22MCA2052 – Big Data Analytics
Module – 5: Programming Hive

Text Book: Jason R, Dean W, Edward C, "Programming Hive", O'reilly, 2012

Programming Hive: Hive in the Hadoop Ecosystem, Data Types and File Formats, HiveQL: Data Definition, Databases in Hive, Alter Database, Creating Tables, External Tables, Partitioned Tables, External Partitioned Tables, Dropping Tables, Alter Tables, HiveQL: Data Manipulation, Queries (till GROUP BY Clauses).

About Hive:

Apache Hive is an open-source warehousing system that is built on top of Hadoop in 2010. Presently, **Hive 3.1.3** is the stable version.

Hive is used for querying and analyzing massive datasets stored within Hadoop. It works by processing **both structured and semi-structured data**.

Apache Hive is simply a **data warehouse** software built by using Hadoop as the base. Before Apache Hive, Big Data engineers had to write complex map-reduce jobs to perform querying tasks. With Hive, on the other hand, things drastically reduced as engineers now only need to know SQL.

Hive works on a language known as **HiveQL** (similar to SQL), making it easier for engineers who have a working knowledge of SQL.

HiveQL **automatically translates SQL queries into map-reduce jobs** that Hadoop can execute.

Hive is most suited for data warehouse applications, where relatively static data is analyzed, fast response times are not required, and when the data is not changing rapidly.

Because most data warehouse applications are implemented using SQL-based relational databases, **Hive lowers the barrier** for moving these applications to Hadoop.

Hive is **not a full database**. The design constraints and limitations of Hadoop and HDFS impose limits on what Hive can do. The biggest limitation is that Hive **does not provide record-level update, insert, nor delete.**

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.

One of the main reasons behind the popularity of this web application framework is its **SQL interface** which operates smoothly on Hadoop.

Hive does not provide crucial features required for OLTP (Online Transaction Processing) but it closer to being an **OLAP tool** (Online Analytic Processing).

Features of Hive:

- Fast, scalable, and user-friendly environment.
- Hadoop as its storage engine.
- SQL-like query language called as HQL (Hive Query Language).
- Can be used for OLAP systems (Online Analytical Processing).
- Supports databases and file systems that can be integrated with Hadoop.
- This includes HBase, and Cassandra, among others. They are responsible for aiding applications in their process of performing analytics and reports on large sets of data.
- Supports different types of storage types like Hbase, ORC, etc.
- Perhaps one of the best features of Hive is that it uses SQL-inspired language. This eliminates all the complexities of MapReduce programming. Furthermore, it also leads to a series of advantages, such as more accessibility to learning.
- This software framework is fully equipped to support User Defined Functions to address specific tasks such as data cleansing or filtering. What's more, Hive UDFs can also be quite easily defined in accordance with the requirements of the programmer.
- Hive is by far one of the most cost-effective web application framework that generates both high performance and scalability. With the help of Hadoop, Hive works as a high-scale database that can run on thousands of nodes.

Limitations of Hive:

- Not ideal for OLTP systems (Online Transactional Processing).
- Real-time queries are not supported in Hive. (Only Batch Processing)
- Does not support unstructured data.
- No Row Level Insert, Update and deletes.
- Does not support ACID properties, hence does not support transactions.

Hive in the Hadoop Ecosystem/Hive Architecture:

Hive is closely integrated with Hadoop and is designed to work fast on petabytes of data. What makes Hive unique is the ability to query large datasets, leveraging MapReduce, with a SQL-like interface.

Hive allows users to read, write, and manage petabytes of data using SQL.

Hive is created to allow non-programmers who are familiar with SQL to work with petabytes of data, say on Hadoop, using a SQL-like interface called HiveQL.

Hive is a **data warehouse infrastructure** tool to process **structured data** in **Hadoop**.

