



**CHENNAI  
INSTITUTE OF TECHNOLOGY**  
(Autonomous)

## **UNIT V HADOOP RELATED TOOLS**

Hbase – data model and implementations – Hbase clients – Hbase examples – praxis. Pig – Grunt – pig data model – Pig Latin – developing and testing Pig Latin scripts. Hive – data types and file formats – HiveQL data definition – HiveQL data manipulation – HiveQL queries.

### **Hbase**

Hadoop can perform only batch processing, and data will be accessed only in a sequential manner. That means one has to search the entire dataset even for the simplest of jobs.

A huge dataset when processed results in another huge data set, which should also be processed sequentially. At this point, a new solution is needed to access any point of data in a single unit of time (random access).

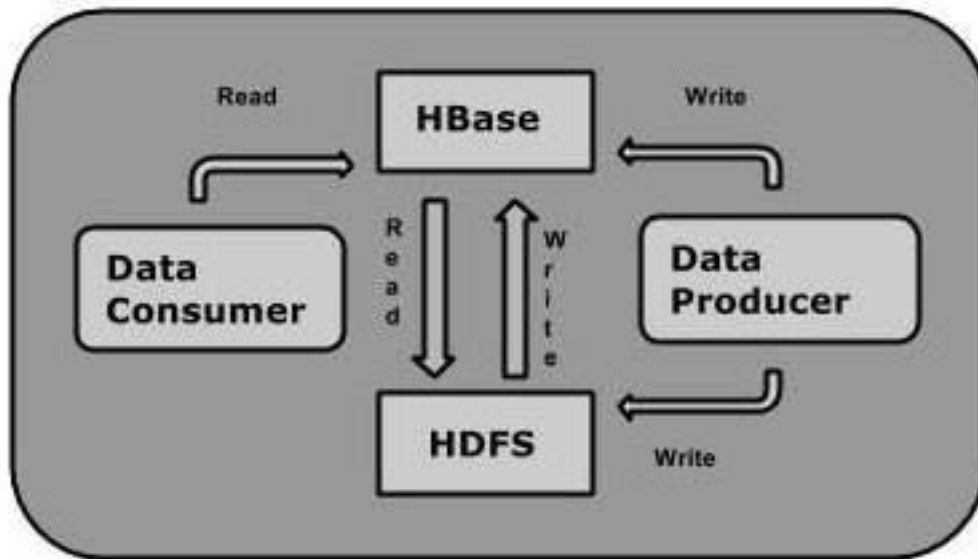
#### **What is HBase?**

HBase is a distributed column-oriented database built on top of the Hadoop file system. It is an open-source project and is horizontally scalable.

HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data. It leverages the fault tolerance provided by the Hadoop File System (HDFS).

It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System.

One can store the data in HDFS either directly or through HBase. Data consumer reads/accesses the data in HDFS randomly using HBase. HBase sits on top of the Hadoop File System and provides read and write access.



## HBase and HDFS

HDFS	HBase
HDFS is a distributed file system suitable for storing large files.	HBase is a database built on top of the HDFS.
HDFS does not support fast individual record lookups.	HBase provides fast lookups for larger tables.
It provides high latency batch processing; no concept of batch processing.	It provides low latency access to single rows from billions of records (Random access).
It provides only sequential access of data.	HBase internally uses Hash tables and provides random access, and it stores the data in indexed HDFS files for faster lookups.

## Storage Mechanism in HBase

HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp. In short, in an HBase:

- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

Given below is an example schema of table in HBase.

Rowid	Column Family	Column Family	Column Family	Column Family

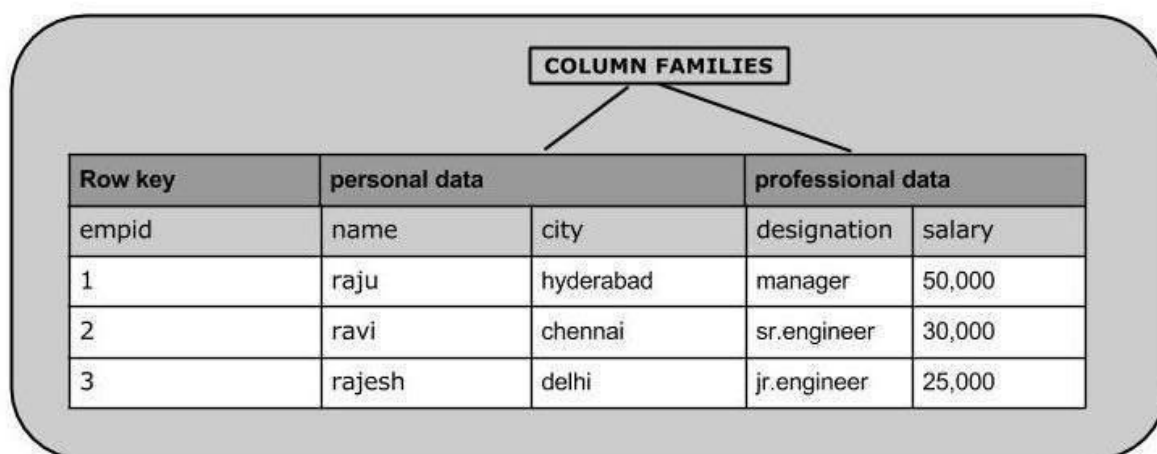
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

### Column Oriented and Row Oriented

Column-oriented databases are those that store data tables as sections of columns of data, rather than as rows of data. Shortly, they will have column families.

Row-Oriented Database	Column-Oriented Database
It is suitable for Online Transaction Process (OLTP).	It is suitable for Online Analytical Processing (OLAP).
Such databases are designed for small number of rows and columns.	Column-oriented databases are designed for huge tables.

The following image shows column families in a column-oriented database:



### HBase and RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

## **Features of HBase**

- HBase is linearly scalable.
- It has automatic failure support.
- It provides consistent read and writes.
- It integrates with Hadoop, both as a source and a destination.
- It has easy java API for client.
- It provides data replication across clusters.

## **Where to Use HBase**

- Apache HBase is used to have random, real-time read/write access to Big Data.
- It hosts very large tables on top of clusters of commodity hardware.
- Apache HBase is a non-relational database modeled after Google's Bigtable. Bigtable acts up on Google File System, likewise Apache HBase works on top of Hadoop and HDFS.


