What is Hive?

- Hive is a data warehouse system for data summarization, analysis and for querying of large data.
- It converts SQL-like queries into MapReduce jobs for easy execution and processing of extremely large volumes of data.
- Hive can process structured and semi-structured data by using a SQL-like language called HQL (Hive Query Language).

Hive is NOT:

Sometimes, few misconceptions occur about Hive. So, let's clarify that:

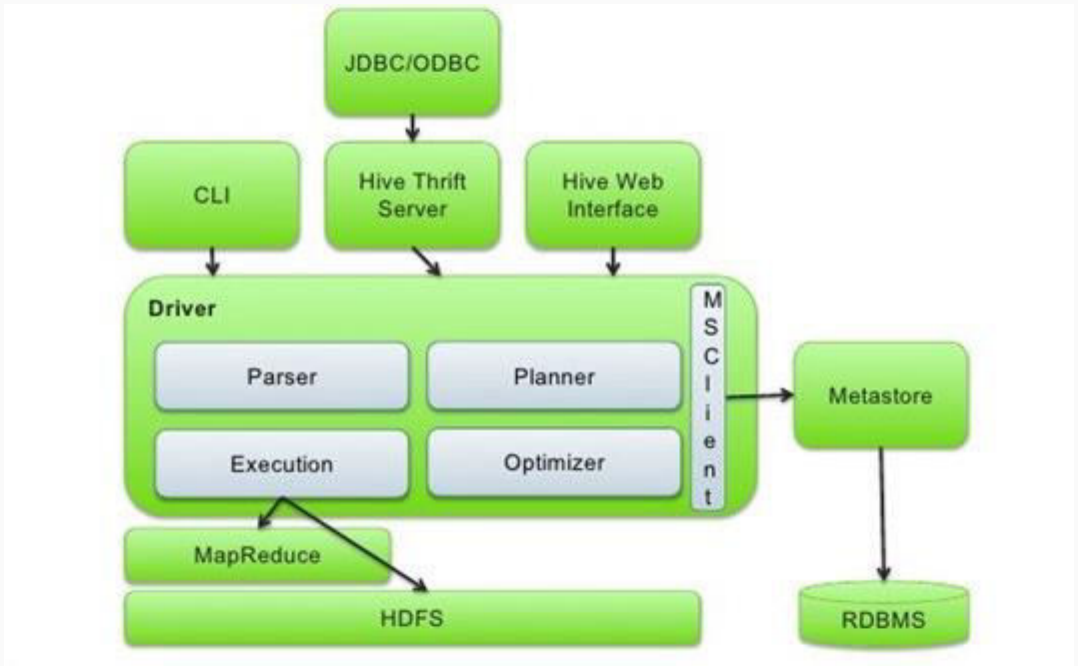
- It is not a relational database.
- It is not a design for Online Transaction Processing (OLTP).
- It is not a language for real-time queries and row-level updates.

Why Apache Hive?

- Facebook had faced a lot of challenges before implementation of Apache Hive. Challenges like the size of data being generated increased or exploded, making it very difficult to handle them.
- The traditional RDBMS could not handle the pressure. As a result, Facebook was looking out for better options. To overcome this problem, Facebook initially tried using MapReduce.
- But it had difficulty in programming, making it an impractical solution. Hence, Apache Hive allowed them to overcome the challenges they were facing.
- Apache Hive saves developers from writing complex Hadoop MapReduce jobs for ad-hoc requirements. Hence, hive provides summarization, analysis, and query of data. Hive is very fast and scalable.
- It is highly extensible. Since Apache Hive is similar to SQL, it becomes very easy for the SQL developers to learn and implement Hive Queries.
- Hive reduces the complexity of MapReduce by providing an interface where the user can submit SQL queries.
- So, now business analysts can play with Big Data using Apache Hive and generate insights. The most important feature of Apache Hive is that to learn Hive we don't have to learn Java.

Hive Architecture:

After the introduction to Apache Hive, now we are going to discuss the major component of Hive Architecture.

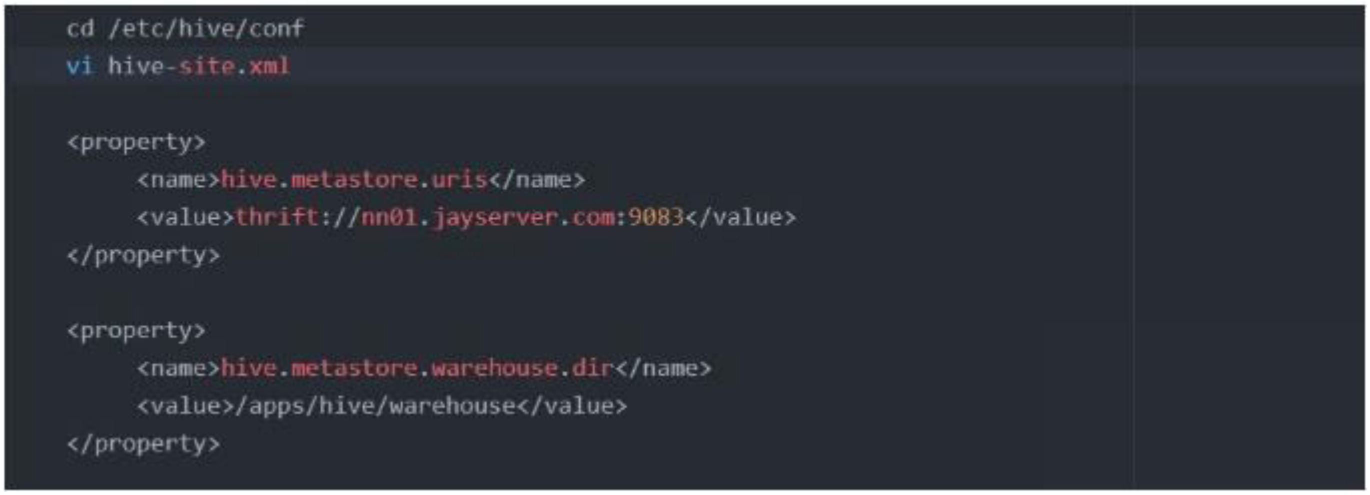


There are 4 main components as part of Hive Architecture.

- Hadoop core components (Hdfs, MapReduce)
- Metastore
- Driver
- Hive Clients

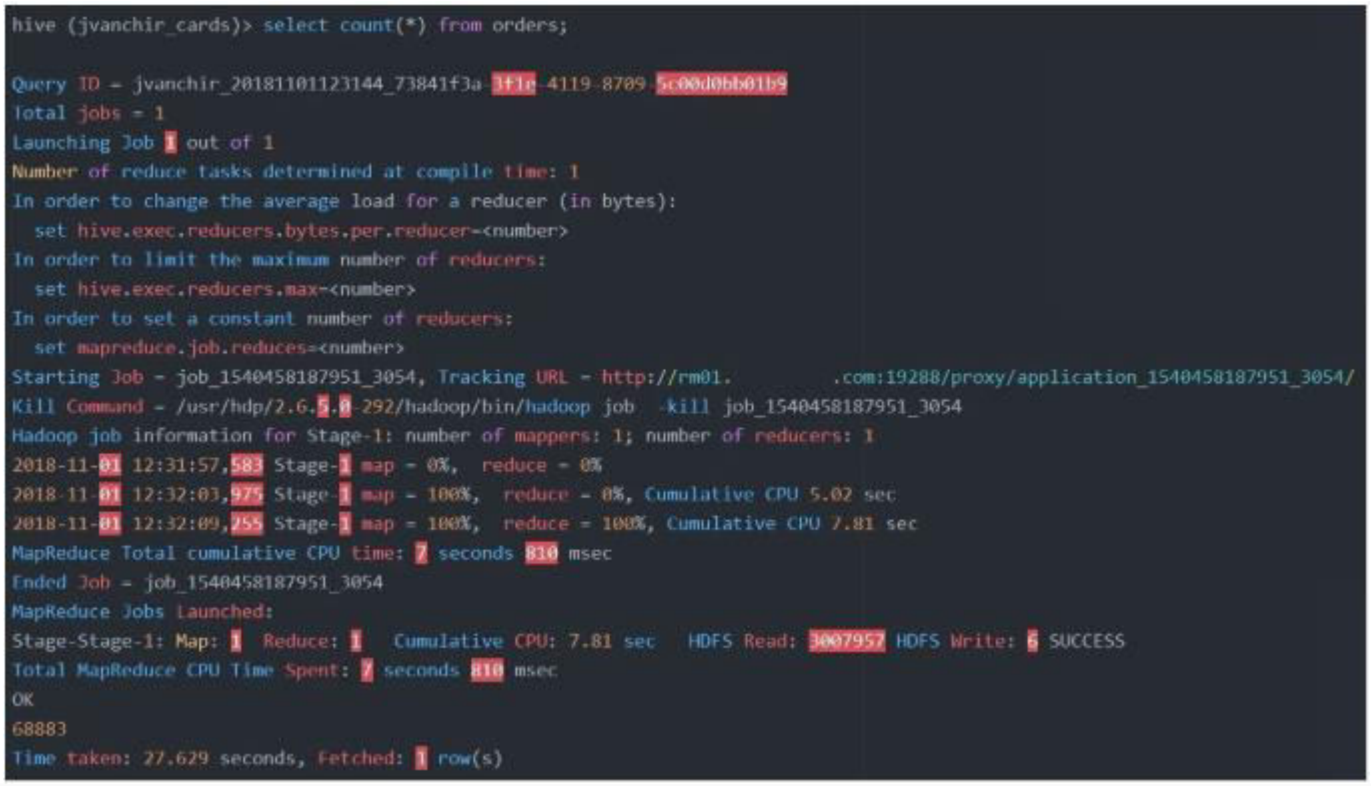
1) Hadoop core components:

a) HDFS: When we load the data into a Hive Table it internally stores the data in HDFS path i.e. by default in hive warehouse directory. The hive default warehouse location can be found at:



```
cd /etc/hive/conf
vi hive-site.xml

<property>
      <name>hive.metastore.uris</name>
      <value>thrift://nn01.jayserver.com:9083</value>
</property>

<property>
      <name>hive.metastore.warehouse.dir</name>
      <value>/apps/hive/warehouse</value>
</property>
```