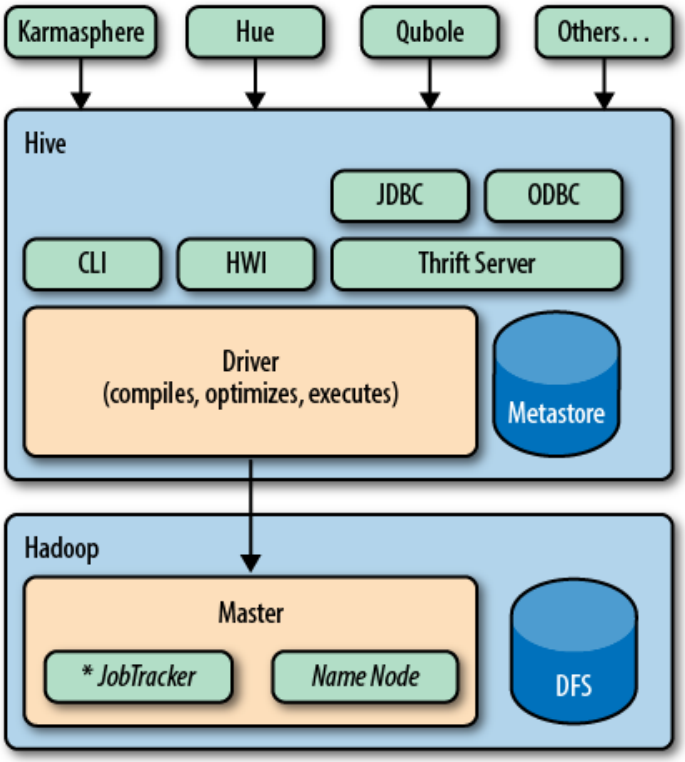
Hive not only provides a familiar programming model for people who know SQL, but also eliminates lots of routine and sometimes-tricky coding we would have to do in Java.

It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Whether the user is a DBA or a Java developer, Hive lets us complete a lot of work with relatively little effort.

Hive is built on top of Apache Hadoop, which is an open-source framework used to efficiently store and process large datasets. As a result, Hive is closely integrated with Hadoop, and is designed to work quickly on petabytes of data.

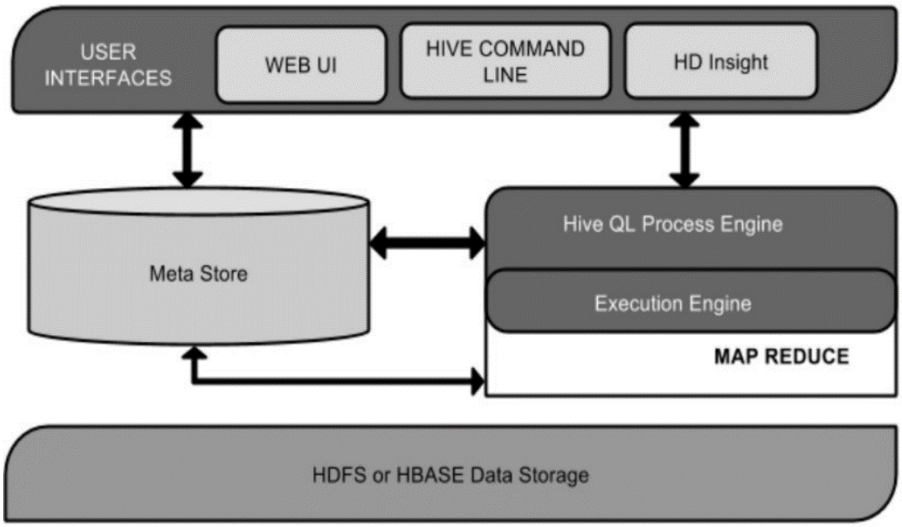
The following diagram depicts major modules of Hive and its association with Hadoop.



CLI, Hive web interface (HWI), JDBC, ODBC, Thrift Server are bundled with Hive Distribution. (A Thrift service provides remote access from other processes.)

Hive communicates with the ***JobTracker*** to initiate the MapReduce job. Hive does not have to be running on the same master node with the JobTracker.

In larger clusters, it is common to have edge nodes where tools like Hive run. They communicate remotely with the JobTracker on the master node to execute jobs.



Usually, the data files to be processed are in HDFS, which is managed by the *NameNode*.

User Interface –Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. There are various ways to interact with Hive either using CLI or GUI. (Kamasphere, Hue, Qubole are GUIs). HWI provides remote access to Hive.

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. The Meta Store is a separate relational database (usually a MySQL instance) where Hive persists table schemas and other system metadata.

HiveQL Process Engine – HiveQL is similar to SQL for querying on schema info on the Meta Store. 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 flavour of MapReduce.

HDFS or HBASE – Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Execution of Hive:

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.