## **Applications of HBase**

- It is used whenever there is a need to write heavy applications.
- HBase is used whenever we need to provide fast random access to available data.
- Companies such as Facebook, Twitter, Yahoo, and Adobe use HBase internally.

## **5.2 Data model and implementations**

The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns. Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster. The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells and Versions.

Row Key	Customer		Sales	
Customer Id	Name	City	Product	Amount
101	John White	Los Angeles, CA	Chairs	\$400.00
102	Jane Brown	Atlanta, GA	Lamps	\$200.00
103	Bill Green	Pittsburgh, PA	Desk	\$500.00
104	Jack Black	St. Louis, MO	Bed	\$1600.00


**Column Families**

*Tables* – The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. As shown above, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

*Rows* – A row is one instance of data in a table and is identified by a *rowkey*. Rowkeys are unique in a Table and are always treated as a byte[].

*Column Families* – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it's important that proper care be taken when designing Column Families in table.

The table above shows Customer and Sales Column Families. The Customer Column Family is made up 2 columns – Name and City, whereas the Sales Column Families is made up to 2 columns – Product and Amount.

*Columns* – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily:columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

*Cell* – A Cell stores data and is essentially a unique combination of *rowkey*, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

*Version* – The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

## HBase Implementations

- Apache HBase: The most common and widely used implementation of HBase is Apache HBase. It is part of the Apache Hadoop ecosystem and is developed as an open-source project.
- Cloudera HBase: Cloudera, a provider of big data solutions, offers its own distribution of Hadoop, which includes HBase as part of its platform.
- Hortonworks HBase: Hortonworks, another prominent provider of Hadoop-based solutions, includes HBase in its distribution of Hadoop.
- MapR HBase: MapR Technologies provides its own distribution of Hadoop, which includes HBase as one of its core components.
- Cloud-based Implementations: Various cloud service providers, such as Amazon Web Services (AWS) with Amazon EMR and Google Cloud Platform (GCP) with Cloud Bigtable, offer managed HBase implementations in their respective cloud environments.

These implementations of HBase provide the necessary infrastructure and tooling to deploy and manage HBase clusters, ensuring scalability, fault tolerance, and efficient data storage and retrieval for big data workloads.

### 5.3 Hbase clients

This describes the java client API for HBase that is used to perform **CRUD** operations on HBase tables. HBase is written in Java and has a Java Native API. Therefore it provides programmatic access to Data Manipulation Language (DML).

#### Class HBase Configuration

Adds HBase configuration files to a Configuration. This class belongs to the **org.apache.hadoop.hbase** package.

Methods and description

S.No.	Methods and Description
1	<b>static org.apache.hadoop.conf.Configuration create()</b>
	This method creates a Configuration with HBase resources.

#### Class HTable

HTable is an HBase internal class that represents an HBase table. It is an implementation of table that is used to communicate with a single HBase table. This class belongs to the **org.apache.hadoop.hbase.client** class.

Constructors

S.No.	Constructors and Description
-------	------------------------------

1	<b>HTable()</b>
2	<b>HTable(TableName tableName, ClusterConnection connection, ExecutorService pool)</b>  Using this constructor, you can create an object to access an HBase table.

### Methods and description

S.No.	Methods and Description
1	<b>void close()</b>  Releases all the resources of the HTable.
2	<b>void delete(Delete delete)</b>  Deletes the specified cells/row.
3	<b>boolean exists(Get get)</b>  Using this method, you can test the existence of columns in the table, as specified by Get.
4	<b>Result get(Get get)</b>  Retrieves certain cells from a given row.
5	<b>org.apache.hadoop.conf.Configuration getConfiguration()</b> Returns the Configuration object used by this instance.
6	<b>TableName getName()</b>  Returns the table name instance of this table.
7	<b>HTableDescriptor getTableDescriptor()</b> Returns the table descriptor for this table.
8	<b>byte[] getTableName()</b>  Returns the name of this table.
9	<b>void put(Put put)</b>  Using this method, you can insert data into the table.

## Class Put

This class is used to perform Put operations for a single row. It belongs to the **org.apache.hadoop.hbase.client** package.

### Constructors

S.No.	Constructors and Description
1	<b>Put(byte[] row)</b> Using this constructor, you can create a Put operation for the specified row.
2	<b>Put(byte[] rowArray, int rowOffset, int rowLength)</b> Using this constructor, you can make a copy of the passed-in row key to keep local.
3	<b>Put(byte[] rowArray, int rowOffset, int rowLength, long ts)</b> Using this constructor, you can make a copy of the passed-in row key to keep local.
4	<b>Put(byte[] row, long ts)</b> Using this constructor, we can create a Put operation for the specified row, using a given timestamp.

### Methods

S.No.	Methods and Description
1	<b>Put add(byte[] family, byte[] qualifier, byte[] value)</b> Adds the specified column and value to this Put operation.
2	<b>Put add(byte[] family, byte[] qualifier, long ts, byte[] value)</b> Adds the specified column and value, with the specified timestamp as its version to this Put operation.
3	<b>Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)</b> Adds the specified column and value, with the specified timestamp as its version to this Put operation.
4	<b>Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)</b> Adds the specified column and value, with the specified timestamp as its version to this Put operation.

## Class Get

This class is used to perform Get operations on a single row. This class belongs to the **org.apache.hadoop.hbase.client** package.



### Constructor

S.No.	Constructor and Description
1	<b>Get(byte[] row)</b> Using this constructor, you can create a Get operation for the specified row.
2	<b>Get(Get get)</b>

### Methods

S.No.	Methods and Description
1	<b>Get addColumn(byte[] family, byte[] qualifier)</b> Retrieves the column from the specific family with the specified qualifier.
2	<b>Get addFamily(byte[] family)</b> Retrieves all columns from the specified family.

### Class Delete

This class is used to perform Delete operations on a single row. To delete an entire row, instantiate a Delete object with the row to delete. This class belongs to the **org.apache.hadoop.hbase.client** package.