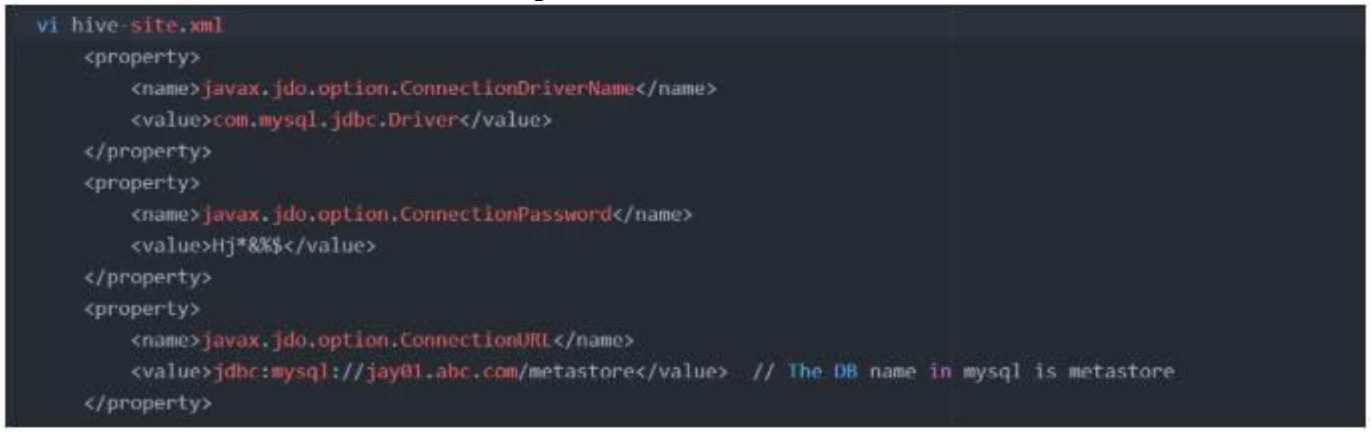
b) MapReduce: When we Run the below query, it will run a Map Reduce job by converting or compiling the query into a java class file, building a jar and execute this jar file.



```
hive (jvanchir_cards)> select count(*) from orders;

Query ID = jvanchir_20181101123144_73841f3a-3f1e-4119-8709-5c00d0bb01b9
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1540458187951_3054, Tracking URL = http://rm01.         .com:19288/proxy/application_1540458187951_3054/
Kill Command = /usr/hdp/2.6.5.0-292/hadoop/bin/hadoop job  -kill job_1540458187951_3054
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2018-11-01 12:31:57,583 Stage-1 map = 0%,  reduce = 0%
2018-11-01 12:32:03,975 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 5.02 sec
2018-11-01 12:32:09,255 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 7.81 sec
MapReduce Total cumulative CPU time: 7 seconds 810 msec
Ended Job = job_1540458187951_3054
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 7.81 sec   HDFS Read: 3007957 HDFS Write: 6 SUCCESS
Total MapReduce CPU Time Spent: 7 seconds 810 msec
OK
68883
Time taken: 27.629 seconds, Fetched: 1 row(s)
```

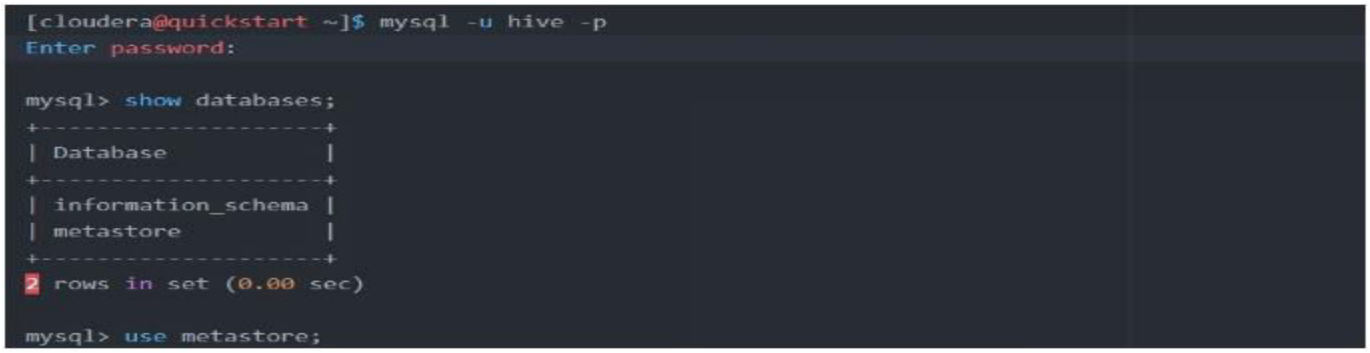
2) Metastore: It is a namespace for tables. This is a crucial part of hive as all the metadata information related to the hive such as detail of tables, columns, partitions and locations is present here. Usually, the Metastore is available as part of the Relational databases e.g. MySQL.

You can check the Database configuration via hive-site.xml



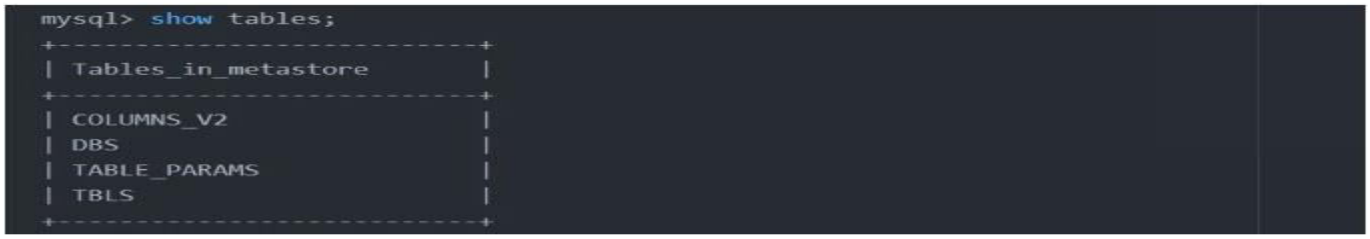
```
  vi hive-site.xml
      <property>
          <name>javax.jdo.option.ConnectionDriverName</name>
          <value>com.mysql.jdbc.Driver</value>
      </property>
      <property>
          <name>javax.jdo.option.ConnectionPassword</name>
          <value>Hj*&%$</value>
      </property>
      <property>
          <name>javax.jdo.option.ConnectionURL</name>
          <value>jdbc:mysql://jay01.abc.com/metastore</value>  // The DB name in mysql is metastore
      </property>
```

The Metastore details can be found as shown below.



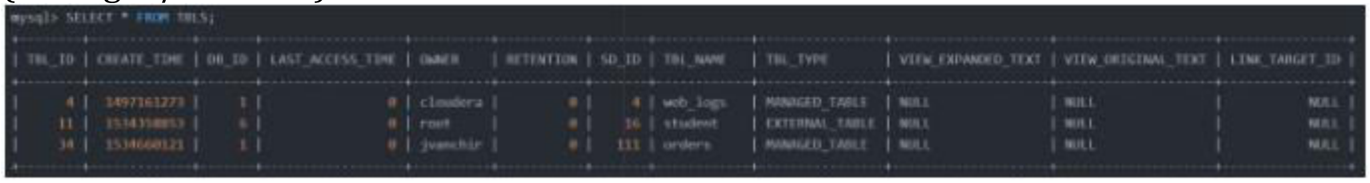
```
[cloudera@quickstart ~]$ mysql -u hive -p
Enter password:

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| metastore          |
+--------------------+
2 rows in set (0.00 sec)

mysql> use metastore;
```

Metastore has an overall 51 tables describing various properties related to the table but out of these 51 the below 3 tables are the ones which provide majority of the information, that in-turn helps hive infer the schema properties to execute the hive commands on a table.