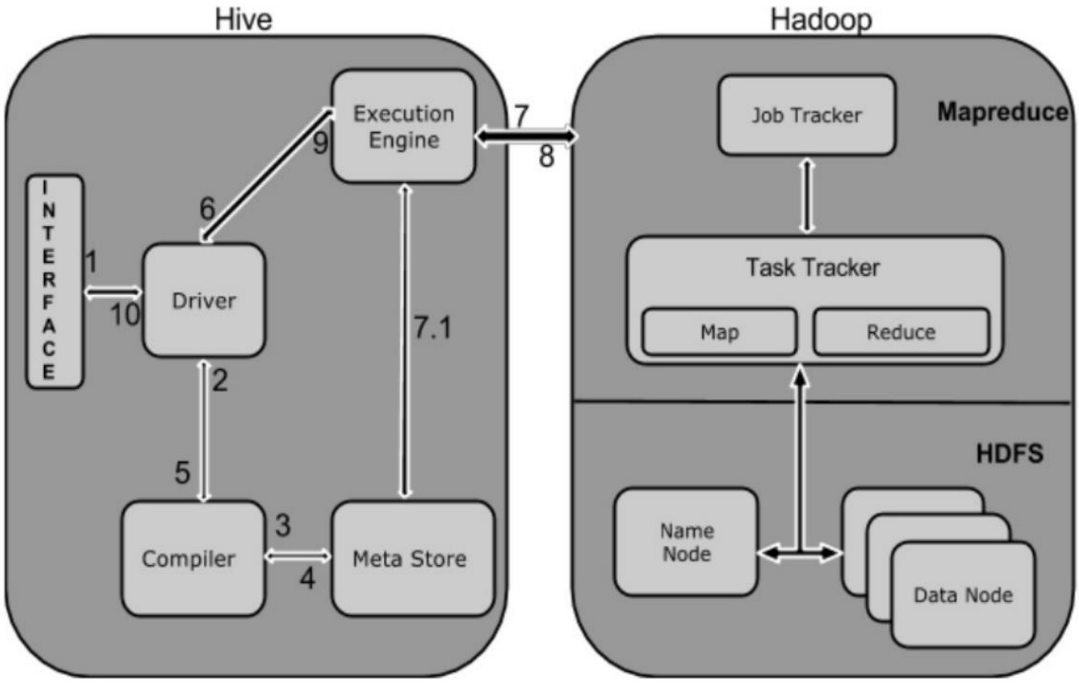
7 **Execute Job** - Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.

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.



Data Types and File Formats:

Hive supports many of the ***primitive*** data types we find in relational databases, as well as three ***collection*** data types that are rarely found in relational databases.

A unique feature of Hive, compared to most databases, is that it provides great flexibility in how data is encoded in files.

Most databases take total control of the data, both how it is stored to disk and its life cycle. However, Hive makes it easier to manage and process data with a variety of tools, by letting us control all these aspects.

Primitive Data Types

Hive supports several sizes of integer and floating-point types, a Boolean type, and character strings of arbitrary length.

It's useful to remember that each of these types is implemented in Java, so the particular behaviour details will be exactly what you would expect from the corresponding Java types. For example, STRING is implemented by the Java String, FLOAT is implemented by Java float, etc.

Type Size	Literal	Syntax examples
TINYINT	1 Byte signed integer	20
SMALLINT	2 Bytes signed integer	20
INT	4 Bytes signed integer	20
BIGINT	8 Bytes signed integer	20
BOOLEAN	Boolean (true or false)	TRUE
FLOAT	Single precision floating point	3.14159
DOUBLE	Double precision floating point	3.14159
STRING	Sequence of characters. The character set can be specified. Single or double quotes can be used.	'It is Hive' "It is Hive"
TIMESTAMP	Integer, float, or string. (Unix epoch seconds) POSIX standard	1327882394

Collection Data Types

Hive supports columns that are ***Arrays, Maps, Structs***.

Most relational databases do not support such collection types, because using them tends to break normal form. For example, in traditional data models, structs might be captured in separate tables, with foreign key relations between the tables, as appropriate.

A practical problem with breaking normal form is the greater risk of data duplication, leading to unnecessary disk space consumption and potential data inconsistencies, as duplicate copies can grow out of sync as changes are made. However, in Big Data systems, a benefit of sacrificing normal form is higher processing throughput.