### Constructor

S.No.	Constructor and Description
1	<b>Delete(byte[] row)</b> Creates a Delete operation for the specified row.
2	<b>Delete(byte[] rowArray, int rowOffset, int rowLength)</b> Creates a Delete operation for the specified row and timestamp.
3	<b>Delete(byte[] rowArray, int rowOffset, int rowLength, long ts)</b> Creates a Delete operation for the specified row and timestamp.
4	<b>Delete(byte[] row, long timestamp)</b> Creates a Delete operation for the specified row and timestamp.

### Methods

S.No.	Methods and Description
1	<b>Delete addColumn(byte[] family, byte[] qualifier)</b>

	Deletes the latest version of the specified column.
2	<b>Delete addColumns(byte[] family, byte[] qualifier, long timestamp)</b>  Deletes all versions of the specified column with a timestamp less than or equal to the specified timestamp.
3	<b>Delete addFamily(byte[] family)</b>  Deletes all versions of all columns of the specified family.
4	<b>Delete addFamily(byte[] family, long timestamp)</b>  Deletes all columns of the specified family with a timestamp less than or equal to the specified timestamp.

### Class Result

This class is used to get a single row result of a Get or a Scan query.

#### Constructors

S.No.	Constructors
1	<b>Result()</b>  Using this constructor, you can create an empty Result with no KeyValue payload; returns null if you call raw Cells().

#### Methods

S.No.	Methods and Description
1	<b>byte[] getValue(byte[] family, byte[] qualifier)</b>  This method is used to get the latest version of the specified column.
2	<b>byte[] getRow()</b>  This method is used to retrieve the row key that corresponds to the row from which this Result was created.

## 5.4 Hbase examples

Here are a few examples of how you can use HBase:

### 1. Creating a Table:

```

❑ TableName tableName = TableName.valueOf("myTable"); HTableDescriptor
tableDescriptor = new HTableDescriptor(tableName);
tableDescriptor.addFamily(new HColumnDescriptor("cf1"));

```

```
tableDescriptor.addFamily(new HColumnDescriptor("cf2"));
admin.createTable(tableDescriptor);
```

#### ❑ Inserting Data:

```
TableName tableName = TableName.valueOf("myTable"); Put put = new
Put(Bytes.toBytes("row1")); put.addColumn(Bytes.toBytes("cf1"),
Bytes.toBytes("col1"), Bytes.toBytes("value1")); put.addColumn(Bytes.toBytes("cf2"),
Bytes.toBytes("col2"), Bytes.toBytes("value2"));
Table table = connection.getTable(tableName); table.put(put);
```

#### ❑ Getting Data:

```
TableName tableName = TableName.valueOf("myTable"); Get
get = new Get(Bytes.toBytes("row1"));
get.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));
Table table = connection.getTable(tableName);
Result result = table.get(get); byte[] value =
result.getValue(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));
```

#### ❑ Scanning Data:

```
TableName tableName = TableName.valueOf("myTable"); Scan
scan = new Scan();
scan.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));
Table table = connection.getTable(tableName); ResultScanner
scanner = table.getScanner(scan);
for (Result result : scanner) {
    byte[] value = result.getValue(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));
    // Process the retrieved value
} scanner.close();
```

#### ❑ Deleting Data:

```
TableName tableName = TableName.valueOf("myTable"); Delete
delete = new Delete(Bytes.toBytes("row1"));
delete.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));
Table table = connection.getTable(tableName); table.delete(delete);
```

These examples demonstrate basic operations in HBase, including creating a table, inserting data, retrieving data, scanning data, and deleting data. Remember to properly handle exceptions and ensure you have established a connection to the HBase cluster using the appropriate configuration settings.

## Pig

Apache Pig is an abstraction over MapReduce. It is a tool/platform which is used to analyze larger sets of data representing them as data flows. Pig is generally used with **Hadoop**; we can perform all the data manipulation operations in Hadoop using Apache Pig.

To write data analysis programs, Pig provides a high-level language known as **Pig Latin**. This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.

To analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language. All these scripts are internally converted to Map and Reduce tasks. Apache Pig has a component known as **Pig Engine** that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

### Why Do We Need Apache Pig?

Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks. Apache Pig is a boon for all such programmers.

- Using **Pig Latin**, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
- Apache Pig uses **multi-query approach**, thereby reducing the length of codes. For example, an operation that would require you to type 200 lines of code (LoC) in Java can be easily done by typing as less as just 10 LoC in Apache Pig. Ultimately Apache Pig reduces the development time by almost 16 times.
- Pig Latin is **SQL-like language** and it is easy to learn Apache Pig when you are familiar with SQL.
- Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc. In addition, it also provides nested data types like tuples, bags, and maps that are missing from MapReduce.

### Features of Pig

Apache Pig comes with the following features –

- **Rich set of operators** – It provides many operators to perform operations like join, sort, filter, etc.
- **Ease of programming** – Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.
- **Optimization opportunities** – The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language. • **Extensibility** – Using the existing operators, users can develop their own functions to read, process, and write data.

- **UDF's** – Pig provides the facility to create **User-defined Functions** in other programming languages such as Java and invoke or embed them in Pig Scripts.
- **Handles all kinds of data** – Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

### Apache Pig Vs MapReduce

Listed below are the major differences between Apache Pig and MapReduce.

Apache Pig	MapReduce
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is pretty simple.	It is quite difficult in MapReduce to perform a Join operation between datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.
Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent.	MapReduce will require almost 20 times more the number of lines to perform the same task.
There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

### Apache Pig Vs SQL

Listed below are the major differences between Apache Pig and SQL.

Pig	SQL
Pig Latin is a <b>procedural</b> language.	SQL is a <b>declarative</b> language.
In Apache Pig, <b>schema</b> is optional. We can store data without designing a schema (values are stored as \$01, \$02 etc.)	Schema is mandatory in SQL.
The data model in Apache Pig is <b>nested relational</b> .	The data model used in SQL is <b>flat relational</b> .
Apache Pig provides limited opportunity for <b>Query optimization</b> .	There is more opportunity for query optimization in SQL.

In addition to above differences, Apache Pig Latin –

- Allows splits in the pipeline.
- Allows developers to store data anywhere in the pipeline.
- Declares execution plans.
- Provides operators to perform ETL (Extract, Transform, and Load) functions.

