) Let us assume it gets downloaded onto the /Downloads directory. Here, we download Hive archive named “apache-hive-0.14.0-bin.tar.gz” for this tutorial. The following command is used to verify the download:

$ cd Downloads

$ ls

On successful download, you get to see the following response:

apache-hive-0.14.0-bin.tar.gz

## Step 4: Installing Hive

The following steps are required for installing Hive on your system. Let us assume the Hive archive is downloaded onto the /Downloads directory.

### Extracting and verifying Hive Archive

The following command is used to verify the download and extract the hive archive:

$ tar zxvf apache-hive-0.14.0-bin.tar.gz

$ ls

On successful download, you get to see the following response:

apache-hive-0.14.0-bin apache-hive-0.14.0-bin.tar.gz

### Copying files to /usr/local/hive directory

We need to copy the files from the super user “su -”. The following commands are used to copy the files from the extracted directory to the /usr/local/hive” directory.

$ su -

passwd:

# cd /home/user/Download

# mv apache-hive-0.14.0-bin /usr/local/hive

# exit

### Setting up environment for Hive

You can set up the Hive environment by appending the following lines to **~/.bashrc** file:

export HIVE\_HOME=/usr/local/hive

export PATH=$PATH:$HIVE\_HOME/bin

export CLASSPATH=$CLASSPATH:/usr/local/Hadoop/lib/\*:.

export CLASSPATH=$CLASSPATH:/usr/local/hive/lib/\*:.

The following command is used to execute ~/.bashrc file.

$ source ~/.bashrc

## Step 5: Configuring Hive

To configure Hive with Hadoop, you need to edit the **hive-env.sh** file, which is placed in the **$HIVE\_HOME/conf** directory. The following commands redirect to Hive **config** folder and copy the template file:

$ cd $HIVE\_HOME/conf

$ cp hive-env.sh.template hive-env.sh

Edit the **hive-env.sh** file by appending the following line:

export HADOOP\_HOME=/usr/local/hadoop

Hive installation is completed successfully. Now you require an external database server to configure Metastore. We use Apache Derby database.

## Step 6: Downloading and Installing Apache Derby

Follow the steps given below to download and install Apache Derby:

### Downloading Apache Derby

The following command is used to download Apache Derby. It takes some time to download.

$ cd ~

$ wget http://archive.apache.org/dist/db/derby/db-derby-10.4.2.0/db-derby-10.4.2.0-bin.tar.gz

The following command is used to verify the download:

$ ls

On successful download, you get to see the following response:

db-derby-10.4.2.0-bin.tar.gz

### Extracting and verifying Derby archive

The following commands are used for extracting and verifying the Derby archive:

$ tar zxvf db-derby-10.4.2.0-bin.tar.gz

$ ls

On successful download, you get to see the following response:

db-derby-10.4.2.0-bin db-derby-10.4.2.0-bin.tar.gz

### Copying files to /usr/local/derby directory

We need to copy from the super user “su -”. The following commands are used to copy the files from the extracted directory to the /usr/local/derby directory:

$ su -

passwd:

# cd /home/user

# mv db-derby-10.4.2.0-bin /usr/local/derby

# exit

### Setting up environment for Derby

You can set up the Derby environment by appending the following lines to **~/.bashrc** file:

export DERBY\_HOME=/usr/local/derby

export PATH=$PATH:$DERBY\_HOME/bin

Apache Hive

18

export CLASSPATH=$CLASSPATH:$DERBY\_HOME/lib/derby.jar:$DERBY\_HOME/lib/derbytools.jar

The following command is used to execute **~/.bashrc** file:

$ source ~/.bashrc

### Create a directory to store Metastore

Create a directory named data in $DERBY\_HOME directory to store Metastore data.

$ mkdir $DERBY\_HOME/data

Derby installation and environmental setup is now complete.

## Step 7: Configuring Metastore of Hive

Configuring Metastore means specifying to Hive where the database is stored. You can do this by editing the hive-site.xml file, which is in the $HIVE\_HOME/conf directory. First of all, copy the template file using the following command:

$ cd $HIVE\_HOME/conf

$ cp hive-default.xml.template hive-site.xml

Edit **hive-site.xml** and append the following lines between the <configuration> and </configuration> tags:

<property>

<name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:derby://localhost:1527/metastore\_db;create=true </value>

<description>JDBC connect string for a JDBC metastore </description>

</property>

Create a file named jpox.properties and add the following lines into it:

javax.jdo.PersistenceManagerFactoryClass =

org.jpox.PersistenceManagerFactoryImpl

org.jpox.autoCreateSchema = false

org.jpox.validateTables = false

org.jpox.validateColumns = false

org.jpox.validateConstraints = false

org.jpox.storeManagerType = rdbms

org.jpox.autoCreateSchema = true

org.jpox.autoStartMechanismMode = checked

org.jpox.transactionIsolation = read\_committed

javax.jdo.option.DetachAllOnCommit = true

javax.jdo.option.NontransactionalRead = true

javax.jdo.option.ConnectionDriverName = org.apache.derby.jdbc.ClientDriver

javax.jdo.option.ConnectionURL = jdbc:derby://hadoop1:1527/metastore\_db;create = true

javax.jdo.option.ConnectionUserName = APP

javax.jdo.option.ConnectionPassword = mine

## Step 8: Verifying Hive Installation

Before running Hive, you need to create the **/tmp** folder and a separate Hive folder in HDFS. Here, we use the **/user/hive/warehouse** folder. You need to set write permission for these newly created folders as shown below:

chmod g+w

Now set them in HDFS before verifying Hive. Use the following commands:

$ $HADOOP\_HOME/bin/hadoop fs -mkdir /tmp

$ $HADOOP\_HOME/bin/hadoop fs -mkdir /user/hive/warehouse

$ $HADOOP\_HOME/bin/hadoop fs -chmod g+w /tmp

$ $HADOOP\_HOME/bin/hadoop fs -chmod g+w /user/hive/warehouse

The following commands are used to verify Hive installation:

$ cd $HIVE\_HOME

$ bin/hive

On successful installation of Hive, you get to see the following response:

Logging initialized using configuration in jar:file:/home/hadoop/hive-0.9.0/lib/hive-common-0.9.0.jar!/hive-log4j.properties

Hive history file=/tmp/hadoop/hive\_job\_log\_hadoop\_201312121621\_1494929084.txt

………………….

hive>

The following sample command is executed to display all the tables:

hive> show tables;

OK

Time taken: 2.798 seconds

hive>

This chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

* Column Types
* Literals
* Null Values
* Complex Types

Column Types

Column type are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

|  |  |  |
| --- | --- | --- |
| **Type** | **Postfix** | **Example** |
| TINYINT | Y | 10Y |
| SMALLINT | S | 10S |
| INT | - | 10 |
| BIGINT | L | 10L |

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

|  |  |
| --- | --- |
| **Data Type** | **Length** |
| VARCHAR | 1 to 65355 |
| CHAR | 255 |

Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.fffffffff” and format “yyyy-mm-dd hh:mm:ss.ffffffffff”.

Dates

DATE values are described in year/month/day format in the form {{YYYY-MM-DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

DECIMAL(precision, scale)

decimal(10,0)

Union Types

Union is a collection of heterogeneous data types. You can create an instance using **create union**. The syntax and example is as follows:

UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>

{0:1}

{1:2.0}

{2:["three","four"]}

{3:{"a":5,"b":"five"}}

{2:["six","seven"]}

{3:{"a":8,"b":"eight"}}

{0:9}

{1:10.0}

Literals

The following literals are used in Hive:

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10-308 to 10308.

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays in Hive are used the same way they are used in Java.

Syntax: ARRAY<data\_type>

Maps

Maps in Hive are similar to Java Maps.

Syntax: MAP<primitive\_type, data\_type>

Structs

Structs in Hive is similar to using complex data with comment.

Syntax: STRUCT<col\_name : data\_type [COMMENT col\_comment], ...>

Hive is a database technology that can define databases and tables to analyze structured data. The theme for structured data analysis is to store the data in a tabular manner, and pass queries to analyze it. This chapter explains how to create Hive database. Hive contains a default database named **default**.

## Create Database Statement

Create Database is a statement used to create a database in Hive. A database in Hive is a **namespace** or a collection of tables. The **syntax** for this statement is as follows:

CREATE DATABASE|SCHEMA [IF NOT EXISTS] <database name>

Here, IF NOT EXISTS is an optional clause, which notifies the user that a database with the same name already exists. We can use SCHEMA in place of DATABASE in this command. The following query is executed to create a database named **userdb**:

hive> CREATE DATABASE [IF NOT EXISTS] userdb;

**or**

hive> CREATE SCHEMA userdb;

The following query is used to verify a databases list:

hive> SHOW DATABASES;

default

userdb

### JDBC Program

The JDBC program to create a database is given below.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveCreateDb {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/default", "", "");

Statement stmt = con.createStatement();

stmt.executeQuery("CREATE DATABASE userdb");

System.out.println(“Database userdb created successfully.”);

con.close();

}

}

Save the program in a file named HiveCreateDb.java. The following commands are used to compile and execute this program.

$ javac HiveCreateDb.java

$ java HiveCreateDb

### Output:

Database userdb created successfully.

This chapter describes how to drop a database in Hive. The usage of SCHEMA and DATABASE are same.

## Drop Database Statement

Drop Database is a statement that drops all the tables and deletes the database. Its syntax is as follows:

DROP DATABASE StatementDROP (DATABASE|SCHEMA) [IF EXISTS] database\_name

[RESTRICT|CASCADE];

The following queries are used to drop a database. Let us assume that the database name is **userdb**.

hive> DROP DATABASE IF EXISTS userdb;

The following query drops the database using **CASCADE**. It means dropping respective tables before dropping the database.

hive> DROP DATABASE IF EXISTS userdb CASCADE;

The following query drops the database using **SCHEMA**.

hive> DROP SCHEMA userdb;

This clause was added in Hive 0.6.

### JDBC Program

The JDBC program to drop a database is given below.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveDropDb {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/default", "", "");

Statement stmt = con.createStatement();

stmt.executeQuery("DROP DATABASE userdb");

System.out.println(“Drop userdb database successful.”);

con.close();

}

}

Save the program in a file named HiveDropDb.java. Given below are the commands to compile and execute this program.

$ javac HiveDropDb.java

$ java HiveDropDb

### Output:

Drop userdb database successful.

This chapter explains how to create a table and how to insert data into it. The conventions of creating a table in HIVE is quite similar to creating a table using SQL.

Create Table Statement

Create Table is a statement used to create a table in Hive. The syntax and example are as follows:

Syntax

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db\_name.] table\_name

[(col\_name data\_type [COMMENT col\_comment], ...)]

[COMMENT table\_comment]

[ROW FORMAT row\_format]

[STORED AS file\_format]

Example

Let us assume you need to create a table named **employee** using **CREATE TABLE** statement. The following table lists the fields and their data types in employee table:

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Field Name** | **Data Type** |
| 1 | Eid | int |
| 2 | Name | String |
| 3 | Salary | Float |
| 4 | Designation | string |

The following data is a Comment, Row formatted fields such as Field terminator, Lines terminator, and Stored File type.

COMMENT ‘Employee details’

FIELDS TERMINATED BY ‘\t’

LINES TERMINATED BY ‘\n’

STORED IN TEXT FILE

The following query creates a table named **employee** using the above data.

hive> CREATE TABLE IF NOT EXISTS employee ( eid int, name String,

salary String, destination String)

COMMENT ‘Employee details’

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘\t’

LINES TERMINATED BY ‘\n’

STORED AS TEXTFILE;

If you add the option IF NOT EXISTS, Hive ignores the statement in case the table already exists.