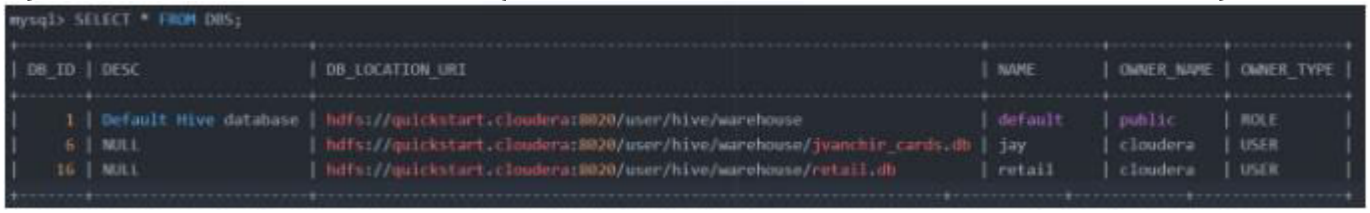
```
mysql> show tables;
+---------------------------+
| Tables_in_metastore       |
+---------------------------+
| COLUMNS_V2                |
| DBS                       |
| TABLE_PARAMS              |
| TBLS                      |
+---------------------------+
```

i) TBLS: Store all table information (Table name, Owner, Type of Table (Managed/External)



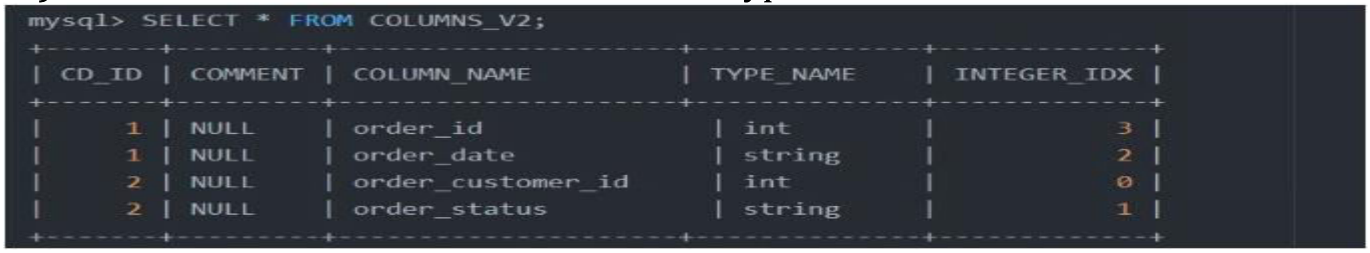
```
mysql> SELECT * FROM TBLS;
+--------+-------------+-------+------------------+----------+-----------+-------+----------+----------------+--------------------+--------------------+----------------+
| TBL_ID | CREATE_TIME | DB_ID | LAST_ACCESS_TIME | OWNER    | RETENTION | SD_ID | TBL_NAME | TBL_TYPE       | VIEW_EXPANDED_TEXT | VIEW_ORIGINAL_TEXT | LINK_TARGET_ID |
+--------+-------------+-------+------------------+----------+-----------+-------+----------+----------------+--------------------+--------------------+----------------+
|      4 |  1497161273 |     1 |                0 | cloudera |         0 |     4 | web_logs | MANAGED_TABLE  | NULL               | NULL               |           NULL |
|     11 |  1534358853 |     6 |                0 | root     |         0 |    16 | student  | EXTERNAL_TABLE | NULL               | NULL               |           NULL |
|     34 |  1534660121 |     1 |                0 | jvanchir |         0 |   111 | orders   | MANAGED_TABLE  | NULL               | NULL               |           NULL |
+--------+-------------+-------+------------------+----------+-----------+-------+----------+----------------+--------------------+--------------------+----------------+
```

ii) DBS: Database information (Location of database, Database name, Owner)



```
mysql> SELECT * FROM DBS;
+-------+-----------------------+--------------------------------------------------------------------------+---------+------------+------------+
| DB_ID | DESC                  | DB_LOCATION_URI                                                          | NAME    | OWNER_NAME | OWNER_TYPE |
+-------+-----------------------+--------------------------------------------------------------------------+---------+------------+------------+
|     1 | Default Hive database | hdfs://quickstart.cloudera:8020/user/hive/warehouse                      | default | public     | ROLE       |
|     6 | NULL                  | hdfs://quickstart.cloudera:8020/user/hive/warehouse/jvanchir_cards.db    | jay     | cloudera   | USER       |
|    16 | NULL                  | hdfs://quickstart.cloudera:8020/user/hive/warehouse/retail.db            | retail  | cloudera   | USER       |
+-------+-----------------------+--------------------------------------------------------------------------+---------+------------+------------+
```

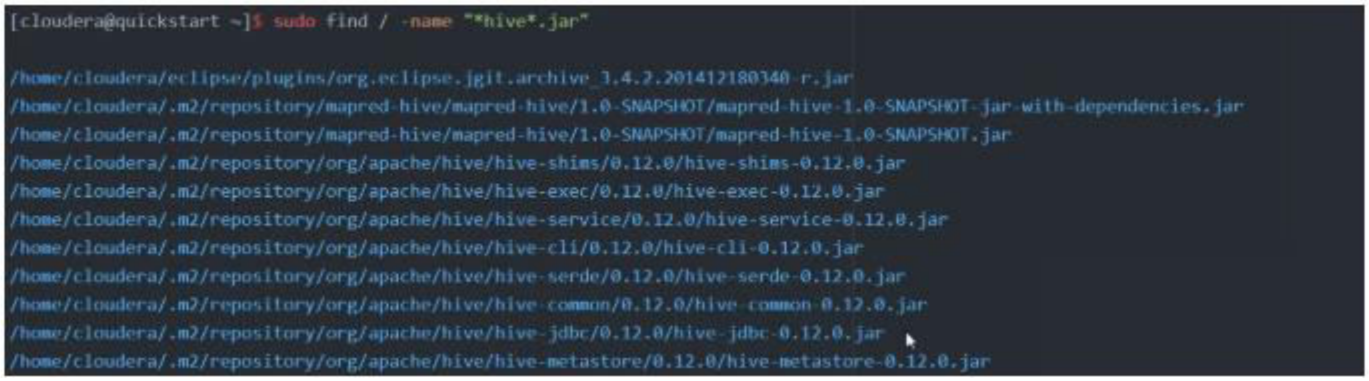
iii) COLUMNS_V2: column name and the datatype



```
mysql> SELECT * FROM COLUMNS_V2;
+-------+---------+--------------------+-----------+-------------+
| CD_ID | COMMENT | COLUMN_NAME        | TYPE_NAME | INTEGER_IDX |
+-------+---------+--------------------+-----------+-------------+
|     1 | NULL    | order_id           | int       |           3 |
|     1 | NULL    | order_date         | string    |           2 |
|     2 | NULL    | order_customer_id  | int       |           0 |
|     2 | NULL    | order_status       | string    |           1 |
+-------+---------+--------------------+-----------+-------------+
```

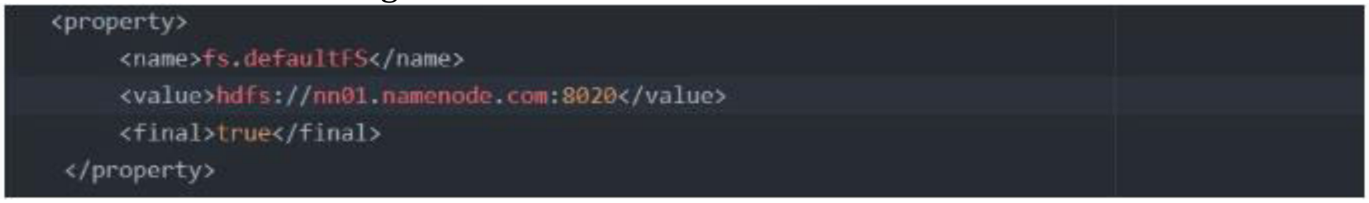
Note: In real-time the access to metastore is restricted to the Admin and specific users.

3) Driver: The component that parses the query, does semantic analysis on the different query blocks and query expressions eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore. The execution plan created by the compiler is a DAG of stages.
A bunch of jar files that are part of hive package help in converting HQL queries into equivalent MapReduce jobs and execute them on MapReduce.



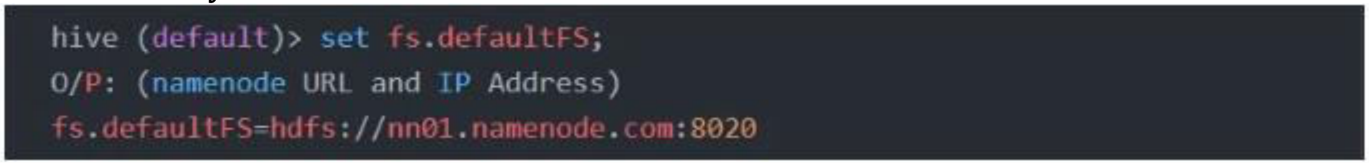
```
[cloudera@quickstart ~]$ sudo find / -name "*hive*.jar"

/home/cloudera/eclipse/plugins/org.eclipse.jgit.archive_3.4.2.201412180340-r.jar
/home/cloudera/.m2/repository/mapred-hive/mapred-hive/1.0-SNAPSHOT/mapred-hive-1.0-SNAPSHOT-jar-with-dependencies.jar
/home/cloudera/.m2/repository/mapred-hive/mapred-hive/1.0-SNAPSHOT/mapred-hive-1.0-SNAPSHOT.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-shims/0.12.0/hive-shims-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-exec/0.12.0/hive-exec-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-service/0.12.0/hive-service-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-cli/0.12.0/hive-cli-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-serde/0.12.0/hive-serde-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-common/0.12.0/hive-common-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-jdbc/0.12.0/hive-jdbc-0.12.0.jar
/home/cloudera/.m2/repository/org/apache/hive/hive-metastore/0.12.0/hive-metastore-0.12.0.jar
```

To check if the hive can talk to the appropriate cluster. i.e for hive to interact, query or execute with the existing cluster. You can check details under core-site.xml.



```
  <property>
       <name>fs.defaultFS</name>
       <value>hdfs://nn01.namenode.com:8020</value>
       <final>true</final>
   </property>
```

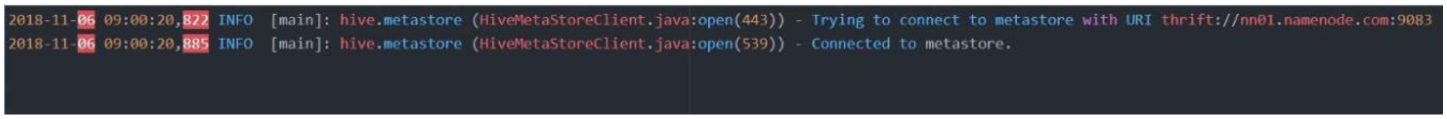
You can verify the same from hive as well.



```
hive (default)> set fs.defaultFS;
O/P: (namenode URL and IP Address)
fs.defaultFS=hdfs://nn01.namenode.com:8020
```

4) Hive Clients: It is the interface through which we can submit the hive queries. E.g. hive CLI, beeline are some of the terminal interfaces. We can also use the Web-interface like Hue, Ambari to perform the same.