## Apache Pig Vs Hive

Both Apache Pig and Hive are used to create MapReduce jobs. And in some cases, Hive operates on HDFS in a similar way Apache Pig does. In the following table, we have listed a few significant points that set Apache Pig apart from Hive.

Apache Pig	Hive
Apache Pig uses a language called <b>Pig Latin</b> . It was originally created at <b>Yahoo</b> .	Hive uses a language called <b>HiveQL</b> . It was originally created at <b>Facebook</b> .
Pig Latin is a data flow language.	HiveQL is a query processing language.
Pig Latin is a procedural language and it fits in pipeline paradigm.	HiveQL is a declarative language.
Apache Pig can handle structured, unstructured, and semi-structured data.	Hive is mostly for structured data.

### Applications of Apache Pig

Apache Pig is generally used by data scientists for performing tasks involving ad-hoc processing and quick prototyping. Apache Pig is used –

- To process huge data sources such as web logs.
- To perform data processing for search platforms.
- To process time sensitive data loads.

### Apache Pig – History

In **2006**, Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset. In **2007**, Apache Pig was open sourced via Apache incubator. In **2008**, the first release of Apache Pig came out. In **2010**, Apache Pig graduated as an Apache top-level project.

## Grunt

After invoking the Grunt shell, you can run your Pig scripts in the shell. In addition to that, there are certain useful shell and utility commands provided by the Grunt shell. This explains the shell and utility commands provided by the Grunt shell.

### Shell Commands

The Grunt shell of Apache Pig is mainly used to write Pig Latin scripts. Prior to that, we can invoke any shell commands using **sh** and **fs**.

## sh Command

Using **sh** command, we can invoke any shell commands from the Grunt shell. Using **sh** command from the Grunt shell, we cannot execute the commands that are a part of the shell environment (**ex** – cd).

### Syntax

Given below is the syntax of **sh** command.

```
grunt> sh shell command parameters
```

### Example

We can invoke the **ls** command of Linux shell from the Grunt shell using the **sh** option as shown below. In this example, it lists out the files in the **/pig/bin/** directory.

```
grunt> sh ls
```

```
pig
pig_1444799121955.log
pig.cmd pig.py
```

## fs Command

Using the **fs** command, we can invoke any FsShell commands from the Grunt shell.

### Syntax

Given below is the syntax of **fs** command. `grunt>`

```
sh File System command parameters
```

### Example

We can invoke the **ls** command of HDFS from the Grunt shell using **fs** command. In the following example, it lists the files in the HDFS root directory.

```
grunt> fs -ls
```

```
Found 3 items drwxrwxrwx - Hadoop supergroup    0 2015-09-08
14:13 Hbase drwxr-xr-x - Hadoop supergroup    0 2015-09-09
14:52 seqgen_data drwxr-xr-x - Hadoop supergroup    0 2015-09-08
11:30 twitter_data
```

In the same way, we can invoke all the other file system shell commands from the Grunt shell using the **fs** command.

## Utility Commands

The Grunt shell provides a set of utility commands. These include utility commands such as **clear**, **help**, **history**, **quit**, and **set**; and commands such as **exec**, **kill**, and **run** to control Pig from the Grunt shell. Given below is the description of the utility commands provided by the Grunt shell.

### clear Command

The **clear** command is used to clear the screen of the Grunt shell.

### Syntax

You can clear the screen of the grunt shell using the **clear** command as shown below.

```
grunt> clear
```

### help Command

The **help** command gives you a list of Pig commands or Pig properties.

### Usage

You can get a list of Pig commands using the **help** command as shown below.

```
grunt> help
```

### history Command

This command displays a list of statements executed / used so far since the Grunt shell is invoked.

### Usage

Assume we have executed three statements since opening the Grunt shell.

```
grunt> customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING  
PigStorage(',');
```

```
grunt> orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING PigStorage(',');
```

```
grunt> student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',');
```

Then, using the **history** command will produce the following output.



**grunt> history**

```
customers = LOAD 'hdfs://localhost:9000/pig_data/customers.txt' USING PigStorage(',');
```

```
orders = LOAD 'hdfs://localhost:9000/pig_data/orders.txt' USING PigStorage(',');
```

```
student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING PigStorage(',');
```

## set Command

The **set** command is used to show/assign values to keys used in Pig.

## Usage

Using this command, you can set values to the following keys.

Key	Description and values
<b>default_parallel</b>	You can set the number of reducers for a map job by passing any whole number as a value to this key.
<b>debug</b>	You can turn off or turn on the debugging freature in Pig by passing on/off to this key.
<b>job.name</b>	You can set the Job name to the required job by passing a string value to this key.
<b>job.priority</b>	You can set the job priority to a job by passing one of the following values to this key – <ul style="list-style-type: none"><li>• very_low</li><li>• low</li><li>• normal</li><li>• high</li><li>• very_high</li></ul>
<b>stream.skippath</b>	For streaming, you can set the path from where the data is not to be transferred, by passing the desired path in the form of a string to this key.

## quit Command

You can quit from the Grunt shell using this command.

## Usage

Quit from the Grunt shell as shown below.

```
grunt> quit
```

Let us now take a look at the commands using which you can control Apache Pig from the Grunt shell.

### **exec Command**

Using the **exec** command, we can execute Pig scripts from the Grunt shell.

#### **Syntax**

Given below is the syntax of the utility command **exec**.

```
grunt> exec [-param param_name = param_value] [-param_file file_name] [script]
```

### **kill Command**

You can kill a job from the Grunt shell using this command.