Type	Description	Literal syntax examples
STRUCT	Analogous to a C `struct` or an "object." Fields can be accessed using the "dot" notation. For example, if a column `name` is of type STRUCT `{first STRING; last STRING}`, then the first name field can be referenced using `name.first`.	`struct('John', 'Doe')`
MAP	A collection of key-value tuples, where the fields are accessed using array notation (e.g., ['key']). For example, if a column `name` is of type MAP with key→value pairs `'first'→'John'` and `'last'→'Doe'`, then the last name can be referenced using `name['last']`.	`map('first', 'John', 'last', 'Doe')`
ARRAY	Ordered sequences of the *same* type that are indexable using zero-based integers. For example, if a column `name` is of type ARRAY of strings with the value `['John', 'Doe']`, then the second element can be referenced using `name[1]`.	`array('John', 'Doe')`

Here is a table declaration that demonstrates how to use these types, an *employees* table in a fictitious Human Resources application:

```
CREATE TABLE emp
(
        name STRING,
        salary FLOAT,
        subordinates ARRAY<STRING>,
        deductions MAP<STRING, FLOAT>,
        address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
);
```

The ***name*** column is a simple string which is the name of each employee and float is for employee's ***salary***.

Note that Java syntax conventions for generics are followed for the collection types. Deductions column is a map that holds the key-value pair for every deduction that will be taken out from employee's salary. For example, **MAP<STRING, FLOAT>** means that every key in the map will be of type STRING and every value will be of type FLOAT.

For an **ARRAY<STRING>,** every item in the array will be a STRING. STRUCTs can mix different types, but the locations are fixed to the declared position in the STRUCT.

Text File Encoding of Data Values

Let us begin our exploration of file formats by looking at the simplest example, **text files**.

We are familiar with text files delimited with commas or tabs, the so-called **comma-separated values (CSVs) or tab-separated values (TSVs),** respectively. Hive can use these formats if we want.

However, there is a drawback to both formats. We have to be careful about commas or tabs embedded in text and not intended as field or column delimiters. For this reason, Hive uses various control characters by default, which are less likely to appear in value

strings. Hive uses the term field when overriding the default delimiter. They are listed in the below Table.

Table 3-3. Hive's default record and field delimiters

Delimiter	Description
\n	For text files, each line is a record, so the line feed character separates records.
^A ("control" A)	Separates all fields (columns). Written using the octal code \001 when explicitly specified in CREATE TABLE statements.
^B	Separate the elements in an ARRAY or STRUCT, or the key-value pairs in a MAP. Written using the octal code \002 when explicitly specified in CREATE TABLE statements.
^C	Separate the key from the corresponding value in MAP key-value pairs. Written using the octal code \003 when explicitly specified in CREATE TABLE statements.

Example:

```
{
  "name":  "John Doe",
  "salary": 100000.0,
  "subordinates": ["Mary Smith", "Todd Jones"],
  "deductions": {
    "Federal Taxes": .2,
    "State Taxes":   .05,
    "Insurance":     .1
  },
  "address": {
    "street": "1 Michigan Ave.",
    "city":   "Chicago",
    "state":  "IL",
    "zip":    60600
  }
}
```

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

HiveQL - Data Definition:

Hive Query Language (HiveQL) is a query language in Apache Hive for processing and analyzing structured data.

It is perhaps closest to MySQL's dialect, but with significant differences.

Hive offers no support for row level inserts, updates, and deletes. Hive does not support transactions.

Data definition language of HiveQL include creating, altering, dropping databases and tables.

Create Database

The Hive concept of a database is essentially just a catalog or namespace of tables.

They are very useful for larger clusters with multiple teams and users, as a way of avoiding table name collisions.

If we do not specify a database, the **default** database is used.

The simplest syntax for creating a database is shown in the following example:

```
CREATE DATABASE <database name>

hive> CREATE DATABASE test;
```

Hive will throw an error if **test** already exists. we can suppress these warnings with the following syntax.

```
CREATE DATABASE [IF NOT EXISTS] <database name>

hive> CREATE DATABASE IF NOT EXISTS test;
```

We can also use the keyword SCHEMA instead of DATABASE in all the database-related commands.

```
hive> CREATE SCHEMA test;
```

At any time, we can see the databases that already exist using **SHOW** as follows:

```
hive> SHOW DATABASES;
default
test

hive> CREATE DATABASE userdb;
hive> SHOW DATABASES;
default
test
userdb
```

https://www.youtube.com/watch?v=4b4v3685JuU (Internal/External Tables)

https://www.youtube.com/watch?v=AXvGkP20CDs (Basic Commands for Tables)

Hive will create a **directory** for each database.