On connecting to hive via CLI or Web-Interface it establishes a connection to Metastore as well.



```
2018-11-06 09:00:20,822 INFO  [main]: hive.metastore (HiveMetaStoreClient.java:open(443)) - Trying to connect to metastore with URI thrift://nn01.namenode.com:9083
2018-11-06 09:00:20,885 INFO  [main]: hive.metastore (HiveMetaStoreClient.java:open(539)) - Connected to metastore.
```

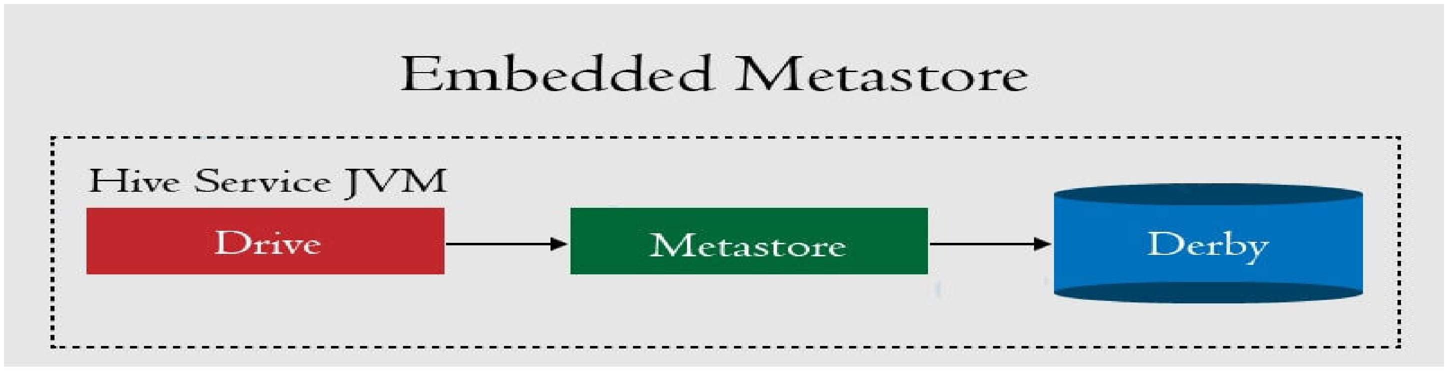
Metastore:

Metastore is the component that stores all structure information of various tables and partitions in the warehouse including column and column type information. Also, serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.

Hive Metastore Modes: There are three modes for Hive Metastore:

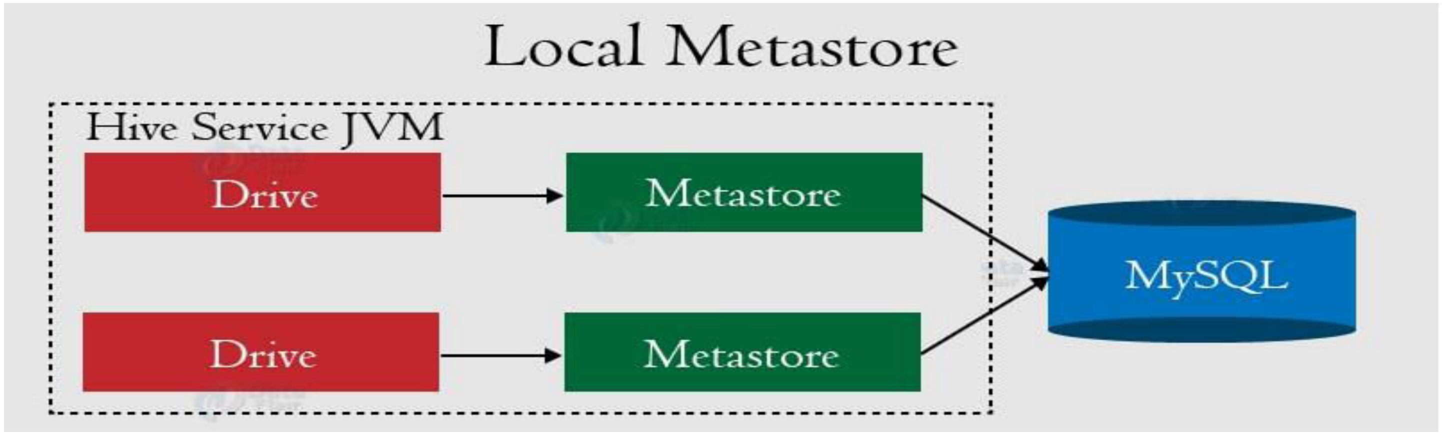
- Embedded Metastore
- Local Metastore
- Remote Metastore

1) Embedded Metastore:



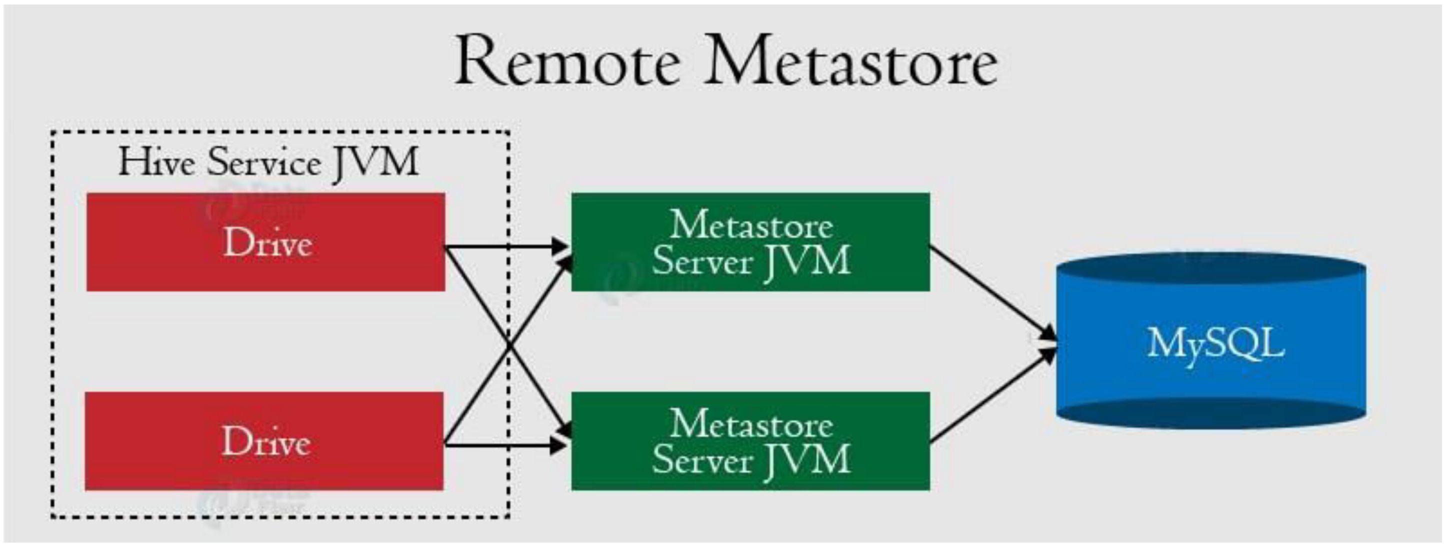
- By default, Hive Metastore Service runs in the same JVM as the Hive Service.
- In this mode, it uses embedded derby database stored on the local file system.
- Both Metastore Service and Hive Service runs in the same JVM by using embedded derby database.
- Its limitation is only one embedded derby database can access the database files on the disk at any given time, so only one Hive session could be open at a time.

2) Local Metastore:



- Hive is a data-warehousing framework, so it does not prefer single session.
- This mode allows us to have many Hive sessions i.e. many users can use the metastore at the same time.
- We can achieve by using any JDBC compliant like MySQL which runs in a separate JVM or different machines than that of the Hive Service and Metastore Service which were running in the same JVM.
- This configuration is called local metastore because Metastore Service still runs in the same process as the Hive Service but it connects to a database running in a separate process either on the same machine or on a remote machine.

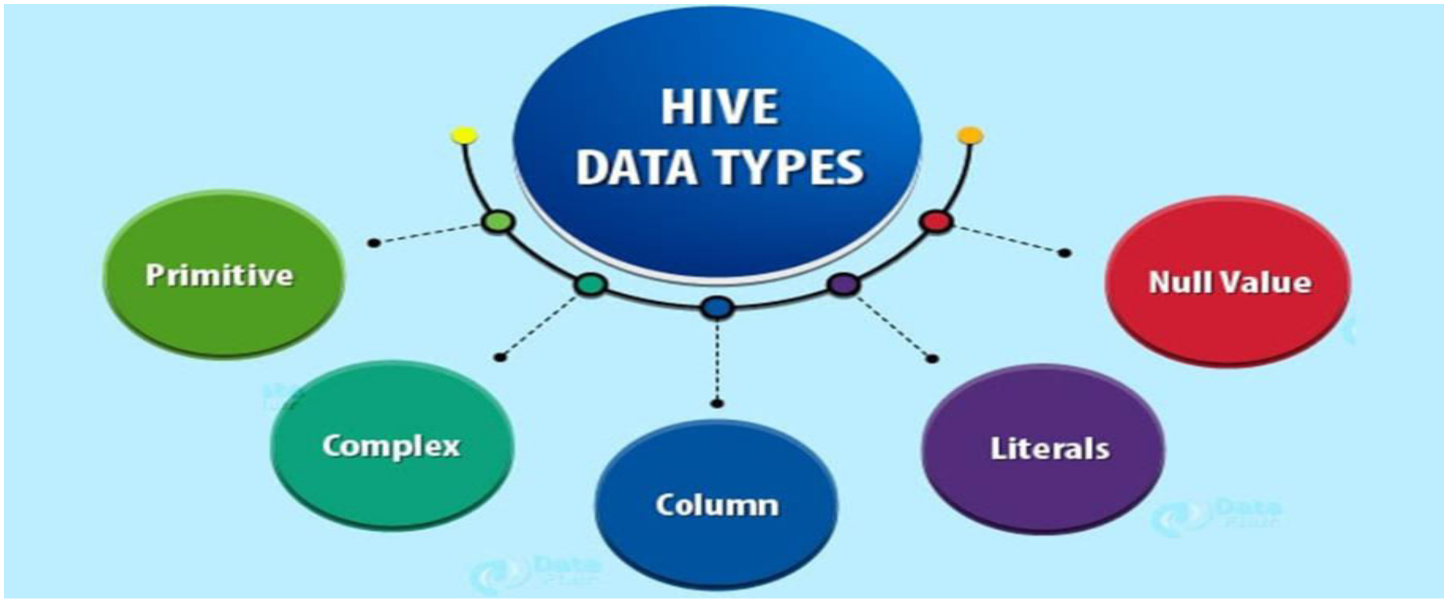
3) Remote Metastore:



- In this mode, Metastore Service runs on separate JVM but not in Hive Service JVM.
- If other processes wants to communicate with the Metastore Server they can communicate using Thrift Network APIs.
- To use this remote metastore, you should configure Hive Service by setting **hive.metastore.uris** to the Metastore Server URI(s).

Data Types:

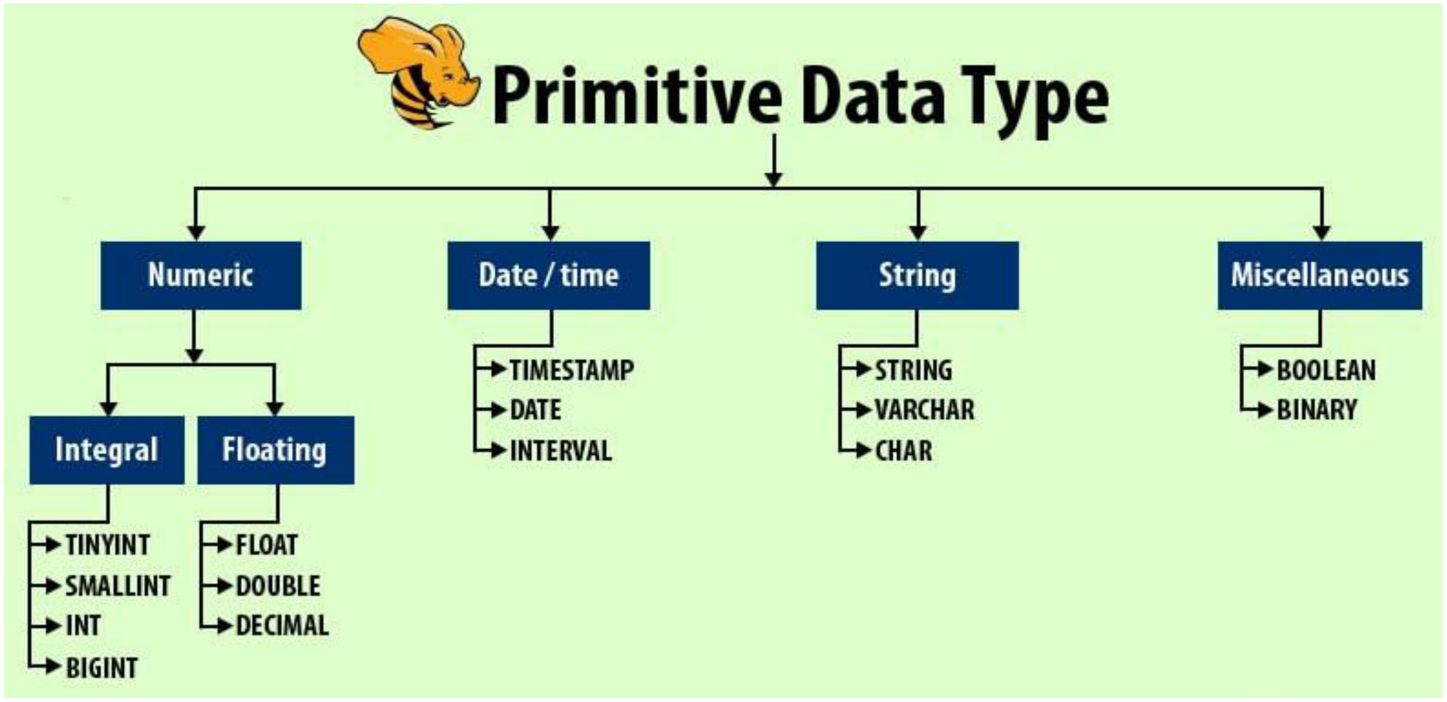
We will discuss different types of data types in Hive. Mainly Hive Data Types are classified into 5 major categories, let's discuss them one by one:



A) Primitive Data Types in Hive:

Primitive Data Types also divide into 4 types which are as follows:

- Numeric Data Type
- Date/Time Data Type
- String Data Type
- Miscellaneous Data Type



Let us now discuss these Hive Primitive data types one by one:

1) Numeric Data Type: The Hive Numeric Data types also classified into two types-

- Integral Data Types
- Floating Data Types

a) Integral Data Types: Integral Hive data types are as follows:

- **TINYINT** (1-byte (8 bit) signed integer, from -128 to 127)
- **SMALLINT** (2-byte (16 bit) signed integer, from -32, 768 to 32, 767)
- **INT** (4-byte (32-bit) signed integer, from –2,147,483,648to 2,147,483,647)
- **BIGINT** (8-byte (64-bit) signed integer, from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

b) Floating Data Types: Floating Hive data types are as follows-

- **FLOAT** (4-byte (32-bit) single-precision floating-point number)
- **DOUBLE** (8-byte (64-bit) double-precision floating-point number)
- **DECIMAL** (Arbitrary-precision signed decimal number)

2) Date/Time Data Type:
The second category of Apache Hive primitive data type is Date/Time data types. The following Hive data types comes into this category:

- **TIMESTAMP** (Timestamp with nanosecond precision)
- **DATE** (date)
- **INTERVAL**

3) String Data Type:
String data types are the third category under Hive data types. Below are the data types that come into this:

- **STRING** (Unbounded variable-length character string)
- **VARCHAR** (Variable-length character string)
- **CHAR** (Fixed-length character string)

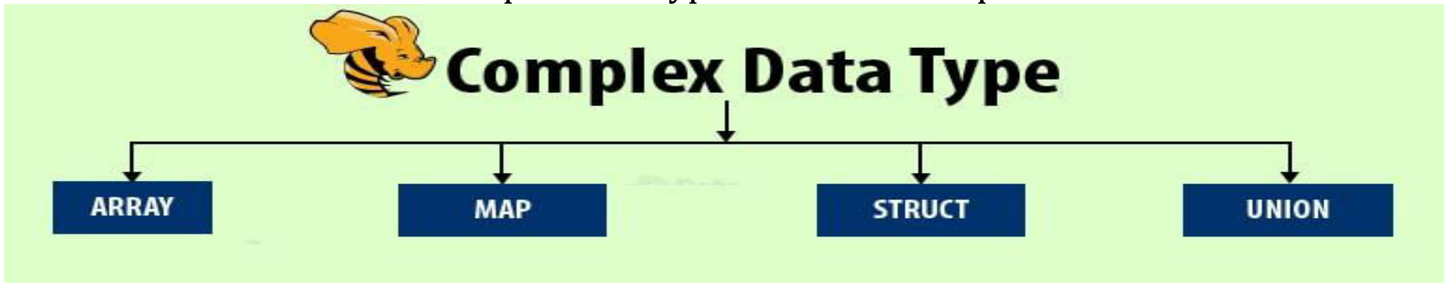
4) Miscellaneous Data Type:
Miscellaneous data types have two types of Hive data types-

- BOOLEAN (True/False value)
- BINARY (Byte array)

B) Complex Data Types in Hive: In this category of Hive data types following data types are come:

- Array
- MAP
- STRUCT
- UNION

Let's now discuss the Hive Complex data types with the example:



1) ARRAY: It's a collection of fields & all fields must be of the same type.
Syntax: ARRAY<Data_Type>
E.g. array (1, 2), array (laptop$mouse)

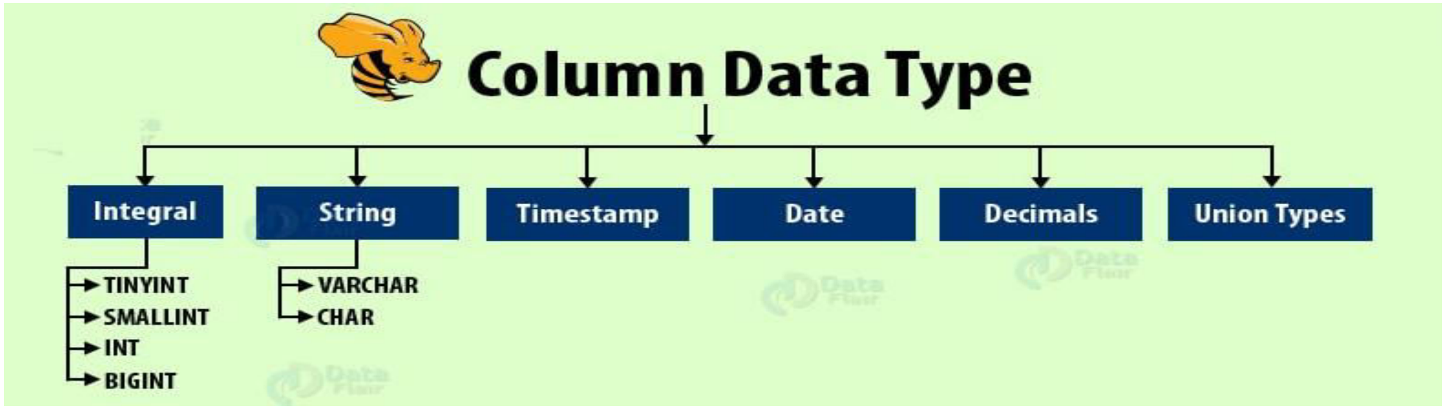
2) MAP: It's a collection of **key-value pairs**. Values from a map can be accessed using the keys.
Syntax: MAP<Primitive_Type, Data_Type>