#### **Syntax**

Given below is the syntax of the **kill** command. `grunt>`

```
kill JobId
```

#### **Example**

Suppose there is a running Pig job having id **Id\_0055**, you can kill it from the Grunt shell using the **kill** command, as shown below.

```
grunt> kill Id_0055
```

### **run Command**

You can run a Pig script from the Grunt shell using the **run** command

#### **Syntax**

Given below is the syntax of the **run** command.

```
grunt> run [-param param_name = param_value] [-param_file file_name] script
```

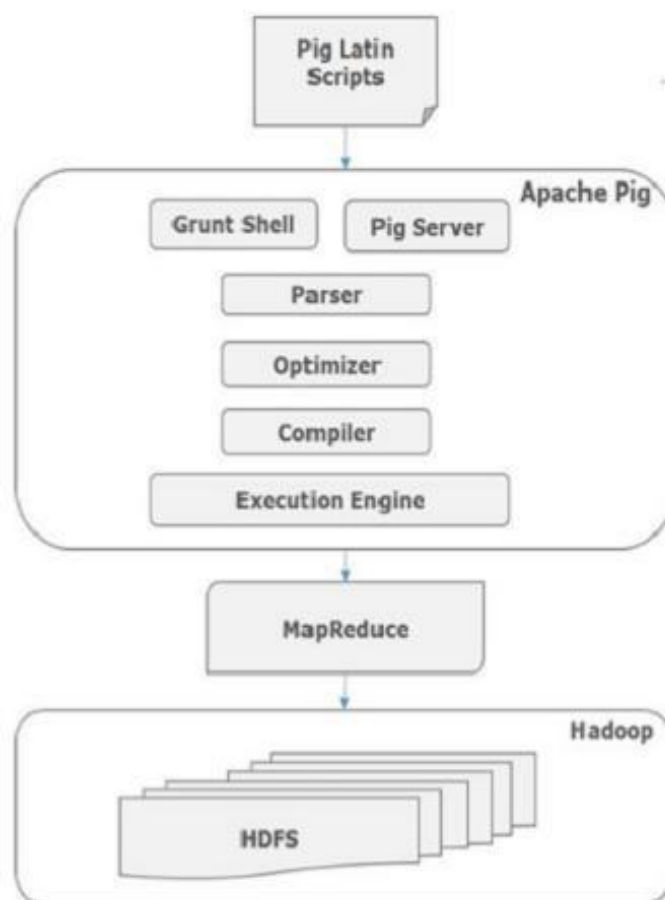
**Note** – The difference between **exec** and the **run** command is that if we use **run**, the statements from the script are available in the command history.

## Pig Architecture

The language used to analyze data in Hadoop using Pig is known as **Pig Latin**. It is a highlevel data processing language which provides a rich set of data types and operators to perform various operations on the data.

To perform a particular task Programmers using Pig, programmers need to write a Pig script using the Pig Latin language, and execute them using any of the execution mechanisms (Grunt Shell, UDFs, Embedded). After execution, these scripts will go through a series of transformations applied by the Pig Framework, to produce the desired output.

Internally, Apache Pig converts these scripts into a series of MapReduce jobs, and thus, it makes the programmer's job easy. The architecture of Apache Pig is shown below.



## Apache Pig Components

As shown in the figure, there are various components in the Apache Pig framework. Let us take a look at the major components.

### Parser

Initially the Pig Scripts are handled by the Parser. It checks the syntax of the script, does type checking, and other miscellaneous checks. The output of the parser will be a DAG (directed acyclic graph), which represents the Pig Latin statements and logical operators.

In the DAG, the logical operators of the script are represented as the nodes and the data flows are represented as edges.

## Optimizer

The logical plan (DAG) is passed to the logical optimizer, which carries out the logical optimizations such as projection and pushdown.

## Compiler

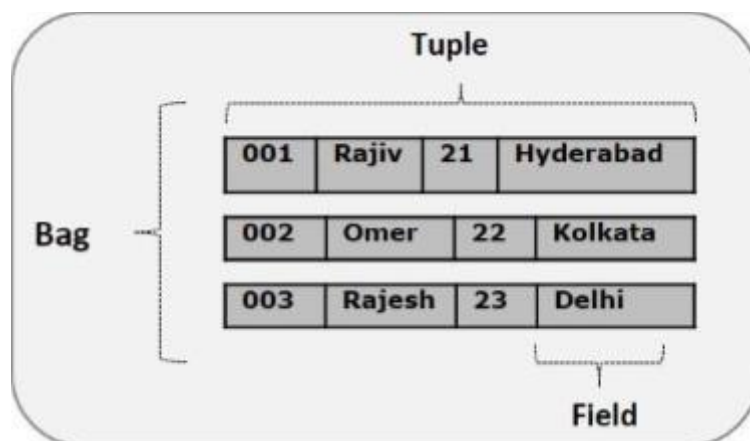
The compiler compiles the optimized logical plan into a series of MapReduce jobs.

## Execution engine

Finally the MapReduce jobs are submitted to Hadoop in a sorted order. Finally, these MapReduce jobs are executed on Hadoop producing the desired results.

## Pig Latin Data Model

The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**. Given below is the diagrammatical representation of Pig Latin's data model.



## Atom

Any single value in Pig Latin, irrespective of their data, type is known as an **Atom**. It is stored as string and can be used as string and number. int, long, float, double, chararray, and bytearray are the atomic values of Pig. A piece of data or a simple atomic value is known as a **field**.

**Example** – 'raja' or '30'

## Tuple

A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type. A tuple is similar to a row in a table of RDBMS.

**Example** – (Raja, 30)

## Bag

A bag is an unordered set of tuples. In other words, a collection of tuples (non-unique) is known as a bag. Each tuple can have any number of fields (flexible schema). A bag is represented by ‘{}’. It is similar to a table in RDBMS, but unlike a table in RDBMS, it is not necessary that every tuple contain the same number of fields or that the fields in the same position (column) have the same type.

**Example** – {(Raja, 30), (Mohammad, 45)}

A bag can be a field in a relation; in that context, it is known as **inner bag**.

**Example** – {Raja, 30, {9848022338, raja@gmail.com,}}

## Map

A map (or data map) is a set of key-value pairs. The **key** needs to be of type chararray and should be unique. The **value** might be of any type. It is represented by ‘[]’

**Example** – [name#Raja, age#30]

## Relation

A relation is a bag of tuples. The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

## Developing and testing Pig Latin scripts

Developing and testing Pig Latin scripts involves the following steps:

1. Install Apache Pig:
  - Start by installing Apache Pig on your machine or cluster. You can download the Pig distribution from the official Apache Pig website.
2. Set up the Environment:
  - Configure the necessary environment variables, such as setting the PIG\_HOME variable to the installation directory of Apache Pig.
  - Add the Pig binary to your system's PATH variable to run Pig commands from the command line.