On successful creation of table, you get to see the following response:

OK

Time taken: 5.905 seconds

hive>

JDBC Program

The JDBC program to create a table is given example.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveCreateTable {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("CREATE TABLE IF NOT EXISTS "

+" employee ( eid int, name String, "

+" salary String, destignation String)"

+" COMMENT ‘Employee details’"

+" ROW FORMAT DELIMITED"

+" FIELDS TERMINATED BY ‘\t’"

+" LINES TERMINATED BY ‘\n’"

+" STORED AS TEXTFILE;");

System.out.println(“ Table employee created.”);

con.close();

}

}

Save the program in a file named HiveCreateDb.java. The following commands are used to compile and execute this program.

$ javac HiveCreateDb.java

$ java HiveCreateDb

Output

Table employee created.

Load Data Statement

Generally, after creating a table in SQL, we can insert data using the Insert statement. But in Hive, we can insert data using the LOAD DATA statement.

While inserting data into Hive, it is better to use LOAD DATA to store bulk records. There are two ways to load data: one is from local file system and second is from Hadoop file system.

Syntax

The syntax for load data is as follows:

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename

[PARTITION (partcol1=val1, partcol2=val2 ...)]

* LOCAL is identifier to specify the local path. It is optional.
* OVERWRITE is optional to overwrite the data in the table.
* PARTITION is optional.

Example

We will insert the following data into the table. It is a text file named **sample.txt** in **/home/user** directory.

1201 Gopal 45000 Technical manager

1202 Manisha 45000 Proof reader

1203 Masthanvali 40000 Technical writer

1204 Kiran 40000 Hr Admin

1205 Kranthi 30000 Op Admin

The following query loads the given text into the table.

hive> LOAD DATA LOCAL INPATH '/home/user/sample.txt'

OVERWRITE INTO TABLE employee;

On successful download, you get to see the following response:

OK

Time taken: 15.905 seconds

hive>

JDBC Program

Given below is the JDBC program to load given data into the table.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveLoadData {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("LOAD DATA LOCAL INPATH '/home/user/sample.txt'" + "OVERWRITE INTO TABLE employee;");

System.out.println("Load Data into employee successful");

con.close();

}

}

Save the program in a file named HiveLoadData.java. Use the following commands to compile and execute this program.

$ javac HiveLoadData.java

$ java HiveLoadData

Output:

Load Data into employee successful

This chapter explains how to alter the attributes of a table such as changing its table name, changing column names, adding columns, and deleting or replacing columns.

## Alter Table Statement

It is used to alter a table in Hive.

### Syntax

The statement takes any of the following syntaxes based on what attributes we wish to modify in a table.

ALTER TABLE name RENAME TO new\_name

ALTER TABLE name ADD COLUMNS (col\_spec[, col\_spec ...])

ALTER TABLE name DROP [COLUMN] column\_name

ALTER TABLE name CHANGE column\_name new\_name new\_type

ALTER TABLE name REPLACE COLUMNS (col\_spec[, col\_spec ...])

## Rename To… Statement

The following query renames the table from **employee** to **emp**.

hive> ALTER TABLE employee RENAME TO emp;

### JDBC Program

The JDBC program to rename a table is as follows.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveAlterRenameTo {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("ALTER TABLE employee RENAME TO emp;");

System.out.println("Table Renamed Successfully");

con.close();

}

}

Save the program in a file named HiveAlterRenameTo.java. Use the following commands to compile and execute this program.

$ javac HiveAlterRenameTo.java

$ java HiveAlterRenameTo

### Output:

Table renamed successfully.

## Change Statement

The following table contains the fields of **employee** table and it shows the fields to be changed (in bold).

|  |  |  |  |
| --- | --- | --- | --- |
| **Field Name** | **Convert from Data Type** | **Change Field Name** | **Convert to Data Type** |
| eid | int | eid | int |
| **name** | String | **ename** | String |
| salary | **Float** | salary | **Double** |
| designation | String | designation | String |

The following queries rename the column name and column data type using the above data:

hive> ALTER TABLE employee CHANGE name ename String;

hive> ALTER TABLE employee CHANGE salary salary Double;

### JDBC Program

Given below is the JDBC program to change a column.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveAlterChangeColumn {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("ALTER TABLE employee CHANGE name ename String;");

stmt.executeQuery("ALTER TABLE employee CHANGE salary salary Double;");

System.out.println("Change column successful.");

con.close();

}

}

Save the program in a file named HiveAlterChangeColumn.java. Use the following commands to compile and execute this program.

$ javac HiveAlterChangeColumn.java

$ java HiveAlterChangeColumn

### Output:

Change column successful.

## Add Columns Statement

The following query adds a column named dept to the employee table.

hive> ALTER TABLE employee ADD COLUMNS (

dept STRING COMMENT 'Department name');

## JDBC Program

The JDBC program to add a column to a table is given below.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveAlterAddColumn {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("ALTER TABLE employee ADD COLUMNS " + " (dept STRING COMMENT 'Department name');");

System.out.prinln("Add column successful.");

con.close();

}

}

Save the program in a file named HiveAlterAddColumn.java. Use the following commands to compile and execute this program.

$ javac HiveAlterAddColumn.java

$ java HiveAlterAddColumn

### Output:

Add column successful.

## Replace Statement

The following query deletes all the columns from the **employee** table and replaces it with **emp** and **name** columns:

hive> ALTER TABLE employee REPLACE COLUMNS (

eid INT empid Int,

ename STRING name String);

## JDBC Program

Given below is the JDBC program to replace **eid** column with **empid** and **ename**column with **name**.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveAlterReplaceColumn {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("ALTER TABLE employee REPLACE COLUMNS "

+" (eid INT empid Int,"

+" ename STRING name String);");

System.out.println(" Replace column successful");

con.close();

}

}

Save the program in a file named HiveAlterReplaceColumn.java. Use the following commands to compile and execute this program.

$ javac HiveAlterReplaceColumn.java

$ java HiveAlterReplaceColumn

### Output:

Replace column successful

This chapter describes how to drop a table in Hive. When you drop a table from Hive Metastore, it removes the table/column data and their metadata. It can be a normal table (stored in Metastore) or an external table (stored in local file system); Hive treats both in the same manner, irrespective of their types.

## Drop Table Statement

The syntax is as follows:

DROP TABLE [IF EXISTS] table\_name;

The following query drops a table named **employee**:

hive> DROP TABLE IF EXISTS employee;

On successful execution of the query, you get to see the following response:

OK

Time taken: 5.3 seconds

hive>

### JDBC Program

The following JDBC program drops the employee table.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveDropTable {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

stmt.executeQuery("DROP TABLE IF EXISTS employee;");

System.out.println("Drop table successful.");

con.close();

}

}

Save the program in a file named HiveDropTable.java. Use the following commands to compile and execute this program.

$ javac HiveDropTable.java

$ java HiveDropTable

### Output:

Drop table successful

The following query is used to verify the list of tables:

hive> SHOW TABLES;

emp

ok

Time taken: 2.1 seconds

hive>

Hive organizes tables into partitions. It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, and department. Using partition, it is easy to query a portion of the data.

Tables or partitions are sub-divided into **buckets,** to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

For example, a table named **Tab1** contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

The following file contains employeedata table.

/tab1/employeedata/file1

id, name, dept, yoj

1, gopal, TP, 2012

2, kiran, HR, 2012

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2

1, gopal, TP, 2012

2, kiran, HR, 2012

/tab1/employeedata/2013/file3

3, kaleel,SC, 2013

4, Prasanth, SC, 2013

## Adding a Partition

We can add partitions to a table by altering the table. Let us assume we have a table called **employee** with fields such as Id, Name, Salary, Designation, Dept, and yoj.

### Syntax:

ALTER TABLE table\_name ADD [IF NOT EXISTS] PARTITION partition\_spec

[LOCATION 'location1'] partition\_spec [LOCATION 'location2'] ...;

partition\_spec:

: (p\_column = p\_col\_value, p\_column = p\_col\_value, ...)

The following query is used to add a partition to the employee table.

hive> ALTER TABLE employee

> ADD PARTITION (year=’2013’)

> location '/2012/part2012';

## Renaming a Partition

The syntax of this command is as follows.

ALTER TABLE table\_name PARTITION partition\_spec RENAME TO PARTITION partition\_spec;

The following query is used to rename a partition:

hive> ALTER TABLE employee PARTITION (year=’1203’)

> RENAME TO PARTITION (Yoj=’1203’);

## Dropping a Partition

The following syntax is used to drop a partition:

ALTER TABLE table\_name DROP [IF EXISTS] PARTITION partition\_spec, PARTITION partition\_spec,...;

The following query is used to drop a partition:

hive> ALTER TABLE employee DROP [IF EXISTS]

> PARTITION (year=’1203’);

This chapter explains the built-in operators of Hive. There are four types of operators in Hive:

* Relational Operators
* Arithmetic Operators
* Logical Operators
* Complex Operators

Relational Operators

These operators are used to compare two operands. The following table describes the relational operators available in Hive:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand** | **Description** |
| A = B | all primitive types | TRUE if expression A is equivalent to expression B otherwise FALSE. |
| A != B | all primitive types | TRUE if expression A is not equivalent to expression B otherwise FALSE. |
| A < B | all primitive types | TRUE if expression A is less than expression B otherwise FALSE. |
| A <= B | all primitive types | TRUE if expression A is less than or equal to expression B otherwise FALSE. |
| A > B | all primitive types | TRUE if expression A is greater than expression B otherwise FALSE. |
| A >= B | all primitive types | TRUE if expression A is greater than or equal to expression B otherwise FALSE. |
| A IS NULL | all types | TRUE if expression A evaluates to NULL otherwise FALSE. |
| A IS NOT NULL | all types | FALSE if expression A evaluates to NULL otherwise TRUE. |
| A LIKE B | Strings | TRUE if string pattern A matches to B otherwise FALSE. |
| A RLIKE B | Strings | NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B , otherwise FALSE. |
| A REGEXP B | Strings | Same as RLIKE. |

Example

Let us assume the **employee** table is composed of fields named Id, Name, Salary, Designation, and Dept as shown below. Generate a query to retrieve the employee details whose Id is 1205.