Tables in that database will be stored in **subdirectories** of the database directory. The **exception** for this is tables in the **default** database, which doesn't have its own directory.

The database directory is created under a top-level directory specified by the property **hive.metastore.warehouse.dir**. Assuming we are using the default value for this property, **/user/hive/warehouse**, when the **test** database is created, the Hive will create the directory **/user/ hive/ warehouse/test.db.**

We can **override this default location** for the new directory as shown in this example:

```
hive> CREATE DATABASE test
    > LOCATION '/my/preferred/directory';
```

The **USE** command sets a database as our working database, similar to changing working directories in a filesystem:

```
hive> USE test;
```

Alter Database

DESCRIBE command can be used to show the description of the database, along with its location.

```
hive> CREATE DATABASE test
    > COMMENT 'Holds all testing tables';

hive> DESCRIBE DATABASE test;
test Holds all testing tables
hdfs://master-server/user/hive/warehouse/test.db
```

We can associate key-value properties with the database, although their only function is to provide a way of adding information to the database.

```
hive> CREATE DATABASE test
    > WITH DBPROPERTIES ('creator' = 'MCA', 'date' = '2023-09-21');

hive> DESCRIBE DATABASE test;
test hdfs://master-server/user/hive/warehouse/test.db

hive> DESCRIBE DATABASE EXTENDED test;
test hdfs://master-server/user/hive/warehouse/test.db
{date=2023-09-21, creator=MCA);
```

We can set key-value pairs in the DBPROPERTIES associated with a database using the **ALTER DATABASE** command. **No other metadata** about the database can be changed, including its name and directory location. There is no way to delete or "unset" a DBPROPERTY.

```
hive> ALTER DATABASE test SET DBPROPERTIES ('edited-by' = 'BDA');
```

Creating Tables

In Hive, we can create a table by using the conventions similar to the SQL. HiveQL supports a wide range of flexibility where the data files for tables are stored.

It provides two types of tables:

- Internal table/Managed Table
- External table

Internal Table/Managed Table

By default, Hive creates internal tables. The internal tables are also called **managed tables** as the lifecycle of their data is controlled by the Hive. By default, these tables are stored in a subdirectory under the directory defined by **hive.metastore.warehouse.dir** (i.e. **/user/hive/warehouse**). The internal tables are not flexible enough to share with other tools like Pig. If we try to drop the internal table, Hive deletes both table schema and data. ACID properties can be applied for Internal Tables. The data is available within the local file system. Managed tables are less convenient for sharing with other tools.

External Table

The external table allows us to create and access a table and data externally. Hive does not manage external table. The external keyword is used to specify the external table. The stored location is HDFS. As the table is external, the data is not present in the Hive directory/MetaStore. Therefore, if we try to drop the table, the metadata of the table will be deleted, but the data still exists. ACID properties cannot be applied for Internal Tables. The data is available within the HDFS. Because it's external, Hive does not assume it owns the data. Therefore, dropping the table does not delete the data, although the metadata for the table will be deleted. With External tables, we can share the data across the tools.

Below are the major differences between Internal vs External tables in Apache Hive.

INTERNAL OR MANAGED TABLE	EXTERNAL TABLE
By default, Hive creates an Internal or Managed Table.	Use EXTERNAL option/clause to create an external table
Hive owns the metadata, table data by managing the lifecycle of the table	Hive manages the table metadata but not the underlying file.
Dropping an Internal table drops metadata from Hive Metastore and files from HDFS	Dropping an external table drops just metadata from Metastore without touching actual file on HDFS.
Supports ACID/Transactional	Not supported

```
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]
```

Case 1:

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>);
```

Case 2:

```
CREATE TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Case 3:

```
CREATE TABLE IF NOT EXISTS mydb.employees (
  name         STRING COMMENT 'Employee name',
  salary       FLOAT  COMMENT 'Employee salary',
  subordinates ARRAY<STRING> COMMENT 'Names of subordinates',
  deductions   MAP<STRING, FLOAT>
               COMMENT 'Keys are deductions names, values are percentages',
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
               COMMENT 'Home address')
COMMENT 'Description of the table'
TBLPROPERTIES ('creator'='me', 'created_at'='2012-01-02 10:00:00', ...)
LOCATION '/user/hive/warehouse/mydb.db/employees';
```

IF NOT EXISTS – You can use IF NOT EXISTS to avoid the error in case the table is already present. Hive checks if the requesting table already presents.
EXTERNAL – Used to create external table.
TEMPORARY – Used to create temporary table.
ROW FORMAT – Specifies the format of the row.
LOCATION – You can specify the custom location where to store the data on HDFS.
STORED AS – File formats - ORC, Parquet, Text etc.