3. Write a Pig Latin Script:
  - Create a new text file with a .pig extension (e.g., script.pig) to write your Pig Latin script.
  - Open the file in a text editor and start writing Pig Latin statements to perform the desired data transformations and analysis.
4. Load Data:
  - Use the LOAD operator to load data into Pig from various sources like HDFS, local file systems, or databases.
  - Specify the input format and schema of the data being loaded.
5. Transform and Analyze Data:
  - Apply a sequence of Pig Latin operations to transform and analyze the data. Use operators such as FILTER, GROUP, JOIN, FOREACH, and others.
  - Leverage built-in functions or write your own User-Defined Functions (UDFs) to perform custom computations or transformations.
6. Store Data:
  - Use the STORE operator to store the results of data processing into various formats or storage systems, such as HDFS, local file systems, or databases.
  - Specify the output format and location for storing the processed data.
7. Run the Script Locally:
  - Open a command prompt or terminal and navigate to the directory where the Pig Latin script is saved.
  - Run the Pig Latin script in local mode using the following command: `pig -x local script.pig`
  - The script will execute on your local machine, and you can view the output and any error messages in the console.
8. Run the Script on a Cluster:
  - To run the Pig Latin script on a Hadoop cluster, use the following command: `pig -x mapreduce script.pig`
  - Pig will generate MapReduce jobs and submit them to the Hadoop cluster for execution.
  - Monitor the job progress using the Hadoop cluster's web interface or commandline tools.
9. Debug and Iterate:
  - If your script encounters any errors or produces unexpected results, debug the script by examining the error messages and intermediate results. ○ Use Pig's built-in diagnostic functions like DUMP or DESCRIBE to inspect the intermediate data at various stages of the script. ○ Make necessary adjustments to the script and repeat the execution until you achieve the desired results.
10. Use Pig Grunt Shell for Interactive Testing:
  - Launch the Pig Grunt shell by running the pig command without specifying a script file.
  - Enter Pig Latin statements directly into the shell and execute them to test small portions of your script interactively. ○ The Grunt shell provides a convenient way to experiment, debug, and refine your Pig Latin code.

By following these steps, you can develop and test your Pig Latin scripts, iterate on them as needed, and ensure that they produce the desired results before executing them at scale on a Hadoop cluster.

## Hive

Hive is an open-source data warehouse infrastructure built on top of Apache Hadoop. It provides a high-level query language, called HiveQL (similar to SQL), that allows users to perform SQL-like queries on large datasets stored in Hadoop's distributed file system (HDFS) or other compatible file systems. Hive translates HiveQL queries into MapReduce, Tez, or Spark jobs, allowing users to leverage the power of Hadoop for data processing and analysis Source.

### Architecture of Hive

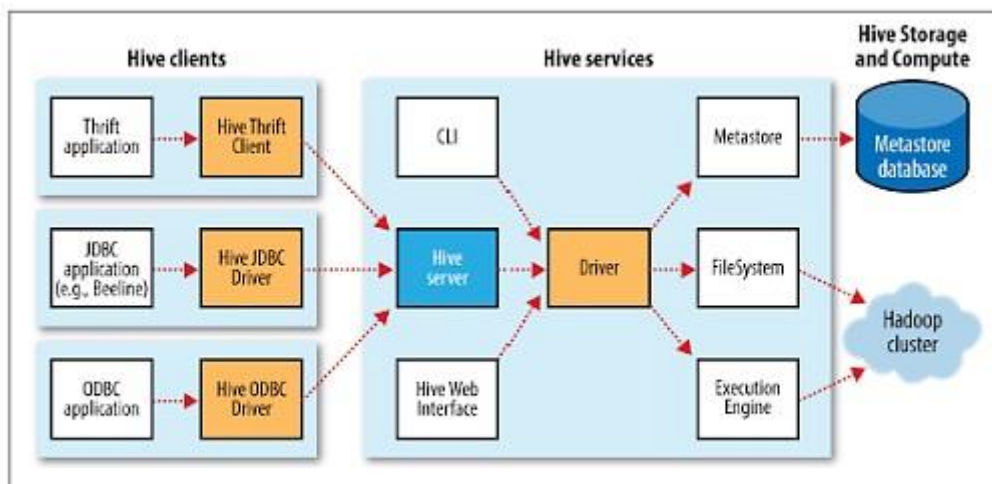


Figure 17-1. Hive architecture

Hive chiefly consists of three core parts:

- **Hive Clients:** Hive offers a variety of drivers designed for communication with different applications. For example, Hive provides Thrift clients for Thrift-based applications. These clients and drivers then communicate with the Hive server, which falls under Hive services.
- **Hive Services:** Hive services perform client interactions with Hive. For example, if a client wants to perform a query, it must talk with Hive services.
- **Hive Storage and Computing:** Hive services such as file system, job client, and meta store then communicates with Hive storage and stores things like metadata table information and query results.

### Hive's Features

These are Hive's chief characteristics:

- Hive is designed for querying and managing only structured data stored in tables
- Hive is scalable, fast, and uses familiar concepts
- Schema gets stored in a database, while processed data goes into a Hadoop Distributed File System (HDFS)
- Tables and databases get created first; then data gets loaded into the proper tables
- Hive supports four file formats: ORC, SEQUENCEFILE, RCFILE (Record Columnar File), and TEXTFILE
- Hive uses an SQL-inspired language, sparing the user from dealing with the complexity of MapReduce programming. It makes learning more accessible by utilizing familiar concepts found in relational databases, such as columns, tables, rows, and schema, etc.
- The most significant difference between the Hive Query Language (HQL) and SQL is that Hive executes queries on Hadoop's infrastructure instead of on a traditional database
- Since Hadoop's programming works on flat files, Hive uses directory structures to "partition" data, improving performance on specific queries
- Hive supports partition and buckets for fast and simple data retrieval
- Hive supports custom user-defined functions (UDF) for tasks like data cleansing and filtering. Hive UDFs can be defined according to programmers' requirements

## **Limitations of Hive**

Of course, no resource is perfect, and Hive has some limitations. They are:

- Hive doesn't support OLTP. Hive supports Online Analytical Processing (OLAP), but not Online Transaction Processing (OLTP). • It doesn't support subqueries.
- It has a high latency.
- Hive tables don't support delete or update operations.