3) STRUCT: It's a collection of fields & they can be of different types.
Syntax: STRUCT<col_name: Data_Type [COMMENT col_comment],.....>
E.g. struct ('col1', 'a', 'col2', 1, 'col3', 1.0)

C) Column Data Types in Hive: Column Hive data types are furthermore divide into 6 categories:

- Integral Type
- Strings
- Timestamp
- Dates
- Decimals
- Union Types

Let us discuss these Hive Column data types one by one:



1) Integral Type:

In this category of Hive data types following 4 data types are come-

- TINYINT
- SMALLINT
- INT/INTEGER
- BIGINT

By default, Integral literals are assumed to be **INT**. When the data range exceeds the range of INT, we need to use **BIGINT**. If the data range is smaller than the INT, we use **SMALLINT**. And **TINYINT** is smaller than SMALLINT.

Table	Postfix	Example
TINYINT	Y	100Y
SMALLINT	S	100S
BIGINT	L	100L

2) Strings:

The string data types in Hive, can be specified with either single quotes (') or double quotes ("). Apache Hive Use C-style escaping within the strings.

Data Type	**Length**
VARCHAR	1 TO 65355
CHAR	255

a) VARCHAR:

Varchar- Hive data types are created with a length specifier (between 1 and 65355). It defines the maximum number of characters allowed in the character string.

b) Char:

Char – Hive data types are similar to VARCHAR. But they are fixed-length meaning that values shorter than the specified length value are padded with spaces but trailing spaces are not important during comparisons. 255 is the maximum fixed length.

c) Timestamp:
Hive supports traditional UNIX timestamp with operational nanosecond precision. Timestamps in text files use format "YYYY-MM-DD HH:MM:SS.fffffffff" and "yyyy-mm-dd hh:mm:ss.ffffffffff".

d) Dates:
DATE values are described in particular year/month/day (YYYY-MM-DD) format. E.g. DATE '2017-01-01'. These types don't have a time of day component. This type supports range of values for 0000-01-01 to 9999-12-31.
e) Decimals:
In Hive DECIMAL type is similar to Big Decimal format of java. This represents immutable arbitrary precision. The syntax and example are below:

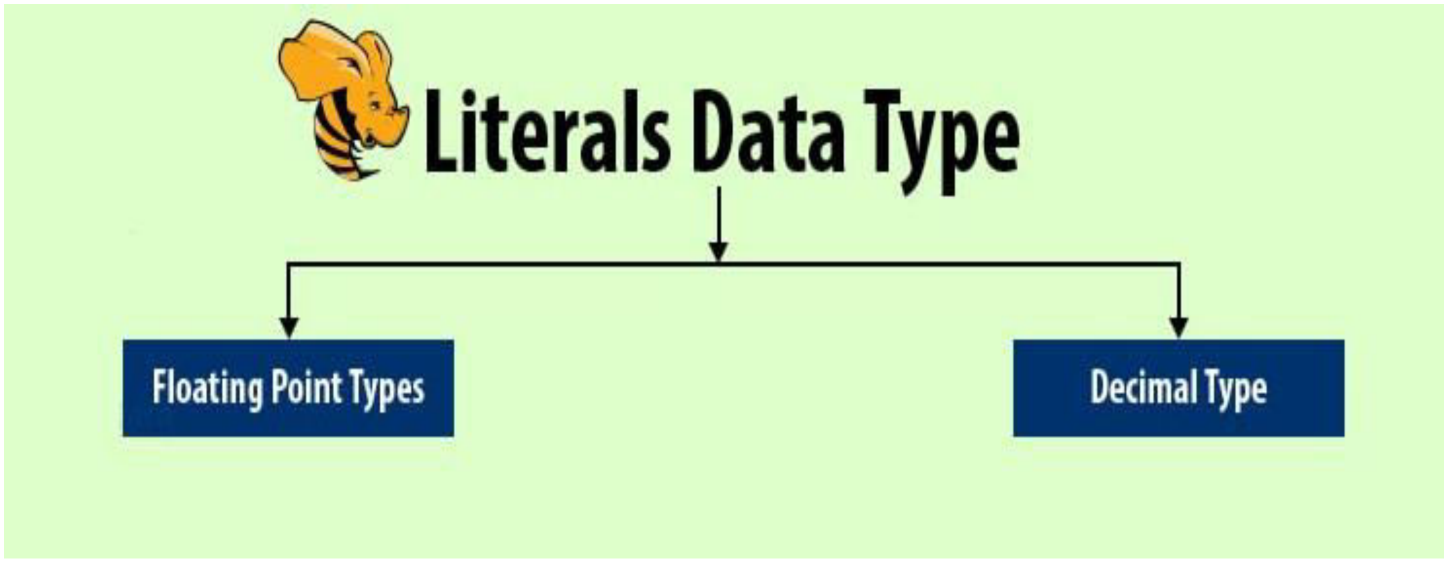
Apache Hive 0.11 and 0.12 has the precision of the DECIMAL type fixed. And it's limited to 38 digits.
Apache Hive 0.13 users can specify scale and precision when creating tables with the DECIMAL data type using DECIMAL (precision, scale) syntax. If the scale is not specified, then it defaults to 0 (no fractional digits). If no precision is specified, then it defaults to 10.

f) Union Types:
Union Hive data types are the collection of heterogeneous data types. By using **create union** we can create an instance. The syntax and example are below:

```
CREATE TABLE UNION_TEST
(foo UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>);
SELECT foo FROM UNION_TEST;
{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}
```

D) Literals Data Types in Hive:
In Hive data types following literals are used

- Floating Point Types
- Decimal Type

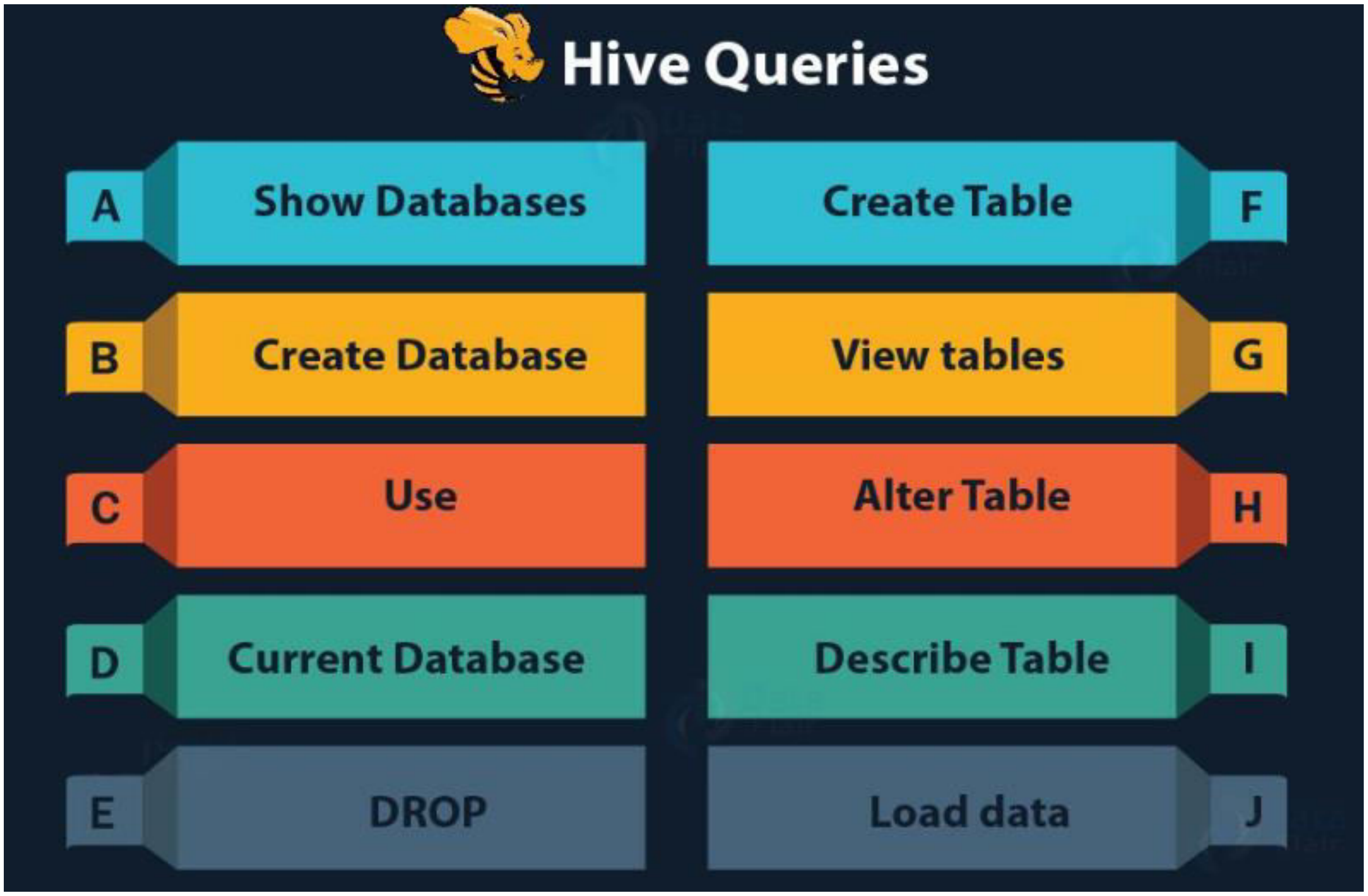


a) Floating Point Types:
These are nothing but numbers with decimal points. This type of data is composed of the DOUBLE data types in Hive.

b) Decimal Type:
This type is nothing but floating point value with the higher range than the DOUBLE data type. The decimal type range is approximate -10-308 to 10308.