+-----+--------------+--------+---------------------------+------+

| Id | Name | Salary | Designation | Dept |

+-----+--------------+------------------------------------+------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin|

+-----+--------------+--------+---------------------------+------+

The following query is executed to retrieve the employee details using the above table:

hive> SELECT \* FROM employee WHERE Id=1205;

On successful execution of query, you get to see the following response:

+-----+-----------+-----------+----------------------------------+

| ID | Name | Salary | Designation | Dept |

+-----+---------------+-------+----------------------------------+

|1205 | Kranthi | 30000 | Op Admin | Admin |

+-----+-----------+-----------+----------------------------------+

The following query is executed to retrieve the employee details whose salary is more than or equal to Rs 40000.

hive> SELECT \* FROM employee WHERE Salary>=40000;

On successful execution of query, you get to see the following response:

+-----+------------+--------+----------------------------+------+

| ID | Name | Salary | Designation | Dept |

+-----+------------+--------+----------------------------+------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali| 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

+-----+------------+--------+----------------------------+------+

Arithmetic Operators

These operators support various common arithmetic operations on the operands. All of them return number types. The following table describes the arithmetic operators available in Hive:

|  |  |  |
| --- | --- | --- |
| **Operators** | **Operand** | **Description** |
| A + B | all number types | Gives the result of adding A and B. |
| A - B | all number types | Gives the result of subtracting B from A. |
| A \* B | all number types | Gives the result of multiplying A and B. |
| A / B | all number types | Gives the result of dividing B from A. |
| A % B | all number types | Gives the reminder resulting from dividing A by B. |
| A & B | all number types | Gives the result of bitwise AND of A and B. |
| A | B | all number types | Gives the result of bitwise OR of A and B. |
| A ^ B | all number types | Gives the result of bitwise XOR of A and B. |
| ~A | all number types | Gives the result of bitwise NOT of A. |

Example

The following query adds two numbers, 20 and 30.

hive> SELECT 20+30 ADD FROM temp;

On successful execution of the query, you get to see the following response:

+--------+

| ADD |

+--------+

| 50 |

+--------+

Logical Operators

The operators are logical expressions. All of them return either TRUE or FALSE.

|  |  |  |
| --- | --- | --- |
| **Operators** | **Operands** | **Description** |
| A AND B | boolean | TRUE if both A and B are TRUE, otherwise FALSE. |
| A && B | boolean | Same as A AND B. |
| A OR B | boolean | TRUE if either A or B or both are TRUE, otherwise FALSE. |
| A || B | boolean | Same as A OR B. |
| NOT A | boolean | TRUE if A is FALSE, otherwise FALSE. |
| !A | boolean | Same as NOT A. |

Example

The following query is used to retrieve employee details whose Department is TP and Salary is more than Rs 40000.

hive> SELECT \* FROM employee WHERE Salary>40000 && Dept=TP;

On successful execution of the query, you get to see the following response:

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

+------+--------------+-------------+-------------------+--------+

Complex Operators

These operators provide an expression to access the elements of Complex Types.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operand** | **Description** |
| A[n] | A is an Array and n is an int | It returns the nth element in the array A. The first element has index 0. |
| M[key] | M is a Map<K, V> and key has type K | It returns the value corresponding to the key in the map. |
| S.x | S is a struct | It returns the x field of S. |

This chapter explains the built-in functions available in Hive. The functions look quite similar to SQL functions, except for their usage.

## Built-In Functions

Hive supports the following built-in functions:

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Signature** | **Description** |
| BIGINT | round(double a) | It returns the rounded BIGINT value of the double. |
| BIGINT | floor(double a) | It returns the maximum BIGINT value that is equal or less than the double. |
| BIGINT | ceil(double a) | It returns the minimum BIGINT value that is equal or greater than the double. |
| double | rand(), rand(int seed) | It returns a random number that changes from row to row. |
| string | concat(string A, string B,...) | It returns the string resulting from concatenating B after A. |
| string | substr(string A, int start) | It returns the substring of A starting from start position till the end of string A. |
| string | substr(string A, int start, int length) | It returns the substring of A starting from start position with the given length. |
| string | upper(string A) | It returns the string resulting from converting all characters of A to upper case. |
|  |  |  |
| string | ucase(string A) | Same as above. |
| string | lower(string A) | It returns the string resulting from converting all characters of B to lower case. |
| string | lcase(string A) | Same as above. |
| string | trim(string A) | It returns the string resulting from trimming spaces from both ends of A. |
| string | ltrim(string A) | It returns the string resulting from trimming spaces from the beginning (left hand side) of A. |
| string | rtrim(string A) | rtrim(string A) It returns the string resulting from trimming spaces from the end (right hand side) of A. |
| string | regexp\_replace(string A, string B, string C) | It returns the string resulting from replacing all substrings in B that match the Java regular expression syntax with C. |
| int | size(Map<K.V>) | It returns the number of elements in the map type. |
| int | size(Array<T>) | It returns the number of elements in the array type. |
| value of <type> | cast(<expr> as <type>) | It converts the results of the expression expr to <type> e.g. cast('1' as BIGINT) converts the string '1' to it integral representation. A NULL is returned if the conversion does not succeed. |
| string | from\_unixtime(int unixtime) | convert the number of seconds from Unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00" |
| string | to\_date(string timestamp) | It returns the date part of a timestamp string: to\_date("1970-01-01 00:00:00") = "1970-01-01" |
| int | year(string date) | It returns the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970 |
| int | month(string date) | It returns the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11 |
| int | day(string date) | It returns the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1 |
| string | get\_json\_object(string json\_string, string path) | It extracts json object from a json string based on json path specified, and returns json string of the extracted json object. It returns NULL if the input json string is invalid. |

### Example

The following queries demonstrate some built-in functions:

### round() function

hive> SELECT round(2.6) from temp;

On successful execution of query, you get to see the following response:

3.0

### floor() function

hive> SELECT floor(2.6) from temp;

On successful execution of the query, you get to see the following response:

2.0

### ceil() function

hive> SELECT ceil(2.6) from temp;

On successful execution of the query, you get to see the following response:

3.0

## Aggregate Functions

Hive supports the following built-in **aggregate functions**. The usage of these functions is as same as the SQL aggregate functions.

|  |  |  |
| --- | --- | --- |
| **Return Type** | **Signature** | **Description** |
| BIGINT | count(\*), count(expr), | count(\*) - Returns the total number of retrieved rows. |
| DOUBLE | sum(col), sum(DISTINCT col) | It returns the sum of the elements in the group or the sum of the distinct values of the column in the group. |
| DOUBLE | avg(col), avg(DISTINCT col) | It returns the average of the elements in the group or the average of the distinct values of the column in the group. |
| DOUBLE | min(col) | It returns the minimum value of the column in the group. |
| DOUBLE | max(col) | It returns the maximum value of the column in the group. |

This chapter describes how to create and manage views. Views are generated based on user requirements. You can save any result set data as a view. The usage of view in Hive is same as that of the view in SQL. It is a standard RDBMS concept. We can execute all DML operations on a view.

## Creating a View

You can create a view at the time of executing a SELECT statement. The syntax is as follows:

CREATE VIEW [IF NOT EXISTS] view\_name [(column\_name [COMMENT column\_comment], ...) ]

[COMMENT table\_comment]

AS SELECT ...

## Example

Let us take an example for view. Assume employee table as given below, with the fields Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000. We store the result in a view named **emp\_30000.**

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

The following query retrieves the employee details using the above scenario:

hive> CREATE VIEW emp\_30000 AS

SELECT \* FROM employee

WHERE salary>30000;

## Dropping a View

Use the following syntax to drop a view:

DROP VIEW view\_name

The following query drops a view named as emp\_30000:

hive> DROP VIEW emp\_30000;

## Creating an Index

An Index is nothing but a pointer on a particular column of a table. Creating an index means creating a pointer on a particular column of a table. Its syntax is as follows:

CREATE INDEX index\_name

ON TABLE base\_table\_name (col\_name, ...)

AS 'index.handler.class.name'

[WITH DEFERRED REBUILD]

[IDXPROPERTIES (property\_name=property\_value, ...)]

[IN TABLE index\_table\_name]

[PARTITIONED BY (col\_name, ...)]

[

[ ROW FORMAT ...] STORED AS ...

| STORED BY ...

]

[LOCATION hdfs\_path]

[TBLPROPERTIES (...)]

## Example

Let us take an example for index. Use the same employee table that we have used earlier with the fields Id, Name, Salary, Designation, and Dept. Create an index named index\_salary on the salary column of the employee table.

The following query creates an index:

hive> CREATE INDEX inedx\_salary ON TABLE employee(salary)

AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler';

It is a pointer to the salary column. If the column is modified, the changes are stored using an index value.

## Dropping an Index

The following syntax is used to drop an index:

DROP INDEX <index\_name> ON <table\_name>

The following query drops an index named index\_salary:

hive> DROP INDEX index\_salary ON employee;

The Hive Query Language (HiveQL) is a query language for Hive to process and analyze structured data in a Metastore. This chapter explains how to use the SELECT statement with WHERE clause.

SELECT statement is used to retrieve the data from a table. WHERE clause works similar to a condition. It filters the data using the condition and gives you a finite result. The built-in operators and functions generate an expression, which fulfils the condition.

## Syntax

Given below is the syntax of the SELECT query:

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

[WHERE where\_condition]

[GROUP BY col\_list]

[HAVING having\_condition]

[CLUSTER BY col\_list | [DISTRIBUTE BY col\_list] [SORT BY col\_list]]

[LIMIT number];

## Example

Let us take an example for SELECT…WHERE clause. Assume we have the employee table as given below, with fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details who earn a salary of more than Rs 30000.

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

The following query retrieves the employee details using the above scenario:

hive> SELECT \* FROM employee WHERE salary>30000;

On successful execution of the query, you get to see the following response:

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

+------+--------------+-------------+-------------------+--------+

### JDBC Program

The JDBC program to apply where clause for the given example is as follows.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveQLWhere {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

Resultset res = stmt.executeQuery("SELECT \* FROM employee WHERE salary>30000;");

System.out.println("Result:");

System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

while (res.next()) {

System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " + res.getString(4) + " " + res.getString(5));

}

con.close();

}

}

Save the program in a file named HiveQLWhere.java. Use the following commands to compile and execute this program.

$ javac HiveQLWhere.java

$ java HiveQLWhere

### Output:

ID Name Salary Designation Dept

1201 Gopal 45000 Technical manager TP

1202 Manisha 45000 Proofreader PR

1203 Masthanvali 40000 Technical writer TP

1204 Krian 40000 Hr Admin HR

This chapter explains how to use the ORDER BY clause in a SELECT statement. The ORDER BY clause is used to retrieve the details based on one column and sort the result set by ascending or descending order.

## Syntax

Given below is the syntax of the ORDER BY clause:

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

[WHERE where\_condition]

[GROUP BY col\_list]

[HAVING having\_condition]

[ORDER BY col\_list]]

[LIMIT number];

## Example

Let us take an example for SELECT...ORDER BY clause. Assume employee table as given below, with the fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details in order by using Department name.

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

The following query retrieves the employee details using the above scenario:

hive> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;

On successful execution of the query, you get to see the following response:

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1205 | Kranthi | 30000 | Op Admin | Admin |

|1204 | Krian | 40000 | Hr Admin | HR |

|1202 | Manisha | 45000 | Proofreader | PR |

|1201 | Gopal | 45000 | Technical manager | TP |

|1203 | Masthanvali | 40000 | Technical writer | TP |

+------+--------------+-------------+-------------------+--------+

### JDBC Program

Here is the JDBC program to apply Order By clause for the given example.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveQLOrderBy {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

Resultset res = stmt.executeQuery("SELECT \* FROM employee ORDER BY DEPT;");

System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

while (res.next()) {

System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " + res.getString(4) + " " + res.getString(5));

}

con.close();

}

}

Save the program in a file named HiveQLOrderBy.java. Use the following commands to compile and execute this program.

$ javac HiveQLOrderBy.java

$ java HiveQLOrderBy

### Output:

ID Name Salary Designation Dept

1205 Kranthi 30000 Op Admin Admin

1204 Krian 40000 Hr Admin HR

1202 Manisha 45000 Proofreader PR

1201 Gopal 45000 Technical manager TP

1203 Masthanvali 40000 Technical writer TP

1204 Krian 40000 Hr Admin HR

This chapter explains how to use the ORDER BY clause in a SELECT statement. The ORDER BY clause is used to retrieve the details based on one column and sort the result set by ascending or descending order.

## Syntax

Given below is the syntax of the ORDER BY clause:

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

[WHERE where\_condition]

[GROUP BY col\_list]

[HAVING having\_condition]

[ORDER BY col\_list]]

[LIMIT number];

## Example

Let us take an example for SELECT...ORDER BY clause. Assume employee table as given below, with the fields named Id, Name, Salary, Designation, and Dept. Generate a query to retrieve the employee details in order by using Department name.