## **Hive Modes**

Depending on the size of Hadoop data nodes, Hive can operate in two different modes:

- Local mode
- Map-reduce mode

User Local mode when:

- Hadoop is installed under the pseudo mode, possessing only one data node
- The data size is smaller and limited to a single local machine
- Users expect faster processing because the local machine contains smaller datasets.

Use Map Reduce mode when:

- Hadoop has multiple data nodes, and the data is distributed across these different nodes
- Users must deal with more massive data sets



MapReduce is Hive's default mode.

## Data types and file formats

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 65535

CHAR	255
------	-----

## Timestamp

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

## 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  $10^{308}$ .

## 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], ...>

## File Formats:

Some special file formats that Hive can handle are available, such as:

- Text File Format
- Sequence File Format
- RC File (Row column file format)
- Avro Files
- ORC Files (Optimized Row Columnar file format) • Parquet
- Custom INPUTFORMAT and OUTPUTFORMAT

### 1) TEXT FILE FORMAT:

Hive Text file format is a default storage format to load data from comma-separated values (CSV), tab-delimited, space-delimited, or text files that delimited by other special characters. You can use the text format to interchange the data with other client applications. The text file format is very common for most of the applications. Data is stored in lines, with each line being a record. Each line is terminated by a newline character (\n). The text file format storage option is defined by specifying “STORED AS TEXTFILE” at the end of the table creation.

### 2) SEQUENCE FILE FORMAT:

Flat files consisting of binary key-value pairs are sequence files. When converting queries to MapReduce jobs, Hive chooses to use the necessary key-value pairs for a given record. The key advantages of using a sequence file are that it incorporates two or more files into one

file. The sequence file format storage option is defined by specifying **“STORED AS SEQUENCEFILE”** at the end of the table creation.

### **3) RCFILE FORMAT:**

The row columnar file format is very much similar to the sequence file format. It is a data placement structure designed for MapReduce-based data warehouse systems. This also stores the data as key-value pairs and offers a high row-level compression rate. This will be used when there is a requirement to perform multiple rows at a time. RCFile format is supported by Hive version 0.6.0 and later. The RC file format storage option is defined by specifying **“STORED AS RCFILE”** at the end of the table creation.

### **4) AVRO FILE FORMAT:**

Hive version 0.14.0 and later versions support Avro files. It is a row-based storage format for Hadoop which is widely used as a serialization platform. It's a remote procedure call and data serialization framework that uses JSON for defining data types and protocols and serializes data in a compact binary format to make it compact and efficient. This file format can be used in any of the Hadoop's tools like Pig and Hive. Avro is one of the common file formats in applications based on Hadoop. The option to store the data in the RC file format is defined by specifying **“STORED AS AVRO”** at the end of the table creation.

### **5) ORC FILE FORMAT:**

The Optimized Row Columnar (ORC) file format provides a highly efficient way to store data in the Hive table. This file system was actually designed to overcome limitations of the other Hive file formats. ORC reduces I/O overhead by accessing only the columns that are required for the current query. It requires significantly fewer seek operations because all columns within a single group of row data are stored together on disk. The ORC file format storage option is defined by specifying **“STORED AS ORC”** at the end of the table creation.

### **6) PARQUET:**

Parquet is a binary file format that is column driven. It is an open-source available to any project in the Hadoop ecosystem and is designed for data storage in an effective and efficient flat columnar format compared to row-based files such as CSV or TSV files. It only reads the necessary columns, which significantly reduces the IO and thus makes it highly efficient for large-scale query types. The Parquet table uses Snappy, which is a fast data compression and decompression library, as the default compression.

The parquet file format storage option is defined by specifying **“STORED AS PARQUET”** at the end of the table creation.

### **7) CUSTOMER INPUTFORMAT & OUTPUTFORMAT:**

We can implement our own “inputformat” and “outputformat” in case the data comes in a different format. These “inputformat” and “outputformat” is similar to Hadoop MapReduce's input and output formats.

## HiveQL data definition

Hive DDL commands are the statements used for defining and changing the structure of a table or database in Hive. It is used to build or modify the tables and other objects in the database.

The several types of Hive DDL commands are:

1. CREATE
2. SHOW
3. DESCRIBE
4. USE
5. DROP
6. ALTER
7. TRUNCATE

**Table-1 Hive DDL commands**

DDL Command	Use With
CREATE	Database, Table
SHOW	Databases, Tables, Table Properties, Partitions, Functions, Index
DESCRIBE	Database, Table, view
USE	Database
DROP	Database, Table
ALTER	Database, Table
TRUNCATE	Table

Before moving forward, note that the Hive commands are **case-insensitive**.

CREATE DATABASE is the same as create database.

So now, let us go through each of the commands deeply. Let's start with the DDL commands on Databases in Hive.

DDL Commands On Databases in Hive

### *1. CREATE DATABASE in Hive*

The **CREATE DATABASE** statement is used to create a database in the Hive. The DATABASE and SCHEMA are interchangeable. We can use either DATABASE or SCHEMA.

#### **Syntax:**

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name  
[COMMENT database_comment]  
[LOCATION hdfs_path]  
[WITH DBPROPERTIES (property_name=property_value, ...)];
```

### *2. SHOW DATABASE in Hive*

The **SHOW DATABASES** statement lists all the databases present in the Hive.

#### **Syntax:**

```
SHOW (DATABASES|SCHEMAS);
```

### *3. DESCRIBE DATABASE in Hive*

The **DESCRIBE DATABASE** statement in Hive shows the name of Database in Hive, its comment (if set), and its location on the file system.

The **EXTENDED** can be used to get the database properties.

#### **Syntax:**

```
DESCRIBE DATABASE/SCHEMA [EXTENDED] db_name;
```

### *4. USE DATABASE in Hive*

The **USE** statement in Hive is used to select the specific database for a session on which all subsequent HiveQL statements would be executed.

#### **Syntax:**

```
USE database_name;
```

### *5. DROP DATABASE in Hive*

The **DROP DATABASE** statement in Hive is used to Drop (delete) the database.

The default behavior is RESTRICT which means that the database is dropped only when it is empty. To drop the database with tables, we can use CASCADE.

**Syntax:**

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

*6. ALTER DATABASE in Hive*

The **ALTER DATABASE** statement in Hive is used to change the metadata associated with the database in Hive.

**Syntax for changing Database Properties:**