E) Null Value Data Types in Hive:
In this Hive Data Types, missing values are represented by the special value **NULL**.

Hive Queries:
Below are some basic Hive queries which you will need while using Apache Hive.



a) Show Databases:
This Hive query gives a list of databases which are present in your Hive. If you had newly installed Hive and had not created any database, then by default a database named **“default”** is present there and would be shown up after executing above query.

b) Create Database:
This will create a new database named “test”. And you can check this database by writing “show databases;” query.

c) Describe Database: It show the database name, directory & user.

c) Use: USE query is used to use the database created by you.

d) Current Database:
It is used to know the name of the database in which you are currently working.

e) DROP: DROP query is used to delete a database

f) Create Table: This command is used to create a new table.

g) View Table: It will list you all the tables created by you in the current directory.

h) Alter Table: It is used to change attributes inside a table.

Note: Data Types of all columns in query should be same, else it won't work.

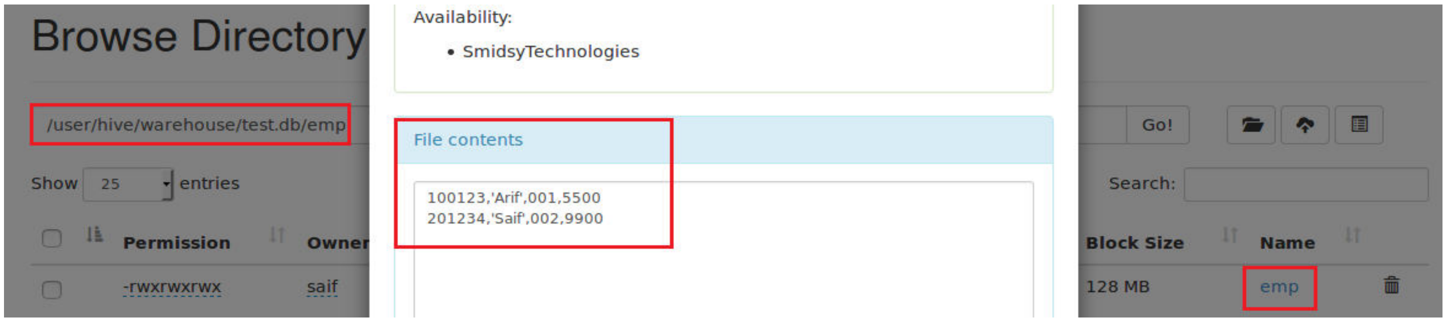
i) Describe Table:

j) TRUNCATE TABLE:

k) Load Data:

This command loads the data from your Local/HDFS file path to the selected table created by you in Hive.

TRUNCATE & LOAD:



```
100123,'Arif',001,5500
201234,'Saif',002,9900
```

Put data into HFS from LFS:

Data from HFS:
If you move data from HFS to HIVE table, the file from HFS is removed.

l) INSERT Command:
It is used to insert records in a table.

m) MULTI-INSERT Command:
It is used to insert records into multiple tables using a single command.

Tables in HIVE:
1) Managed/Internal Table
2) External Table

1) Managed/Internal table

- Managed table is also called as Internal table. This is the default table in Hive. When we create a table in Hive without specifying it as external, by default we will get a Managed table.
- If we create a table as a managed table, the table will be created in a specific location in HDFS.
- By default, the table data will be created in /usr/hive/warehouse directory of HDFS.
- If we delete a Managed table, both the table data and metadata for that table will be deleted from the HDFS.

Loading data from LFS to HIVE Table:

Now let's drop the table & see:

2) EXTERNAL TABLE

- External table is created for external use as when the data is used outside Hive.
- Whenever we want to delete the table's metadata and we want to keep the table's data as it is, we use External table.
- External table only deletes the schema of the table.

Load data from LFS to HIVE:

Now let's drop the table & see:

You can see that the contents of the table are still present in the HDFS location.

If we create an External table, after deleting the table only the metadata related to table is deleted but not the contents of the table.

The above approach will work only if your data is in **/user/hive/warehouse** directory. But if your data is in another location, if you delete the table the data will also get deleted. In case of LFS **not HFS**.

Difference between Internal & External tables:

For External Tables:
External table stores files on the HDFS server but tables are not linked to the source file completely.
If you delete an external table the file still remains on the HDFS server.
As an example if you create an external table called “table_test” in HIVE using HIVE-QL and link the table to file “file”, then deleting “table_test” from HIVE will not delete “file” from HDFS.
External table files are accessible to anyone who has access to HDFS file structure and therefore security needs to be managed at the HDFS file/folder level.
Meta data is maintained on master node, and deleting an external table from HIVE only deletes the metadata not the data/file.

For Internal Tables:
Stored in a directory based on settings in hive.metastore.warehouse.dir, by default internal tables are stored in the following directory “/user/hive/warehouse” you can change it by updating the location in the config file.
Deleting the table deletes the metadata and data from master-node and HDFS respectively.
Internal table file security is controlled solely via HIVE. Security needs to be managed within HIVE, probably at the schema level (depends on organization).

Functions:

1) unix_timestamp: We will get current Unix timestamp in seconds.

2) from_unixtime: It converts seconds into datetime.

3) year/quarter/month/day/hour/minute/second: It will select year, quarter, month, day, hour, minute & second from a given datetime.

4) to_date: It will display date from a given datetime/timestamp.

5) weekofyear: It will display week of the year.

6) datediff: It will display the date difference.

7) date_add: It adds/subtract days to a given date.

8) date_sub: It subtract days from a given date.

9) current_date: It displays current date.

10) last_day: It displays the last day from the given date.

11) ceil: This will return the next integral value.

12) floor: This will return the same integral value.

13) round: This will increment to next integral value if precision is 5 or more than 5 else it will give the same integral value.

14) concat: It concatenates two string values specified by the delimiter.

15) length: It counts the length including spaces.

16) lower: It converts the string to lower case.

17) upper: It converts the string to upper case.

18) lpad(string str, int length, string pad): It pad to a length of length(integer value) to left.

19) rpad(string str, int length, string pad): It pad to a length of length(integer value) to right.

20) trim: It trims spaces from both sides.

21) ltrim: It trims spaces from left side.

22) rtrim: It trims spaces from right side.

23) reverse: It reverses the given string.

24) split(STRING str, STRING pat): It will split the string based on the given pattern.

25) substr(string, start pos, len): This will return the characters from the given specified string.

26) instr(string, pattern): This will return the position of the character specified.

27) nvl: It replaces null values with a specified string.

28) coalesce: It returns first not null value from a string. If all the values are null then null is displayed.

29) if/case: It gives the flexibility to use if/else statement.

30) rank: In this function sequence is skipped.

31) dense_rank: In this function sequence is not skipped.

32) row_number: It gives an incremented value in a sequence.

33) Explode: Explodes an array to multiple rows.

34) Lateral View: The lateral views are used along with EXPLODE or INLINE functions. Both functions work on the complex data types such as array. Explode function in the lateral view can contain embedded functions such as *map*, *array*, *struct*, *stack*, etc.

Hive Partitions:

Tables, Partitions, and Buckets are the parts of Hive data modeling.

What is Partitions?
Hive Partitions is a way to organizes tables into partitions by dividing tables into different parts based on partition keys.
Table partitioning means dividing table data into some parts based on the values of particular columns like date or country, which segregates the input records into different files/directories based on date or country.

Partitioning can be done based on more than one column which will impose multi-dimensional structure on directory storage. For e.g. in addition to partitioning records by date column, we can also sub-divide the single day records into country wise separate files by including country column into partitioning.

Advantages:
1) Partitioning is used for distributing execution load horizontally.
2) As the data is stored in slices/parts, query response time is faster to process the small parts of data instead of searching in the entire data set.
3) For e.g. in a larger table where the table is partitioned by country, then selecting users of country 'IN' will just scan one directory 'country=IN' instead of all directories.

Disadvantages:
1) Having too many partitions in table create large number of files/directories in HDFS, which is an overhead to NameNode since it has to keep all Metadata for the file system in memory only.
2) Partitions may optimized some queries based on Where clause but may be less responsive for other queries based on grouping etc.

Create Table Statement: Refer Training Data provided.

Note: We didn't include state column in table definition but included in partition definition. If we include in table definition then it will throw an error.

External Partition Table:
We can create external partition table as well just by using the EXTERNAL keyword in the CREATE statement. For creation of external partitioned tables, we do not need to mention LOCATION clause as we will mention locations of each partitions separately while inserting data into table.

Inserting Data into Partitioned Tables:
Data insertion into partitioned tables can be done in two ways.
1) Static Partition
2) Dynamic Partition

Static Partitioning in Hive:
In this mode, input data should contain the columns listed in table definition but not the columns defined in partition by clause.

If our input column layout is according to the expected layout and we already have separate input files for each partitioned columns then these files can be easily loaded into partitioned tables.

Loading data into managed table from local fs:

This will create separate directory under the default warehouse directory in HDFS. Similarly, we have to add other partitions, which will create corresponding directories in HDFS. Or else we can load the entire directory in HIVE table with single command and can add partitions for each with ALTER command.