Note that we can **prefix** a database name, **mydb** in this case, which is a one of the best practices to do so as we can specify the table name in the respective database.

Hive **automatically adds two table properties**: **last_modified_by** holds the username of the last user to modify the table, and **last_modified_time** holds the epoch time in seconds of that modification.

By default, Hive always creates the table's directory under the directory for the enclosing database. The exception is the default database. It doesn't have a directory under /user/hive/warehouse, so a table in the default database will have its directory created directly in /user/hive/warehouse (unless explicitly overridden).

Finally, we can optionally specify a location for the table data (as opposed to metadata, which the metastore will always hold). In this example, we are showing the default location that Hive would use, /user/hive/warehouse/mydb.db/employees, where /user/hive/warehouse is the default "warehouse" location, mydb.db is the database directory, and employees is the table directory.

To avoid potential confusion, it's usually better to use an external table if we don't want to use the default location table.

You can also **copy the schema** (but not the data) of an existing table. Note that no other properties, including the schema, can be re-defined and they are determined from the original table.

```
CREATE TABLE IF NOT EXISTS mydb.employeesnew
LIKE mydb.employees;
```

The **SHOW TABLES** command lists the tables. With no additional arguments, it shows the tables in the current working database. Let uss assume we have already created a few other tables, table1 and table2, and we did so in the **mydb** database:

```
hive> USE mydb;
hive> SHOW TABLES;
employees
table1
table2
```

If we are not in the same database, we can still list the tables in that database:

```
hive> USE default;
hive> SHOW TABLES IN mydb;
employees
table1
table2
```

We can also use the **DESCRIBE EXTENDED** mydb.employees command to show details about the table. (mydb can be omitted if we are using the mydb database at present).

```
hive> DESCRIBE EXTENDED mydb.employees;
name    string  Employee name
salary  float   Employee salary
subordinates    array<string>   Names of subordinates
deductions      map<string,float> Keys are deductions names, values are percentages
address struct<street:string,city:string,state:string,zip:int>  Home address

Detailed Table Information      Table(tableName:employees, dbName:mydb, owner:me,
...
location:hdfs://master-server/user/hive/warehouse/mydb.db/employees,
parameters:{creator=me, created_at='2012-01-02 10:00:00',
            last_modified_user=me, last_modified_time=1337544510,
            comment:Description of the table, ...}, ...)
```

Replacing EXTENDED with FORMATTED provides more readable but also more verbose output.

The first section shows the output of DESCRIBE without EXTENDED or FORMATTED (i.e., the schema including the comments for each column).

External Tables:

The external table allows us to create and access a table and data externally by other tools.

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
exchange STRING,
symbol STRING,
ymd STRING,
price_open FLOAT,
price_high FLOAT,
price_low FLOAT,
price_close FLOAT,
volume INT,
price_adj_close FLOAT)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/stocks';
```

The **EXTERNAL** keyword tells Hive this table is external and the **LOCATION** clause is required to tell Hive where it is located.

Because it is external, Hive does not assume it owns the data of external table. Therefore, dropping the table does not delete the data, although the metadata for the table will be deleted.

We can know whether or not a table is managed or external using the output of **DESCRIBE EXTENDED tablename**. Near the end of the Detailed Table Information output, we will see the following for managed tables:
... tableType:MANAGED_TABLE)

For external tables, you will see the following:
... tableType:EXTERNAL_TABLE)

As for managed tables, you can also copy the schema (but not the data) of an existing table:

```
CREATE EXTERNAL TABLE IF NOT EXISTS mydb.employees3
LIKE mydb.employees
LOCATION '/path/to/data';
```

Partitioned Managed Tables:

https://www.youtube.com/watch?v=RFS4JJFlzGc&t=5s (Hive Partitions)

Hive partitions are used to split the larger table into several smaller parts based on one or multiple columns (partition key - for example, date, state, country etc.).

The Hive partition is similar to table partitioning available in SQL server or any other RDBMS database tables.

These smaller logical tables are not visible to users and users still access the data from just one table.

Partition eliminates creating smaller tables, accessing, and managing them separately.

Partition tables can be either **static or dynamic**. By default, Hive creates static partitions. In static partitioning, we need to specify the partition column value in each and every LOAD statement while dynamic partition allows us not to specify partition column value each time.

When we load the data into the partition table, Hive internally splits the records based on the partition key and stores each partition data into a sub-directory of tables directory on HDFS. The name of the directory would be partition key and it's value.

Also, while loading the data into the partition table, Hive eliminates the partition key from the actual loaded file on HDFS as it is redundant information and could be get from the partition folder name.

Partition tables have certain performance benefits and they can help organize data in a logical fashion, such as hierarchically. Perhaps the most important reason to partition data is for faster queries.

To create a Hive table with partitions, you need to use **PARTITIONED BY** clause along with the column you wanted to partition and its type.

Consider the **following non-partition table** to begin with.

```
create table user_data_no_partition
(
sno int,
usr_name string,
city string)
ROW FORAMT delimited fields terminated by ' , ' LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Here, a non-partition managed table is created as usual.

Let us consider creating a partitioned managed table as below.

```
create table user_data
(
sno int,
usr_name string
)partitioned by (city string);
```

Observe, how partitioned table is created for city attribute compared to non-partition table.

Once created, the partition keys (city, in this case) behave like regular columns.

The data can be only inserted into a Partitioned table and data cannot be loaded into a partitioned table.

Assume the data is inserted into the table with city as bengaluru.

After inserting the data into the Hive partition table, you can use **SHOW PARTITIONS** command to see all partitions that are present.

```
SHOW PARTITIONS user_data
city=bengaluru
```

The DESCRIBE EXTENDED command shows the partition keys.

We can use **partitioning with external tables** as well. Consider the following external partitioned table.

```
CREATE EXTERNAL TABLE IF NOT EXISTS stocks (
        exchange STRING,
        symbol STRING,
        price_open FLOAT,
        price_high FLOAT,
        price_low FLOAT,
        price_close FLOAT,
        volume INT,
        price_adj_close FLOAT)
        PARTITIONED BY (year INT, month INT, day INT)
        ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
```

Alter Tables:

Most table properties can be altered with ALTER TABLE statements, which change metadata about the table but not the data itself. These statements can be used to fix mistakes in schema, move partition locations

Renaming a Table

Use this statement to rename the table log_messages to logmsgs:

```
ALTER TABLE log_messages RENAME TO logmsgs;
```

Adding, Modifying, and Dropping a Table Partition

ALTER TABLE table ADD PARTITION ⋯ is used to add a new partition to a table (usually an external table).

```
ALTER TABLE log_messages ADD IF NOT EXISTS
PARTITION (year = 2011, month = 1, day = 1) LOCATION '/logs/2011/01/01'
PARTITION (year = 2011, month = 1, day = 2) LOCATION '/logs/2011/01/02'
PARTITION (year = 2011, month = 1, day = 3) LOCATION '/logs/2011/01/03'
...;
```

Alter Table Properties

We can add additional table properties or modify existing properties, but not remove them:

```
ALTER TABLE log_messages SET TBLPROPERTIES (
'notes' = 'The process id is no longer captured; this column is always NULL');
```

Adding Columns

We can add new columns to the end of the existing columns, before any partition columns.

```
ALTER TABLE log_messages ADD COLUMNS (
app_name STRING COMMENT 'Application name',
session_id LONG COMMENT 'The current session id');
```

Changing Columns

We can rename a column, change its position, type, or comment:

```
ALTER TABLE log_messages
CHANGE COLUMN hms hours_minutes_seconds INT
COMMENT 'The hours, minutes, and seconds part of the timestamp'
AFTER severity;
```

We have to specify the old name, a new name, and the type, even if the name or type is not changing.

Dropping Tables:

The familiar **DROP TABLE** command from SQL is supported:

DROP TABLE IF **EXISTS** employees;

The IF EXISTS keywords are optional. If not used and the table doesn't exist, Hive returns an error.

For managed tables, both the table metadata and data are deleted.

For external tables, only the metadata is deleted but the data is not.

HiveQL: Data Manipulation

Loading Data into Managed Tables
Since Hive has no row-level insert, update, and delete operations, the only way to put data into a table is to use one of the "bulk" load operations.

It is conventional practice to specify a path that is a directory, rather than an individual file. Hive will copy all the files in the directory.