```
ALTER (DATABASE|SCHEMA) database_name SET DBPROPERTIES  
(property_name=property_value, ...);
```

**DDL Commands on Tables in Hive**

*1. CREATE TABLE*

The **CREATE TABLE** statement in Hive is used to create a table with the given name. If a table or view already exists with the same name, then the error is thrown. We can use **IF NOT EXISTS** to skip the error.

**Syntax:**

```
CREATE TABLE [IF NOT EXISTS] [db_name.] table_name [(col_name data_type  
[COMMENT col_comment], ... [COMMENT col_comment])] [COMMENT table_comment]  
[ROW FORMAT row_format] [STORED AS file_format] [LOCATION hdfs_path];
```

*2. SHOW TABLES in Hive*

The **SHOW TABLES** statement in Hive lists all the base tables and **views** in the current database.

**Syntax:**

```
SHOW TABLES [IN database_name];
```

*3. DESCRIBE TABLE in Hive*

The **DESCRIBE** statement in Hive shows the lists of columns for the specified table.

**Syntax:**

```
DESCRIBE [EXTENDED|FORMATTED] [db_name.] table_name[.col_name (
[.field_name]])];
```

*4. DROP TABLE in Hive*

The **DROP TABLE** statement in Hive deletes the data for a particular table and remove all metadata associated with it from Hive metastore.

If **PURGE** is not specified then the data is actually moved to the .Trash/current directory. If **PURGE** is specified, then data is lost completely.

**Syntax:**

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

*5. ALTER TABLE in Hive*

The **ALTER TABLE** statement in Hive enables you to change the structure of an existing table. Using the ALTER TABLE statement we can rename the table, add columns to the table, change the table properties, etc.

**Syntax to Rename a table:**

```
ALTER TABLE table_name RENAME TO new_table_name;
```

*6. TRUNCATE TABLE*

**TRUNCATE TABLE** statement in Hive removes all the rows from the table or partition.

**Syntax:**

```
TRUNCATE TABLE table_name;
```

## **HiveQL data manipulation**

Apache Hive DML stands for (Data Manipulation Language) which is used to insert, update, delete, and fetch data from Hive tables. Using DML commands we can load files into Apache Hive tables, write data into the filesystem from Hive queries, perform merge operation on the table, and so on.

The following list of DML statements is supported by Apache Hive.

- **LOAD**



- `SELECT`
- `INSERT`
- `DELETE`
- `UPDATE`
- `EXPORT`
- `IMPORT`

**LOAD:** The `LOAD` statement is used to load data from external files (e.g., in HDFS) into a Hive table. It's more of a data loading operation rather than a traditional DML statement. For example:

sql

```
LOAD DATA LOCAL INPATH '/path/to/data/file' INTO TABLE mytable;
```

- **SELECT:** The `SELECT` statement is used for querying data from Hive tables. It retrieves data and does not modify the underlying data.
- **INSERT:** Hive provides an `INSERT INTO` statement to insert data into tables. However, it appends data to existing data rather than performing updates or modifications. It's more of an insert operation than a traditional SQL `INSERT` statement:

sql

```
INSERT INTO mytable VALUES (1, 'John', 25);
```

- **DELETE:** Hive does support a `DELETE` statement for deleting rows from a table, but this operation is limited to ACID-compliant Hive tables (e.g., ORC or Parquet) with transactions enabled. It's not widely used in typical Hive workloads.
- **UPDATE:** Hive traditionally does not support a standard SQL `UPDATE` statement to modify data in existing rows. Instead, you often need to work with Hive's batch processing model and recreate tables with updated data.
- **EXPORT and IMPORT:** These are HiveQL extensions for exporting data from Hive tables into external storage (`EXPORT`) or importing data from external storage into Hive tables (`IMPORT`). They are not standard SQL DML statements.

## HiveQL queries

In Apache Hive, HiveQL (Hive Query Language) is used to query data stored in Hive tables. HiveQL provides a SQL-like syntax for querying structured and semi-structured data. Here are some commonly used query types in HiveQL:

1. SELECT: ○ Used to retrieve data from one or more tables. ○ Syntax:

```
SELECT column1 [, column2, ...]
FROM table_name
[JOIN table2_name ON join_condition]
[WHERE condition]
[GROUP BY column1 [, column2, ...]]
[HAVING condition]
[ORDER BY column1 [ASC|DESC] [, column2 [ASC|DESC], ...]] [LIMIT
n];
```

### □ JOIN:

- Used to combine records from two or more tables based on a common column.

Syntax:

```
SELECT column1 [, column2, ...]
FROM table1_name
JOIN table2_name ON join_condition;
```

### □ GROUP BY:

Used to group rows based on a specified column or columns.

Syntax:

```
SELECT column1 [, column2, ...], aggregate_function(column)
FROM table_name
GROUP BY column1 [, column2, ...];
```

### □ HAVING:

Used to filter the result set based on aggregate function results.

Syntax:

```
SELECT column1 [, column2, ...], aggregate_function(column)
FROM table_name
GROUP BY column1 [, column2, ...] HAVING
condition;
```

#### ❑ ORDER BY:

Used to sort the result set based on one or more columns.

Syntax:

```
SELECT column1 [, column2, ...]  
FROM table_name  
ORDER BY column1 [ASC|DESC] [, column2 [ASC|DESC], ...];
```

#### ❑ LIMIT:

Used to restrict the number of rows returned in the result set.

Syntax:

```
SELECT column1 [, column2, ...]  
FROM table_name LIMIT  
n;
```

#### ❑ Subqueries:

Used to nest one query inside another query.

Syntax:

```
SELECT column1 [, column2, ...]  
FROM table_name  
WHERE column IN (SELECT column FROM table2_name WHERE condition);
```

#### ❑ Conditional Expressions:

Used to apply conditional logic in queries.

Syntax:

```
SELECT column1 [, column2, ...]  
FROM table_name  
WHERE column = value  
OR column <> value  
AND column > value  
...
```

These are some of the common query types in HiveQL. They allow you to retrieve, filter, aggregate, and sort data from Hive tables. HiveQL supports various SQL-like constructs, making it easier to work with structured data in Hive.