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 40000 | Hr Admin | HR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

The following query retrieves the employee details using the above scenario:

hive> SELECT Id, Name, Dept FROM employee ORDER BY DEPT;

On successful execution of the query, you get to see the following response:

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1205 | Kranthi | 30000 | Op Admin | Admin |

|1204 | Krian | 40000 | Hr Admin | HR |

|1202 | Manisha | 45000 | Proofreader | PR |

|1201 | Gopal | 45000 | Technical manager | TP |

|1203 | Masthanvali | 40000 | Technical writer | TP |

+------+--------------+-------------+-------------------+--------+

### JDBC Program

Here is the JDBC program to apply Order By clause for the given example.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveQLOrderBy {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

Resultset res = stmt.executeQuery("SELECT \* FROM employee ORDER BY DEPT;");

System.out.println(" ID \t Name \t Salary \t Designation \t Dept ");

while (res.next()) {

System.out.println(res.getInt(1) + " " + res.getString(2) + " " + res.getDouble(3) + " " + res.getString(4) + " " + res.getString(5));

}

con.close();

}

}

Save the program in a file named HiveQLOrderBy.java. Use the following commands to compile and execute this program.

$ javac HiveQLOrderBy.java

$ java HiveQLOrderBy

### Output:

ID Name Salary Designation Dept

1205 Kranthi 30000 Op Admin Admin

1204 Krian 40000 Hr Admin HR

1202 Manisha 45000 Proofreader PR

1201 Gopal 45000 Technical manager TP

1203 Masthanvali 40000 Technical writer TP

1204 Krian 40000 Hr Admin HR

This chapter explains the details of GROUP BY clause in a SELECT statement. The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

## Syntax

The syntax of GROUP BY clause is as follows:

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

[WHERE where\_condition]

[GROUP BY col\_list]

[HAVING having\_condition]

[ORDER BY col\_list]]

[LIMIT number];

## Example

Let us take an example of SELECT…GROUP BY clause. Assume employee table as given below, with Id, Name, Salary, Designation, and Dept fields. Generate a query to retrieve the number of employees in each department.

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 45000 | Proofreader | PR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

The following query retrieves the employee details using the above scenario.

hive> SELECT Dept,count(\*) FROM employee GROUP BY DEPT;

On successful execution of the query, you get to see the following response:

+------+--------------+

| Dept | Count(\*) |

+------+--------------+

|Admin | 1 |

|PR | 2 |

|TP | 3 |

+------+--------------+

### JDBC Program

Given below is the JDBC program to apply the Group By clause for the given example.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveQLGroupBy {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.

getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

Resultset res = stmt.executeQuery(“SELECT Dept,count(\*) ” + “FROM employee GROUP BY DEPT; ”);

System.out.println(" Dept \t count(\*)");

while (res.next()) {

System.out.println(res.getString(1) + " " + res.getInt(2));

}

con.close();

}

}

Save the program in a file named HiveQLGroupBy.java. Use the following commands to compile and execute this program.

$ javac HiveQLGroupBy.java

$ java HiveQLGroupBy

### Output:

Dept Count(\*)

Admin 1

PR 2

TP 3

JOIN is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database. It is more or less similar to SQL JOIN.

Syntax

join\_table:

table\_reference JOIN table\_factor [join\_condition]

| table\_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table\_reference

join\_condition

| table\_reference LEFT SEMI JOIN table\_reference join\_condition

| table\_reference CROSS JOIN table\_reference [join\_condition]

Example

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Consider another table ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

There are different types of joins given as follows:

* JOIN
* LEFT OUTER JOIN
* RIGHT OUTER JOIN
* FULL OUTER JOIN

JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT

FROM CUSTOMERS c JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

LEFT OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables.

notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

FULL OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

# **Hive Tutorial**

* [Hive Tutorial](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-HiveTutorial)
* [Concepts](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Concepts)
  + [What Is Hive](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-WhatIsHive)
  + [What Hive Is NOT](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-WhatHiveIsNOT)
  + [Getting Started](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-GettingStarted)
  + [Data Units](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-DataUnits)
  + [Type System](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-TypeSystem)
  + [Built In Operators and Functions](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-BuiltInOperatorsandFunctions)
  + [Language Capabilities](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-LanguageCapabilities)
* [Usage and Examples](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-UsageandExamples)
  + [Creating, Showing, Altering, and Dropping Tables](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Creating,Showing,Altering,andDroppingTables)
  + [Loading Data](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-LoadingData)
  + [Querying and Inserting Data](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-QueryingandInsertingData)

# **Concepts**

## What Is Hive

Hive is a data warehousing infrastructure based on [Apache Hadoop](http://hadoop.apache.org/). Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing on commodity hardware.

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data. It provides SQL which enables users to do ad-hoc querying, summarization and data analysis easily. At the same time, Hive's SQL gives users multiple places to integrate their own functionality to do custom analysis, such as User Defined Functions (UDFs).

## What Hive Is NOT

Hive is not designed for online transaction processing.  It is best used for traditional data warehousing tasks.

## Getting Started

For details on setting up Hive, HiveServer2, and Beeline, please refer to the [GettingStarted](https://cwiki.apache.org/confluence/display/Hive/GettingStarted) guide.

[Books about Hive](https://cwiki.apache.org/confluence/display/Hive/Books+about+Hive) lists some books that may also be helpful for getting started with Hive.

In the following sections we provide a tutorial on the capabilities of the system. We start by describing the concepts of data types, tables, and partitions (which are very similar to what you would find in a traditional relational DBMS) and then illustrate the capabilities of Hive with the help of some examples.

## Data Units

In the order of granularity - Hive data is organized into:

* **Databases**: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on.  Databases can also be used to enforce security for a user or group of users.
* **Tables**: Homogeneous units of data which have the same schema. An example of a table could be page\_views table, where each row could comprise of the following columns (schema):
  + timestamp—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
  + userid —which is of BIGINT type that identifies the user who viewed the page.
  + page\_url—which is of STRING type that captures the location of the page.
  + referer\_url—which is of STRING that captures the location of the page from where the user arrived at the current page.
  + IP—which is of STRING type that captures the IP address from where the page request was made.
* **Partitions**: Each Table can have one or more partition Keys which determines how the data is stored. Partitions—apart from being storage units—also allow the user to efficiently identify the rows that satisfy a specified criteria; for example, a date\_partition of type STRING and country\_partition of type STRING. Each unique value of the partition keys defines a partition of the Table. For example, all "US" data from "2009-12-23" is a partition of the page\_views table. Therefore, if you run analysis on only the "US" data for 2009-12-23, you can run that query only on the relevant partition of the table, thereby speeding up the analysis significantly. Note however, that just because a partition is named 2009-12-23 does not mean that it contains all or only data from that date; partitions are named after dates for convenience; it is the user's job to guarantee the relationship between partition name and data content! Partition columns are virtual columns, they are not part of the data itself but are derived on load.
* **Buckets** (or **Clusters**): Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page\_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page\_view table. These can be used to efficiently sample the data.

Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

## Type System

Hive supports primitive and complex data types, as described below. See [Hive Data Types](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types) for additional information.

### Primitive Types

* Types are associated with the columns in the tables. The following Primitive types are supported:
* Integers
  + TINYINT—1 byte integer
  + SMALLINT—2 byte integer
  + INT—4 byte integer
  + BIGINT—8 byte integer
* Boolean type
  + BOOLEAN—TRUE/FALSE
* Floating point numbers
  + FLOAT—single precision
  + DOUBLE—Double precision
* Fixed point numbers
  + DECIMAL—a fixed point value of user defined scale and precision
* String types
  + STRING—sequence of characters in a specified character set
  + VARCHAR—sequence of characters in a specified character set with a maximum length
  + CHAR—sequence of characters in a specified character set with a defined length
* Date and time types
  + TIMESTAMP— a specific point in time, up to nanosecond precision
  + DATE—a date
* Binary types
  + BINARY—a sequence of bytes

The Types are organized in the following hierarchy (where the parent is a super type of all the children instances):

* Type

|  |
| --- |
| Primitive Type |

|  |
| --- |
| Number |

|  |
| --- |
| DOUBLE |

|  |
| --- |
| FLOAT |

|  |
| --- |
| BIGINT |

|  |
| --- |
| INT |

|  |
| --- |
| SMALLINT |

|  |
| --- |
| TINYINT |

|  |
| --- |
| STRING |

|  |
| --- |
| BOOLEAN |

This type hierarchy defines how the types are implicitly converted in the query language. Implicit conversion is allowed for types from child to an ancestor. So when a query expression expects type1 and the data is of type2, type2 is implicitly converted to type1 if type1 is an ancestor of type2 in the type hierarchy. Note that the type hierarchy allows the implicit conversion of STRING to DOUBLE.

Explicit type conversion can be done using the cast operator as shown in the [#Built In Functions](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-BuiltInFunctions) section below.

### Complex Types

Complex Types can be built up from primitive types and other composite types using:

* Structs: the elements within the type can be accessed using the DOT (.) notation. For example, for a column c of type STRUCT {a INT; b INT}, the a field is accessed by the expression c.a
* Maps (key-value tuples): The elements are accessed using ['element name'] notation. For example in a map M comprising of a mapping from 'group' -> gid the gid value can be accessed using M['group']
* Arrays (indexable lists): The elements in the array have to be in the same type. Elements can be accessed using the [n] notation where n is an index (zero-based) into the array. For example, for an array A having the elements ['a', 'b', 'c'], A[1] retruns 'b'.

Using the primitive types and the constructs for creating complex types, types with arbitrary levels of nesting can be created. For example, a type User may comprise of the following fields:

* gender—which is a STRING.
* active—which is a BOOLEAN.

## Built In Operators and Functions

The operators and functions listed below are not necessarily up to date. ([Hive Operators and UDFs](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF) has more current information.) In [Beeline](https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients#HiveServer2Clients-Beeline–NewCommandLineShell) or the Hive [CLI](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Cli), use these commands to show the latest documentation:

|  |
| --- |
| SHOW FUNCTIONS;  DESCRIBE FUNCTION <function\_name>;  DESCRIBE FUNCTION EXTENDED <function\_name>; |

**Case-insensitive**

All Hive keywords are case-insensitive, including the names of Hive operators and functions.