Consider

```
create table user_data
(
sno int,
usr_name string,
city string)
ROW FORAMT delimited fields terminated by ' , ' LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Now let us load data into this table.

LOAD DATA LOCAL INPATH '/home/test/usr_data.txt' into table user_data

LOAD DATA INPATH '/usr_data.txt' into table user_data

If the LOCAL keyword is used, the path is assumed to be in the local filesystem, say **Linux** file system. The data is copied into the final location.

If LOCAL is omitted, the path is assumed to be in the distributed filesystem, say **HDFS**. In this case, the data is moved from the path to the final location.

If you specify the OVERWRITE keyword, any data already present in the target directory will be deleted first. Without the keyword, the new files are simply added to the target directory.

Ex:

LOAD DATA LOCAL INPATH '/usr_data.txt' into table user_data

OVERWRITE **INTO TABLE** user_data

Inserting Data into Tables from Queries

https://sparkbyexamples.com/apache-hive/hive-insert-into-vs-insert-overwrite/

https://cwiki.apache.org/confluence/display/hive/languagemanual+dml

Hive supports two different types of insert HiveQL commands namely **INSERT INTO** and **INSERT OVERWRITE** to load data into tables and partitions.

```
create table user_data
(
sno int,
usr_name string,
city string)
```

Simple insert command to insert a single record/row into the table/partition. (Note: Only INSERT to be used to add data to a partitioned table. LOAD statement can not be used for a partitioned table)

```
Ex: INSERT INTO user_data VALUES
(10, 'MCA', 'BENGALURU');
```

Insert command to insert a multiple records/rows into the table/partition.

```
Ex: INSERT INTO user_data VALUES
(10, 'MCA', 'BENGALURU'),
(11,'BCA',MYSURU');
```

Insert data into selected columns.

```
Ex: INSERT INTO test.user_data(sno,city) VALUES
(12, 'DELHI');
```

Here, Since we are not inserting the data into age and gender columns, these columns inserted with NULL values.

Also, the INSERT statement lets us load data into a table from a query.

```
CREATE TABLE user_data_temp LIKE user_data;
INSERT INTO user_data_temp SELECT * FROM user_data;
```

Here, SELECT statement can be any valid SELECT query; for example we can add WHERE condition to the SELECT query to filter the rows.

Next, INSERT OVERWRITE is used to replace any existing data in the table or partition and insert with the new rows.

```
INSERT OVERWRITE TABLE  test.user_data VALUES
(13, 'MBA', 'BENGALURU'),
```

With OVERWRITE, any previous contents of the table/partition (or whole table if not partitioned) are replaced.
If we omit the keyword OVERWRITE or replace it with INTO, Hive appends the data rather than replacing it.

Dynamic Partition Inserts

https://www.geeksforgeeks.org/overview-of-dynamic-partition-in-hive/

By default, Hive creates Static partitions.

Dynamic partitioning is the strategic approach to load the data from the non-partitioned table where the single insert to the partition table.

In dynamic partitioning, the values of the partitioned table are existed by default so there is no need to pass the value for those columns manually.

If we want to perform partition on the tables without knowing the number of columns, in that case we can use Dynamic partitioning.

Dynamic partitioning is not enabled by default.

When it is enabled, it works in "strict" mode by default, where it expects at least some columns to be static.

We need to enable the dynamic partition using the following commands.

hive> set hive.exec.dynamic.partition=true;

hive> set hive.exec.dynamic.partition.mode=nonstrict;

Let us consider creating a static partitioned table as below.

```
create table user_data_no_partition
(
sno int,
usr_name string,
city string)
ROW FORAMT delimited fields terminated by ' , ' LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Assume data is loaded into user_data_no_partition table.

```
create table user_data
(
sno int,
usr_name string
)partitioned by (city string);
```

(Note: Only INSERT to be used to add data to a partitioned table. LOAD statement cannot be used for a partitioned table)

```
insert into table user_data partition (city='bengaluru') select sno, usr_name from
user_data_no_partition where city='bengaluru';
```

Let us create table with dynamic partition as below. As we observe the creation of static and dynamic partitions are same. No change.

```
create table user_data_dynamic
(
sno int,
usr_name string
)partitioned by (city string);
```

We need to enable the dynamic partition using the following commands.

```
hive> set hive.exec.dynamic.partition=true;

hive> set hive.exec.dynamic.partition.mode=nonstrict;

insert into table user_data_dynamic partition (city) select sno, usr_name, city
from user_data_no_partition where city='bengaluru';
```