### Built In Operators

* **Relational Operators**—The following operators compare the passed operands and generate a TRUE or FALSE value, depending on whether the comparison between the operands holds or not.

| **Relational Operator** | **Operand types** | **Description** |
| --- | --- | --- |
| A = B | all primitive types | TRUE if expression A is equivalent to expression B; otherwise FALSE |
| A != B | all primitive types | TRUE if expression A is not equivalent to expression B; otherwise FALSE |
| A < B | all primitive types | TRUE if expression A is less than expression B; otherwise FALSE |
| A <= B | all primitive types | TRUE if expression A is less than or equal to expression B; otherwise FALSE |
| A > B | all primitive types | TRUE if expression A is greater than expression B] otherwise FALSE |
| A >= B | all primitive types | TRUE if expression A is greater than or equal to expression B otherwise FALSE |
| A IS NULL | all types | TRUE if expression A evaluates to NULL otherwise FALSE |
| A IS NOT NULL | all types | FALSE if expression A evaluates to NULL otherwise TRUE |
| A LIKE B | strings | TRUE if string A matches the SQL simple regular expression B, otherwise FALSE. The comparison is done character by character. The \_ character in B matches any character in A (similar to **.** in posix regular expressions), and the % character in B matches an arbitrary number of characters in A (similar to **.\*** in posix regular expressions). For example, 'foobar' LIKE 'foo' evaluates to FALSE where as 'foobar' LIKE 'foo\_\_\_' evaluates to TRUE and so does 'foobar' LIKE 'foo%'. To escape % use \ (% matches one % character). If the data contains a semicolon, and you want to search for it, it needs to be escaped, columnValue LIKE 'a\;b' |
| A RLIKE B | strings | NULL if A or B is NULL, TRUE if any (possibly empty) substring of A matches the Java regular expression B (see [Java regular expressions syntax](http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html)), otherwise FALSE. For example, 'foobar' rlike 'foo' evaluates to TRUE and so does 'foobar' rlike '^f.\*r$'. |
| A REGEXP B | strings | Same as RLIKE |

* **Arithmetic Operators**—The following operators support various common arithmetic operations on the operands. All of them return number types.

| **Arithmetic Operators** | **Operand types** | **Description** |
| --- | --- | --- |
| A + B | all number types | Gives the result of adding A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands, for example, since every integer is a float. Therefore, float is a containing type of integer so the + operator on a float and an int will result in a float. |
| A - B | all number types | Gives the result of subtracting B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A \* B | all number types | Gives the result of multiplying A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. Note that if the multiplication causing overflow, you will have to cast one of the operators to a type higher in the type hierarchy. |
| A / B | all number types | Gives the result of dividing B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. If the operands are integer types, then the result is the quotient of the division. |
| A % B | all number types | Gives the reminder resulting from dividing A by B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A & B | all number types | Gives the result of bitwise AND of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A | B | all number types | Gives the result of bitwise OR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| A ^ B | all number types | Gives the result of bitwise XOR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. |
| ~A | all number types | Gives the result of bitwise NOT of A. The type of the result is the same as the type of A. |

* **Logical Operators** — The following operators provide support for creating logical expressions. All of them return boolean TRUE or FALSE depending upon the boolean values of the operands.

| **Logical Operators** | **Operands types** | **Description** |
| --- | --- | --- |
| A AND B | boolean | TRUE if both A and B are TRUE, otherwise FALSE |
| A && B | boolean | Same as A AND B |
| A OR B | boolean | TRUE if either A or B or both are TRUE, otherwise FALSE |
| A || B | boolean | Same as A OR B |
| NOT A | boolean | TRUE if A is FALSE, otherwise FALSE |
| !A | boolean | Same as NOT A |

* **Operators on Complex Types**—The following operators provide mechanisms to access elements in Complex Types

| **Operator** | **Operand types** | **Description** |
| --- | --- | --- |
| A[n] | A is an Array and n is an int | returns the nth element in the array A. The first element has index 0, for example, if A is an array comprising of ['foo', 'bar'] then A[0] returns 'foo' and A[1] returns 'bar' |
| M[key] | M is a Map<K, V> and key has type K | returns the value corresponding to the key in the map for example, if M is a map comprising of  {'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'} then M['all'] returns 'foobar' |
| S.x | S is a struct | returns the x field of S, for example, for struct foobar {int foo, int bar} foobar.foo returns the integer stored in the foo field of the struct. |

### Built In Functions

* Hive supports the following built in functions:  
  [(Function list in source code: FunctionRegistry.java)](http://svn.apache.org/viewvc/hive/trunk/ql/src/java/org/apache/hadoop/hive/ql/exec/FunctionRegistry.java?view=markup)

| **Return Type** | **Function Name (Signature)** | **Description** |
| --- | --- | --- |
| BIGINT | round(double a) | returns the rounded BIGINT value of the double |
| BIGINT | floor(double a) | returns the maximum BIGINT value that is equal or less than the double |
| BIGINT | ceil(double a) | returns the minimum BIGINT value that is equal or greater than the double |
| double | rand(), rand(int seed) | returns a random number (that changes from row to row). Specifiying the seed will make sure the generated random number sequence is deterministic. |
| string | concat(string A, string B,...) | returns the string resulting from concatenating B after A. For example, concat('foo', 'bar') results in 'foobar'. This function accepts arbitrary number of arguments and return the concatenation of all of them. |
| string | substr(string A, int start) | returns the substring of A starting from start position till the end of string A. For example, substr('foobar', 4) results in 'bar' |
| string | substr(string A, int start, int length) | returns the substring of A starting from start position with the given length, for example,  substr('foobar', 4, 2) results in 'ba' |
| string | upper(string A) | returns the string resulting from converting all characters of A to upper case, for example, upper('fOoBaR') results in 'FOOBAR' |
| string | ucase(string A) | Same as upper |
| string | lower(string A) | returns the string resulting from converting all characters of B to lower case, for example, lower('fOoBaR') results in 'foobar' |
| string | lcase(string A) | Same as lower |
| string | trim(string A) | returns the string resulting from trimming spaces from both ends of A, for example, trim(' foobar ') results in 'foobar' |
| string | ltrim(string A) | returns the string resulting from trimming spaces from the beginning(left hand side) of A. For example, ltrim(' foobar ') results in 'foobar ' |
| string | rtrim(string A) | returns the string resulting from trimming spaces from the end(right hand side) of A. For example, rtrim(' foobar ') results in ' foobar' |
| string | regexp\_replace(string A, string B, string C) | returns the string resulting from replacing all substrings in B that match the Java regular expression syntax(See [Java regular expressions syntax](http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html)) with C. For example, regexp\_replace('foobar', 'oo|ar', ) returns 'fb' |
| int | size(Map<K.V>) | returns the number of elements in the map type |
| int | size(Array<T>) | returns the number of elements in the array type |
| value of <type> | cast(<expr> as <type>) | converts the results of the expression expr to <type>, for example, cast('1' as BIGINT) will convert the string '1' to it integral representation. A null is returned if the conversion does not succeed. |
| string | from\_unixtime(int unixtime) | convert the number of seconds from the UNIX epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00" |
| string | to\_date(string timestamp) | Return the date part of a timestamp string: to\_date("1970-01-01 00:00:00") = "1970-01-01" |
| int | year(string date) | Return the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970 |
| int | month(string date) | Return the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11 |
| int | day(string date) | Return the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1 |
| string | get\_json\_object(string json\_string, string path) | Extract json object from a json string based on json path specified, and return json string of the extracted json object. It will return null if the input json string is invalid. |

* The following built in aggregate functions are supported in Hive:

| **Return Type** | **Aggregation Function Name (Signature)** | **Description** |
| --- | --- | --- |
| BIGINT | count(\*), count(expr), count(DISTINCT expr[, expr\_.]) | count(\*)—Returns the total number of retrieved rows, including rows containing NULL values; count(expr)—Returns the number of rows for which the supplied expression is non-NULL; count(DISTINCT expr[, expr])—Returns the number of rows for which the supplied expression(s) are unique and non-NULL. |
| DOUBLE | sum(col), sum(DISTINCT col) | returns the sum of the elements in the group or the sum of the distinct values of the column in the group |
| DOUBLE | avg(col), avg(DISTINCT col) | returns the average of the elements in the group or the average of the distinct values of the column in the group |
| DOUBLE | min(col) | returns the minimum value of the column in the group |
| DOUBLE | max(col) | returns the maximum value of the column in the group |

## Language Capabilities

[Hive's SQL](https://cwiki.apache.org/confluence/display/Hive/LanguageManual) provides the basic SQL operations. These operations work on tables or partitions. These operations are:

* Ability to filter rows from a table using a WHERE clause.
* Ability to select certain columns from the table using a SELECT clause.
* Ability to do equi-joins between two tables.
* Ability to evaluate aggregations on multiple "group by" columns for the data stored in a table.
* Ability to store the results of a query into another table.
* Ability to download the contents of a table to a local (for example,, nfs) directory.
* Ability to store the results of a query in a hadoop dfs directory.
* Ability to manage tables and partitions (create, drop and alter).
* Ability to plug in custom scripts in the language of choice for custom map/reduce jobs.

# **Usage and Examples**

**NOTE: Many of the following examples are out of date.  More up to date information can be found in the**[**LanguageManual**](https://cwiki.apache.org/confluence/display/Hive/LanguageManual)**.**

The following examples highlight some salient features of the system. A detailed set of query test cases can be found at [Hive Query Test Cases](http://svn.apache.org/viewvc/hive/trunk/ql/src/test/queries/clientpositive/) and the corresponding results can be found at [Query Test Case Results](http://svn.apache.org/viewvc/hive/trunk/ql/src/test/results/clientpositive/).

* [Creating, Showing, Altering, and Dropping Tables](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Creating,Showing,Altering,andDroppingTables)
* [Loading Data](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-LoadingData)
* [Querying and Inserting Data](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-QueryingandInsertingData)

## Creating, Showing, Altering, and Dropping Tables

See [Hive Data Definition Language](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL) for detailed information about creating, showing, altering, and dropping tables.

### Creating Tables

An example statement that would create the page\_view table mentioned above would be like:

|  |
| --- |
| CREATE TABLE page\_view(viewTime INT, userid BIGINT,                  page\_url STRING, referrer\_url STRING,                  ip STRING COMMENT 'IP Address of the User')  COMMENT 'This is the page view table'  PARTITIONED BY(dt STRING, country STRING)  STORED AS SEQUENCEFILE; |

In this example, the columns of the table are specified with the corresponding types. Comments can be attached both at the column level as well as at the table level. Additionally, the partitioned by clause defines the partitioning columns which are different from the data columns and are actually not stored with the data. When specified in this way, the data in the files is assumed to be delimited with ASCII 001(ctrl-A) as the field delimiter and newline as the row delimiter.

The field delimiter can be parametrized if the data is not in the above format as illustrated in the following example:

|  |
| --- |
| CREATE TABLE page\_view(viewTime INT, userid BIGINT,                  page\_url STRING, referrer\_url STRING,                  ip STRING COMMENT 'IP Address of the User')  COMMENT 'This is the page view table'  PARTITIONED BY(dt STRING, country STRING)  ROW FORMAT DELIMITED          FIELDS TERMINATED BY '1'  STORED AS SEQUENCEFILE; |

The row delimintor currently cannot be changed since it is not determined by Hive but Hadoop delimiters.

It is also a good idea to bucket the tables on certain columns so that efficient sampling queries can be executed against the data set. If bucketing is absent, random sampling can still be done on the table but it is not efficient as the query has to scan all the data. The following example illustrates the case of the page\_view table that is bucketed on the userid column:

|  |
| --- |
| CREATE TABLE page\_view(viewTime INT, userid BIGINT,                  page\_url STRING, referrer\_url STRING,                  ip STRING COMMENT 'IP Address of the User')  COMMENT 'This is the page view table'  PARTITIONED BY(dt STRING, country STRING)  CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  ROW FORMAT DELIMITED          FIELDS TERMINATED BY '1'          COLLECTION ITEMS TERMINATED BY '2'          MAP KEYS TERMINATED BY '3'  STORED AS SEQUENCEFILE; |

In the example above, the table is clustered by a hash function of userid into 32 buckets. Within each bucket the data is sorted in increasing order of viewTime. Such an organization allows the user to do efficient sampling on the clustered column—n this case userid. The sorting property allows internal operators to take advantage of the better-known data structure while evaluating queries with greater efficiency.

|  |
| --- |
| CREATE TABLE page\_view(viewTime INT, userid BIGINT,                  page\_url STRING, referrer\_url STRING,                  friends ARRAY<BIGINT>, properties MAP<STRING, STRING>                  ip STRING COMMENT 'IP Address of the User')  COMMENT 'This is the page view table'  PARTITIONED BY(dt STRING, country STRING)  CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  ROW FORMAT DELIMITED          FIELDS TERMINATED BY '1'          COLLECTION ITEMS TERMINATED BY '2'          MAP KEYS TERMINATED BY '3'  STORED AS SEQUENCEFILE; |

In this example, the columns that comprise of the table row are specified in a similar way as the definition of types. Comments can be attached both at the column level as well as at the table level. Additionally, the partitioned by clause defines the partitioning columns which are different from the data columns and are actually not stored with the data. The CLUSTERED BY clause specifies which column to use for bucketing as well as how many buckets to create. The delimited row format specifies how the rows are stored in the hive table. In the case of the delimited format, this specifies how the fields are terminated, how the items within collections (arrays or maps) are terminated, and how the map keys are terminated. STORED AS SEQUENCEFILE indicates that this data is stored in a binary format (using hadoop SequenceFiles) on hdfs. The values shown for the ROW FORMAT and STORED AS clauses in the above, example represent the system defaults.

Table names and column names are case insensitive.

### Browsing Tables and Partitions

|  |
| --- |
| SHOW TABLES; |

To list existing tables in the warehouse; there are many of these, likely more than you want to browse.

|  |
| --- |
| SHOW TABLES 'page.\*'; |

To list tables with prefix 'page'. The pattern follows Java regular expression syntax (so the period is a wildcard).

|  |
| --- |
| SHOW PARTITIONS page\_view; |

To list partitions of a table. If the table is not a partitioned table then an error is thrown.

|  |
| --- |
| DESCRIBE page\_view; |

To list columns and column types of table.

|  |
| --- |
| DESCRIBE EXTENDED page\_view; |

To list columns and all other properties of table. This prints lot of information and that too not in a pretty format. Usually used for debugging.

|  |
| --- |
| DESCRIBE EXTENDED page\_view PARTITION (ds='2008-08-08'); |

To list columns and all other properties of a partition. This also prints lot of information which is usually used for debugging.

### Altering Tables

To rename existing table to a new name. If a table with new name already exists then an error is returned:

|  |
| --- |
| ALTER TABLE old\_table\_name RENAME TO new\_table\_name; |

To rename the columns of an existing table. Be sure to use the same column types, and to include an entry for each preexisting column:

|  |
| --- |
| ALTER TABLE old\_table\_name REPLACE COLUMNS (col1 TYPE, ...); |

To add columns to an existing table:

|  |
| --- |
| ALTER TABLE tab1 ADD COLUMNS (c1 INT COMMENT 'a new int column', c2 STRING DEFAULT 'def val'); |

Note that a change in the schema (such as the adding of the columns), preserves the schema for the old partitions of the table in case it is a partitioned table. All the queries that access these columns and run over the old partitions implicitly return a null value or the specified default values for these columns.

In the later versions, we can make the behavior of assuming certain values as opposed to throwing an error in case the column is not found in a particular partition configurable.

### Dropping Tables and Partitions

Dropping tables is fairly trivial. A drop on the table would implicitly drop any indexes(this is a future feature) that would have been built on the table. The associated command is:

|  |
| --- |
| DROP TABLE pv\_users; |

To dropping a partition. Alter the table to drop the partition.

|  |
| --- |
| ALTER TABLE pv\_users DROP PARTITION (ds='2008-08-08') |

* Note that any data for this table or partitions will be dropped and may not be recoverable. \*

## Loading Data

There are multiple ways to load data into Hive tables. The user can create an external table that points to a specified location within [HDFS](http://hadoop.apache.org/common/docs/current/hdfs_design.html). In this particular usage, the user can copy a file into the specified location using the HDFS put or copy commands and create a table pointing to this location with all the relevant row format information. Once this is done, the user can transform the data and insert them into any other Hive table. For example, if the file /tmp/pv\_2008-06-08.txt contains comma separated page views served on 2008-06-08, and this needs to be loaded into the page\_view table in the appropriate partition, the following sequence of commands can achieve this:

|  |
| --- |
| CREATE EXTERNAL TABLE page\_view\_stg(viewTime INT, userid BIGINT,                  page\_url STRING, referrer\_url STRING,                  ip STRING COMMENT 'IP Address of the User',                  country STRING COMMENT 'country of origination')  COMMENT 'This is the staging page view table'  ROW FORMAT DELIMITED FIELDS TERMINATED BY '44' LINES TERMINATED BY '12'  STORED AS TEXTFILE  LOCATION '/user/data/staging/page\_view';    hadoop dfs -put /tmp/pv\_2008-06-08.txt /user/data/staging/page\_view    FROM page\_view\_stg pvs  INSERT OVERWRITE TABLE page\_view PARTITION(dt='2008-06-08', country='US')  SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip  WHERE pvs.country = 'US'; |

\* This code results in an error due to LINES TERMINATED BY limitation

FAILED: SemanticException 6:67 LINES TERMINATED BY only supports newline '\n' right now. Error encountered near token ''12''

See [[https://issues.apache.org/jira/images/icons/issuetypes/improvement.png](https://issues.apache.org/jira/browse/HIVE-5999)HIVE-5999](https://issues.apache.org/jira/browse/HIVE-5999) - Allow other characters for LINES TERMINATED BY **OPEN**[[https://issues.apache.org/jira/images/icons/issuetypes/bug.png](https://issues.apache.org/jira/browse/HIVE-11996)HIVE-11996](https://issues.apache.org/jira/browse/HIVE-11996) - Row Delimiter other than '\n' throws error in Hive. **OPEN**

In the example above, nulls are inserted for the array and map types in the destination tables but potentially these can also come from the external table if the proper row formats are specified.

This method is useful if there is already legacy data in HDFS on which the user wants to put some metadata so that the data can be queried and manipulated using Hive.

Additionally, the system also supports syntax that can load the data from a file in the local files system directly into a Hive table where the input data format is the same as the table format. If /tmp/pv\_2008-06-08\_us.txt already contains the data for US, then we do not need any additional filtering as shown in the previous example. The load in this case can be done using the following syntax:

|  |
| --- |
| LOAD DATA LOCAL INPATH /tmp/pv\_2008-06-08\_us.txt INTO TABLE page\_view PARTITION(date='2008-06-08', country='US') |

The path argument can take a directory (in which case all the files in the directory are loaded), a single file name, or a wildcard (in which case all the matching files are uploaded). If the argument is a directory, it cannot contain subdirectories. Similarly, the wildcard must match file names only.

In the case that the input file /tmp/pv\_2008-06-08\_us.txt is very large, the user may decide to do a parallel load of the data (using tools that are external to Hive). Once the file is in HDFS - the following syntax can be used to load the data into a Hive table:

|  |
| --- |
| LOAD DATA INPATH '/user/data/pv\_2008-06-08\_us.txt' INTO TABLE page\_view PARTITION(date='2008-06-08', country='US') |

It is assumed that the array and map fields in the input.txt files are null fields for these examples.

See [Hive Data Manipulation Language](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML) for more information about loading data into Hive tables, and see [External Tables](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-ExternalTables) for another example of creating an external table.

## Querying and Inserting Data

* [Simple Query](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-SimpleQuery)
* [Partition Based Query](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-PartitionBasedQuery)
* [Joins](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Joins)
* [Aggregations](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Aggregations)
* [Multi Table/File Inserts](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-MultiTable/FileInserts)
* [Dynamic-Partition Insert](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Dynamic-PartitionInsert)
* [Inserting into Local Files](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-InsertingintoLocalFiles)
* [Sampling](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Sampling)
* [Union All](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-UnionAll)
* [Array Operations](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-ArrayOperations)
* [Map (Associative Arrays) Operations](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Map(AssociativeArrays)Operations)
* [Custom Map/Reduce Scripts](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-CustomMap/ReduceScripts)
* [Co-Groups](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Co-Groups)

The Hive query operations are documented in [Select](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Select), and the insert operations are documented in [Inserting data into Hive Tables from queries](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-InsertingdataintoHiveTablesfromqueries) and [Writing data into the filesystem from queries](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-Writingdataintothefilesystemfromqueries).

### Simple Query

For all the active users, one can use the query of the following form:

|  |
| --- |
| INSERT OVERWRITE TABLE user\_active  SELECT user.\*  FROM user  WHERE user.active = 1; |

Note that unlike SQL, we always insert the results into a table. We will illustrate later how the user can inspect these results and even dump them to a local file. You can also run the following query in [Beeline](https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients#HiveServer2Clients-Beeline–NewCommandLineShell) or the Hive [CLI](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Cli):

|  |
| --- |
| SELECT user.\*  FROM user  WHERE user.active = 1; |

This will be internally rewritten to some temporary file and displayed to the Hive client side.

### Partition Based Query

What partitions to use in a query is determined automatically by the system on the basis of where clause conditions on partition columns. For example, in order to get all the page\_views in the month of 03/2008 referred from domain xyz.com, one could write the following query:

|  |
| --- |
| INSERT OVERWRITE TABLE xyz\_com\_page\_views  SELECT page\_views.\*  FROM page\_views  WHERE page\_views.date >= '2008-03-01' AND page\_views.date <= '2008-03-31' AND        page\_views.referrer\_url like '%xyz.com'; |

Note that page\_views.date is used here because the table (above) was defined with PARTITIONED BY(date DATETIME, country STRING) ; if you name your partition something different, don't expect .date to do what you think!

### Joins

In order to get a demographic breakdown (by gender) of page\_view of 2008-03-03 one would need to join the page\_view table and the user table on the userid column. This can be accomplished with a join as shown in the following query:

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT pv.\*, u.gender, u.age  FROM user u JOIN page\_view pv ON (pv.userid = u.id)  WHERE pv.date = '2008-03-03'; |

In order to do outer joins the user can qualify the join with LEFT OUTER, RIGHT OUTER or FULL OUTER keywords in order to indicate the kind of outer join (left preserved, right preserved or both sides preserved). For example, in order to do a full outer join in the query above, the corresponding syntax would look like the following query:

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT pv.\*, u.gender, u.age  FROM user u FULL OUTER JOIN page\_view pv ON (pv.userid = u.id)  WHERE pv.date = '2008-03-03'; |

In order check the existence of a key in another table, the user can use LEFT SEMI JOIN as illustrated by the following example.

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_users  SELECT u.\*  FROM user u LEFT SEMI JOIN page\_view pv ON (pv.userid = u.id)  WHERE pv.date = '2008-03-03'; |

In order to join more than one tables, the user can use the following syntax:

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_friends  SELECT pv.\*, u.gender, u.age, f.friends  FROM page\_view pv JOIN user u ON (pv.userid = u.id) JOIN friend\_list f ON (u.id = f.uid)  WHERE pv.date = '2008-03-03'; |

Note that Hive only supports [equi-joins](http://en.wikipedia.org/wiki/Join_(SQL)#Equi-join). Also it is best to put the largest table on the rightmost side of the join to get the best performance.

### Aggregations

In order to count the number of distinct users by gender one could write the following query:

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_gender\_sum  SELECT pv\_users.gender, count (DISTINCT pv\_users.userid)  FROM pv\_users  GROUP BY pv\_users.gender; |

Multiple aggregations can be done at the same time, however, no two aggregations can have different DISTINCT columns .e.g while the following is possible

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_gender\_agg  SELECT pv\_users.gender, count(DISTINCT pv\_users.userid), count(\*), sum(DISTINCT pv\_users.userid)  FROM pv\_users  GROUP BY pv\_users.gender; |

however, the following query is not allowed

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_gender\_agg  SELECT pv\_users.gender, count(DISTINCT pv\_users.userid), count(DISTINCT pv\_users.ip)  FROM pv\_users  GROUP BY pv\_users.gender; |

### Multi Table/File Inserts

The output of the aggregations or simple selects can be further sent into multiple tables or even to hadoop dfs files (which can then be manipulated using hdfs utilities). For example, if along with the gender breakdown, one needed to find the breakdown of unique page views by age, one could accomplish that with the following query:

|  |
| --- |
| FROM pv\_users  INSERT OVERWRITE TABLE pv\_gender\_sum      SELECT pv\_users.gender, count\_distinct(pv\_users.userid)      GROUP BY pv\_users.gender    INSERT OVERWRITE DIRECTORY '/user/data/tmp/pv\_age\_sum'      SELECT pv\_users.age, count\_distinct(pv\_users.userid)      GROUP BY pv\_users.age; |

The first insert clause sends the results of the first group by to a Hive table while the second one sends the results to a hadoop dfs files.

### Dynamic-Partition Insert

In the previous examples, the user has to know which partition to insert into and only one partition can be inserted in one insert statement. If you want to load into multiple partitions, you have to use multi-insert statement as illustrated below.

|  |
| --- |
| FROM page\_view\_stg pvs  INSERT OVERWRITE TABLE page\_view PARTITION(dt='2008-06-08', country='US')         SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip WHERE pvs.country = 'US'  INSERT OVERWRITE TABLE page\_view PARTITION(dt='2008-06-08', country='CA')         SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip WHERE pvs.country = 'CA'  INSERT OVERWRITE TABLE page\_view PARTITION(dt='2008-06-08', country='UK')         SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip WHERE pvs.country = 'UK'; |

In order to load data into all country partitions in a particular day, you have to add an insert statement for each country in the input data. This is very inconvenient since you have to have the priori knowledge of the list of countries exist in the input data and create the partitions beforehand. If the list changed for another day, you have to modify your insert DML as well as the partition creation DDLs. It is also inefficient since each insert statement may be turned into a MapReduce Job.

[*Dynamic-partition insert*](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-DynamicPartitionInserts) (or multi-partition insert) is designed to solve this problem by dynamically determining which partitions should be created and populated while scanning the input table. This is a newly added feature that is only available from version 0.6.0. In the dynamic partition insert, the input column values are evaluated to determine which partition this row should be inserted into. If that partition has not been created, it will create that partition automatically. Using this feature you need only one insert statement to create and populate all necessary partitions. In addition, since there is only one insert statement, there is only one corresponding MapReduce job. This significantly improves performance and reduce the Hadoop cluster workload comparing to the multiple insert case.

Below is an example of loading data to all country partitions using one insert statement:

|  |
| --- |
| FROM page\_view\_stg pvs  INSERT OVERWRITE TABLE page\_view PARTITION(dt='2008-06-08', country)         SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip, pvs.country |

There are several syntactic differences from the multi-insert statement:

* country appears in the PARTITION specification, but with no value associated. In this case, country is a dynamic partition column. On the other hand, ds has a value associated with it, which means it is a static partition column. If a column is dynamic partition column, its value will be coming from the input column. Currently we only allow dynamic partition columns to be the last column(s) in the partition clause because the partition column order indicates its hierarchical order (meaning dt is the root partition, and country is the child partition). You cannot specify a partition clause with (dt, country='US') because that means you need to update all partitions with any date and its country sub-partition is 'US'.
* An additional pvs.country column is added in the select statement. This is the corresponding input column for the dynamic partition column. Note that you do not need to add an input column for the static partition column because its value is already known in the PARTITION clause. Note that the dynamic partition values are selected by ordering, not name, and taken as the last columns from the select clause.

Semantics of the dynamic partition insert statement:

* When there are already non-empty partitions exists for the dynamic partition columns, (for example, country='CA' exists under some ds root partition), it will be overwritten if the dynamic partition insert saw the same value (say 'CA') in the input data. This is in line with the 'insert overwrite' semantics. However, if the partition value 'CA' does not appear in the input data, the existing partition will not be overwritten.
* Since a Hive partition corresponds to a directory in HDFS, the partition value has to conform to the HDFS path format (URI in Java). Any character having a special meaning in URI (for example, '%', ':', '/', '#') will be escaped with '%' followed by 2 bytes of its ASCII value.
* If the input column is a type different than STRING, its value will be first converted to STRING to be used to construct the HDFS path.
* If the input column value is NULL or empty string, the row will be put into a special partition, whose name is controlled by the hive parameter hive.exec.default.partition.name. The default value is HIVE\_DEFAULT\_PARTITION{}. Basically this partition will contain all "bad" rows whose value are not valid partition names. The caveat of this approach is that the bad value will be lost and is replaced by HIVE\_DEFAULT\_PARTITION{} if you select them Hive. JIRA HIVE-1309 is a solution to let user specify "bad file" to retain the input partition column values as well.
* Dynamic partition insert could potentially be a resource hog in that it could generate a large number of partitions in a short time. To get yourself buckled, we define three parameters:
  + **hive.exec.max.dynamic.partitions.pernode** (default value being 100) is the maximum dynamic partitions that can be created by each mapper or reducer. If one mapper or reducer created more than that the threshold, a fatal error will be raised from the mapper/reducer (through counter) and the whole job will be killed.
  + **hive.exec.max.dynamic.partitions** (default value being 1000) is the total number of dynamic partitions could be created by one DML. If each mapper/reducer did not exceed the limit but the total number of dynamic partitions does, then an exception is raised at the end of the job before the intermediate data are moved to the final destination.
  + **hive.exec.max.created.files** (default value being 100000) is the maximum total number of files created by all mappers and reducers. This is implemented by updating a Hadoop counter by each mapper/reducer whenever a new file is created. If the total number is exceeding hive.exec.max.created.files, a fatal error will be thrown and the job will be killed.
* Another situation we want to protect against dynamic partition insert is that the user may accidentally specify all partitions to be dynamic partitions without specifying one static partition, while the original intention is to just overwrite the sub-partitions of one root partition. We define another parameter hive.exec.dynamic.partition.mode=strict to prevent the all-dynamic partition case. In the strict mode, you have to specify at least one static partition. The default mode is strict. In addition, we have a parameter hive.exec.dynamic.partition=true/false to control whether to allow dynamic partition at all. The default value is false prior to Hive 0.9.0 and true in Hive 0.9.0 and later.
* In Hive 0.6, dynamic partition insert does not work with hive.merge.mapfiles=true or hive.merge.mapredfiles=true, so it internally turns off the merge parameters. Merging files in dynamic partition inserts are supported in Hive 0.7 (see JIRA HIVE-1307 for details).

Troubleshooting and best practices:

* As stated above, there are too many dynamic partitions created by a particular mapper/reducer, a fatal error could be raised and the job will be killed. The error message looks something like:

|  |
| --- |
| beeline> set hive.exec.dynamic.partition.mode=nonstrict;      beeline> FROM page\_view\_stg pvs            INSERT OVERWRITE TABLE page\_view PARTITION(dt, country)                   SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip,                          from\_unixtimestamp(pvs.viewTime, 'yyyy-MM-dd') ds, pvs.country;  ...  2010-05-07 11:10:19,816 Stage-1 map = 0%,  reduce = 0%  [Fatal Error] Operator FS\_28 (id=41): fatal error. Killing the job.  Ended Job = job\_201005052204\_28178 with errors  ... |

* The problem of this that one mapper will take a random set of rows and it is very likely that the number of distinct (dt, country) pairs will exceed the limit of hive.exec.max.dynamic.partitions.pernode. One way around it is to group the rows by the dynamic partition columns in the mapper and distribute them to the reducers where the dynamic partitions will be created. In this case the number of distinct dynamic partitions will be significantly reduced. The above example query could be rewritten to:

|  |
| --- |
| beeline> set hive.exec.dynamic.partition.mode=nonstrict;  beeline> FROM page\_view\_stg pvs        INSERT OVERWRITE TABLE page\_view PARTITION(dt, country)               SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip,                      from\_unixtimestamp(pvs.viewTime, 'yyyy-MM-dd') ds, pvs.country               DISTRIBUTE BY ds, country; |

* This query will generate a MapReduce job rather than Map-only job. The SELECT-clause will be converted to a plan to the mappers and the output will be distributed to the reducers based on the value of (ds, country) pairs. The INSERT-clause will be converted to the plan in the reducer which writes to the dynamic partitions.

Additional documentation:

* [Design Document for Dynamic Partitions](https://cwiki.apache.org/confluence/display/Hive/DynamicPartitions)
  + [Original design doc](https://issues.apache.org/jira/secure/attachment/12437909/dp_design.txt)
  + [HIVE-936](https://issues.apache.org/jira/browse/HIVE-936)
* [Hive DML: Dynamic Partition Inserts](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-DynamicPartitionInserts)
* [HCatalog Dynamic Partitioning](https://cwiki.apache.org/confluence/display/Hive/HCatalog+DynamicPartitions)
  + [Usage with Pig](https://cwiki.apache.org/confluence/display/Hive/HCatalog+DynamicPartitions#HCatalogDynamicPartitions-UsagewithPig)
  + [Usage from MapReduce](https://cwiki.apache.org/confluence/display/Hive/HCatalog+DynamicPartitions#HCatalogDynamicPartitions-UsagefromMapReduce)

### Inserting into Local Files

In certain situations you would want to write the output into a local file so that you could load it into an excel spreadsheet. This can be accomplished with the following command:

|  |
| --- |
| INSERT OVERWRITE LOCAL DIRECTORY '/tmp/pv\_gender\_sum'  SELECT pv\_gender\_sum.\*  FROM pv\_gender\_sum; |

### Sampling

The sampling clause allows the users to write queries for samples of the data instead of the whole table. Currently the sampling is done on the columns that are specified in the CLUSTERED BY clause of the CREATE TABLE statement. In the following example we choose 3rd bucket out of the 32 buckets of the pv\_gender\_sum table:

|  |
| --- |
| INSERT OVERWRITE TABLE pv\_gender\_sum\_sample  SELECT pv\_gender\_sum.\*  FROM pv\_gender\_sum TABLESAMPLE(BUCKET 3 OUT OF 32); |

In general the TABLESAMPLE syntax looks like:

|  |
| --- |
| TABLESAMPLE(BUCKET x OUT OF y) |

y has to be a multiple or divisor of the number of buckets in that table as specified at the table creation time. The buckets chosen are determined if bucket\_number module y is equal to x. So in the above example the following tablesample clause

|  |
| --- |
| TABLESAMPLE(BUCKET 3 OUT OF 16) |

would pick out the 3rd and 19th buckets. The buckets are numbered starting from 0.

On the other hand the tablesample clause

|  |
| --- |
| TABLESAMPLE(BUCKET 3 OUT OF 64 ON userid) |

would pick out half of the 3rd bucket.

### Union All

The language also supports union all, for example, if we suppose there are two different tables that track which user has published a video and which user has published a comment, the following query joins the results of a union all with the user table to create a single annotated stream for all the video publishing and comment publishing events:

|  |
| --- |
| INSERT OVERWRITE TABLE actions\_users  SELECT u.id, actions.date  FROM (      SELECT av.uid AS uid      FROM action\_video av      WHERE av.date = '2008-06-03'        UNION ALL        SELECT ac.uid AS uid      FROM action\_comment ac      WHERE ac.date = '2008-06-03'      ) actions JOIN users u ON(u.id = actions.uid); |

### Array Operations

Array columns in tables can be as follows:

|  |
| --- |
| CREATE TABLE array\_table (int\_array\_column ARRAY<INT>); |

Assuming that pv.friends is of the type ARRAY<INT> (i.e. it is an array of integers), the user can get a specific element in the array by its index as shown in the following command:

|  |
| --- |
| SELECT pv.friends[2]  FROM page\_views pv; |

The select expression gets the third item in the pv.friends array.

The user can also get the length of the array using the size function as shown below:

|  |
| --- |
| SELECT pv.userid, size(pv.friends)  FROM page\_view pv; |

### Map (Associative Arrays) Operations

Maps provide collections similar to associative arrays. Such structures can only be created programmatically currently. We will be extending this soon. For the purpose of the current example assume that pv.properties is of the type map<String, String> i.e. it is an associative array from strings to string. Accordingly, the following query:

|  |
| --- |
| INSERT OVERWRITE page\_views\_map  SELECT pv.userid, pv.properties['page type']  FROM page\_views pv; |

can be used to select the 'page\_type' property from the page\_views table.

Similar to arrays, the size function can also be used to get the number of elements in a map as shown in the following query:

|  |
| --- |
| SELECT size(pv.properties)  FROM page\_view pv; |

### Custom Map/Reduce Scripts

Users can also plug in their own custom mappers and reducers in the data stream by using features natively supported in the Hive language. for example, in order to run a custom mapper script - map\_script - and a custom reducer script - reduce\_script - the user can issue the following command which uses the TRANSFORM clause to embed the mapper and the reducer scripts.

Note that columns will be transformed to string and delimited by TAB before feeding to the user script, and the standard output of the user script will be treated as TAB-separated string columns. User scripts can output debug information to standard error which will be shown on the task detail page on hadoop.

|  |
| --- |
| FROM (       FROM pv\_users       MAP pv\_users.userid, pv\_users.date       USING 'map\_script'       AS dt, uid       CLUSTER BY dt) map\_output     INSERT OVERWRITE TABLE pv\_users\_reduced       REDUCE map\_output.dt, map\_output.uid       USING 'reduce\_script'       AS date, count; |

Sample map script (weekday\_mapper.py )

|  |
| --- |
| import sys  import datetime    for line in sys.stdin:    line = line.strip()    userid, unixtime = line.split('\t')    weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()    print ','.join([userid, str(weekday)]) |

Of course, both MAP and REDUCE are "syntactic sugar" for the more general select transform. The inner query could also have been written as such:

|  |
| --- |
| SELECT TRANSFORM(pv\_users.userid, pv\_users.date) USING 'map\_script' AS dt, uid CLUSTER BY dt FROM pv\_users; |

Schema-less map/reduce: If there is no "AS" clause after "USING map\_script", Hive assumes the output of the script contains 2 parts: key which is before the first tab, and value which is the rest after the first tab. Note that this is different from specifying "AS key, value" because in that case value will only contains the portion between the first tab and the second tab if there are multiple tabs.

In this way, we allow users to migrate old map/reduce scripts without knowing the schema of the map output. User still needs to know the reduce output schema because that has to match what is in the table that we are inserting to.

|  |
| --- |
| FROM (      FROM pv\_users      MAP pv\_users.userid, pv\_users.date      USING 'map\_script'      CLUSTER BY key) map\_output    INSERT OVERWRITE TABLE pv\_users\_reduced        REDUCE map\_output.dt, map\_output.uid      USING 'reduce\_script'      AS date, count; |

Distribute By and Sort By: Instead of specifying "cluster by", the user can specify "distribute by" and "sort by", so the partition columns and sort columns can be different. The usual case is that the partition columns are a prefix of sort columns, but that is not required.

|  |
| --- |
| FROM (      FROM pv\_users      MAP pv\_users.userid, pv\_users.date      USING 'map\_script'      AS c1, c2, c3      DISTRIBUTE BY c2      SORT BY c2, c1) map\_output    INSERT OVERWRITE TABLE pv\_users\_reduced        REDUCE map\_output.c1, map\_output.c2, map\_output.c3      USING 'reduce\_script'      AS date, count; |

### Co-Groups

Amongst the user community using map/reduce, cogroup is a fairly common operation wherein the data from multiple tables are sent to a custom reducer such that the rows are grouped by the values of certain columns on the tables. With the UNION ALL operator and the CLUSTER BY specification, this can be achieved in the Hive query language in the following way. Suppose we wanted to cogroup the rows from the actions\_video and action\_comments table on the uid column and send them to the 'reduce\_script' custom reducer, the following syntax can be used by the user:

|  |
| --- |
| FROM (       FROM (               FROM action\_video av               SELECT av.uid AS uid, av.id AS id, av.date AS date                UNION ALL                 FROM action\_comment ac               SELECT ac.uid AS uid, ac.id AS id, ac.date AS date       ) union\_actions       SELECT union\_actions.uid, union\_actions.id, union\_actions.date       CLUSTER BY union\_actions.uid) map     INSERT OVERWRITE TABLE actions\_reduced       SELECT TRANSFORM(map.uid, map.id, map.date) USING 'reduce\_script' AS (uid, id, reduced\_val); |

 =======================================

# **What is HIVE**

Hive is a data ware house system for Hadoop. It runs SQL like queries called HQL (Hive query language) which gets internally converted to map reduce jobs. Hive was developed by Facebook. Hive supports Data definition Language(DDL), Data Manipulation Language(DML) and user defined functions.

**DDL:** create table, create index, create views.

**DML:** Select, Where, group by, Join, Order By

Pluggable Functions:

**UDF:**User Defined Function  
**UDAF:** User Defined Aggregate Function  
**UDTF:**User Defined Table Function

## What HIVE is not

* Hive is not RDBMS.
* Hive is not used for OLTP(Online Transaction Processing).
* Even with small amount of data time to return the response can?t be compared to RDBMS.

## Points to remember

* Hive's metastore is used to persist schema i.e. table definition(table name, columns, types), location of table files, row format of table files, storage format of files.
* Hive Query Language is similar to SQL and gets reduced to map reduce jobs in backend.
* Hive's default database is derby.

# **HIVE Data Types**

Hive data types are categorized in numeric types, string types, misc types and complex types. A list of Hive data types are given below.

## Numeric Types

**TINYINT** (1-byte signed integer, from -128 to 127)

**SMALLINT** (2-byte signed integer, from -32,768 to 32,767)

**INT** (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)

**BIGINT** (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

**FLOAT** (4-byte single precision floating point number)

**DOUBLE** (8-byte double precision floating point number)

**DECIMAL** (Hive 0.13.0 introduced user definable precision and scale)

## Date/Time Types

TIMESTAMP  
DATE

## String Types

STRING  
VARCHAR  
CHAR

## Misc Types

BOOLEAN  
BINARY

## Complex Type

**arrays:** ARRAY<data\_type>

**maps:** MAP<primitive\_type, data\_type>

**structs:** STRUCT<col\_name : data\_type [COMMENT col\_comment], ...>

**union:** UNIONTYPE<data\_type, data\_type, ...>

# **Hive Commands**

Hive supports Data definition Language(DDL), Data Manipulation Language(DML) and User defined functions.

## Hive DDL Commands

create database

drop database

create table

drop table

alter table

create index

create views

## Hive DML Commands

Select

Where

Group By

Order By

Load Data

Join:

* Inner Join
* Left Outer Join
* Right Outer Join
* Full Outer Join

# **Hive DDL Commands**

Create Database Statement

A database in Hive is a namespace or a collection of tables.

1. hive**>** CREATE SCHEMA userdb;
2. hive**>** SHOW DATABASES;

Drop database

1. hive**>** DROP DATABASE IF EXISTS userdb;

Creating Hive Tables

Create a table called Sonoo with two columns, the first being an integer and the other a string.

1. hive**>** CREATE TABLE Sonoo(foo INT, bar STRING);

Create a table called HIVE\_TABLE with two columns and a partition column called ds. The partition column is a virtual column. It is not part of the data itself but is derived from the partition that a particular dataset is loaded into.By default, tables are assumed to be of text input format and the delimiters are assumed to be ^A(ctrl-a).

1. hive**>** CREATE TABLE HIVE\_TABLE (foo INT, bar STRING) PARTITIONED BY (ds STRING);

Browse the table

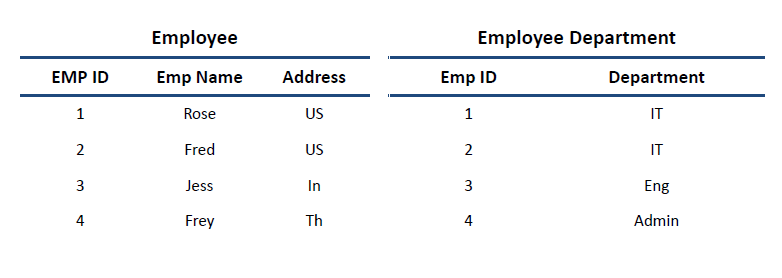
1. hive**>**  Show tables;

Altering and Dropping Tables

1. hive**>** ALTER TABLE Sonoo RENAME TO Kafka;
2. hive**>** ALTER TABLE Kafka ADD COLUMNS (col INT);
3. hive**>** ALTER TABLE HIVE\_TABLE ADD COLUMNS (col1 INT COMMENT 'a comment');
4. hive**>** ALTER TABLE HIVE\_TABLE REPLACE COLUMNS (col2 INT, weight STRING, baz INT COMMENT 'baz replaces new\_col1');

# **Hive DML Commands**

To understand the Hive DML commands, let's see the employee and employee\_department table first.



LOAD DATA

1. hive**>** LOAD DATA LOCAL INPATH './usr/Desktop/kv1.txt' OVERWRITE INTO TABLE Employee;

SELECTS and FILTERS

1. hive**>** SELECT  E.EMP\_ID FROM Employee E  WHERE E.Address='US';

GROUP BY

1. hive**>** hive**>** SELECT  E.EMP\_ID FROM Employee E GROUP BY E.Addresss;

# **Hive Sort By vs Order By**

Hive sort by and order by commands are used to fetch data in sorted order. The main differences between sort by and order by commands are given below.

**Sort by**

1. hive**>** SELECT  E.EMP\_ID FROM Employee E SORT BY E.empid;

May use multiple reducers for final output.

Only guarantees ordering of rows within a reducer.

May give partially ordered result.

**Order by**

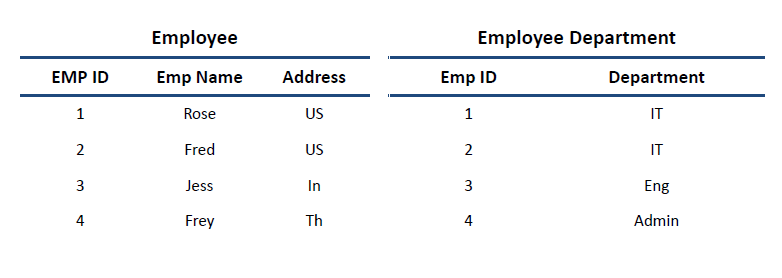
1. hive**>** SELECT  E.EMP\_ID FROM Employee E order BY E.empid;

Uses single reducer to guarantee total order in output.

LIMIT can be used to minimize sort time.

# **Hive Join**

Let's see two tables Employee and EmployeeDepartment that are going to be joined.



**Inner joins**

1. Select  \*  from employee join employeedepartment  ON (employee.empid=employeedepartment.empId)

Output : <<InnerJoin.png>>

**Left outer joins**

1. Select e.empId, empName, department from employee e Left outer join employeedepartment ed on(e.empId=ed.empId);

Output : <<LeftOuterJoin.png>>

**Right outer joins**

1. Select e.empId, empName, department from employee e Right outer join employeedepartment ed on(e.empId=ed.empId);

**Full outer joins**

1. Select e.empId, empName, department from employee e FULL outer join employeedepartment ed on(e.empId=ed.empId);

Buckting

Metastore

Serde

Custom serde

As select

Subqueties

Joins

UDF

UDAF

SERILIZATION

DESERILIZATION