Loading partition table from another table:
We can load or add partitions with query results from another table as shown below:
Insert overwrite table <Partition_Table_Name>
Partition (state='MH')

Select <Column_Names> from <Non_Partition_Table_Name>
Where state='MH';

Overwriting Existing Partition:
We can overwrite existing partition with help of Overwrite into table <Partition_Table_Name> commands

Loading data into External Partitioned Table from HDFS:
If data is present in HDFS and should be made accessible to HIVE, we will just mention the location of HDFS files for each partition

If our files are on LFS then move to HFS, and then we can add partition for each file in that directory with command below:

ALTER TABLE <Partition_Table_Name> ADD IF NOT EXISTS
ADD PARTITION (<Column_Name>='<Value>') Location 'Path_of_File_HFS'
ADD PARTITION (<Column_Name>='<Value>') Location 'Path_of_File_HFS';

Dynamic Partition in HIVE:
Instead of loading each partition with single SQL statements as above which will result in writing lots of SQL's statements for huge no of partitions. Hive supports dynamic partitioning with which we can add any number of partitions with single SQL execution.

Hive will automatically splits our data into separate partitions files based on the values of partition keys present in input files.

It gives the advantages of easy coding and no need of manual identifications of partitions. And, for dynamic partition loading we will not provide the partition keys.

By default, dynamic partitioning is disabled in HIVE to prevent accidental partition creations. To use dynamic partitions we need to set below properties either in HIVE SHELL or hive-site.xml file
We can set these through hive shell with below commands:

Now, run your dynamic partition query:

Note: We can also mix dynamic & static partitions by specifying it as Partition (state='MH', country). But static partition keys must come before the dynamic partition keys.
Note that, dynamic partition strict mode is different from hive.mapred.mode=strict because if this property is set to strict, then we cannot user certain queries on partitioned tables. So in mapreduce strict mode (hive.mapred.mode=strict) some risky queries are not allowed to run.

1) Cartesian product
2) No partition being picked up for a query
3) Comparing bigints with String and bigints with double
4) Order by without limit

According to point 2 & 4, we cannot use SELECT statements without at least one partition key filter (e.g. Where state='MH') or ORDER BY clause without limit condition on partitioned tables.

show partitions <Partition_Table_Name>;

If we have lots of partitions and want to see partitions for some particular keys, we can further restrict the command with an Optional partition clause with specific values.

show partitions <Partition_Table_Name> partition (<Column_Name>='<Value>');

Describe partitions:
As we know how to see the description of partition tables, we can see the entire partition description with below command.

DESCRIBE FORMATTED <Partition_Table_Name> PARTITION (<Column_Name>='<Value>');

Alter Partitions:
We can add/change/drop partitions with command.

Drop Partitions: We can drop partitions of a table with DROP IS EXISTS PARTITION.

Pros & Cons of Partitions:

Pros:
1) It's used for distributing execution load horizontally.
2) Query response is faster as query is processed on a small dataset instead of entire dataset.
3) If we selected records for US, records would be fetched from directory 'Country=US' from all directories/partitions.

Cons:
1) Having large number of partitions create number of files/ directories in HDFS, which creates overhead for NameNode as it maintains metadata.
2) It may optimize certain queries based on where clause, but may cause slow response for queries based on grouping clause.

No_Drop Table/Partition:

create table emp_drop as select * from partition_emp_dynamic;
alter table emp_drop enable no_drop;
alter table partition_emp_dynamic partition (country='IN') enable no_drop;
alter table partition_emp_dynamic drop if exists partition (country='IN');

Hive Buckets:

At times, even after partitioning on a particular field or fields, the partitioned file size doesn't match with the actual expectation and remains huge and we want to manage the partition results into different parts. To overcome this problem of partitioning, Hive provides Bucketing concept, which allows user to divide table data sets into more manageable parts.

Hive partition divides table into number of partitions and these partitions can be further subdivided into more manageable parts known as Buckets or Clusters. The Bucketing concept is based on Hash function, which depends on the type of the bucketing column. Records which are bucketed by the same column will always be saved in the same bucket.

Here, CLUSTERED BY clause is used to divide the table into buckets.
In Hive Partition, each partition will be created as directory. But in Hive Buckets, each bucket will be created as file. Bucketing can also be done even without partitioning on Hive tables.

Advantages of Bucketing:
Bucketing can sometimes be more efficient when used alone.
Bucket is physically a fine & all data files are equal sized parts, map-side joins will be faster on the bucketed tables.

Set the Below Properties in Hive Command Line Before Proceeding Further for Bucketing Scripts:
To populate the bucketed table, we have to set hive.enforce.bucketing property to 'true', so that the Hive knows to create the number of buckets declared in the table definition.

The property hive.enforce.bucketing = true is similar to hive.exec.dynamic.partition = true, in Hive partitioning. By setting this property, we will enable dynamic bucketing while loading data into the Hive table.
The above hive.enforce.bucketing = true property sets the number of reduce tasks to be equal to the number of buckets mentioned in the table definition (Which is '4' in our case) and automatically selects the clustered by column from table definition.

Creating Bucket Table:

From the above image, we can see that we have created a new bucket table with name 'Bucket_Table', which is partitioned by 'city' and clustered by 'street' field with the bucket size of '4'.

Partitioned on city:

Bucketed into 4: We can observe from the above image, four Buckets have been created in the above Hive warehouse path for every city name.

Hence, from the above execution steps, we can observe that we have created a structure to decompose the data into more manageable parts or equal parts using Bucketing technique in Hive.

Table Samplings:

Advantages of Bucketing:

1) It provides faster query response like portioning.
2) In bucketing due to equal volumes of data in each partition, joins at Map side will be quicker.

HIVE Joins:

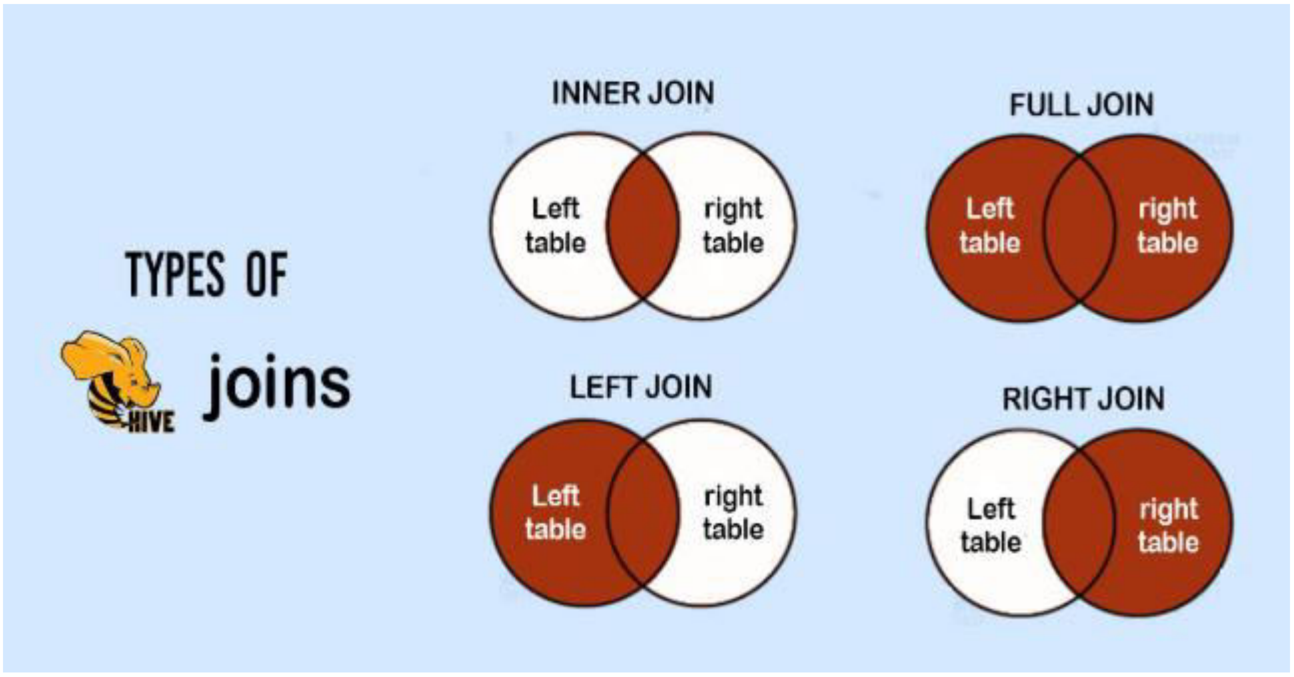
Hive JOIN clause is used to combine records from two or more tables in the database we use JOIN clause. However, it is more or less similar to SQL JOIN. Also, we use it to combine rows from multiple tables. Moreover, there are some points we need to observe.

- In Joins, only Equality joins are allowed.
- However, in the same query more than two tables can be joined.
- Basically, to offer more control over ON Clause for which there is no match LEFT, RIGHT, FULL OUTER joins exist in order.
- Also, note that Hive Joins are not Commutative
- Whether they are LEFT or RIGHT joins in Hive, even then Joins are left-associative.

Type of Joins in Hive:

Basically, there are 4 types of Hive Joins

- Inner join in Hive
- Left Outer Join in Hive
- Right Outer Join in Hive
- Full Outer Join in Hive



Put your files from LFS to HFS:

Creation of customers table & pointing to HDFS location:

Creation of orders table & pointing to HDFS location:

1) Inner Join:
It returns the matching rows from both the tables.

2) Left Outer Join:
This type of join returns all rows from the left table even if there is no matching row in the right table. Table returns all rows from the left table and matching rows from right table. Unmatched right table records will be NULL.

3) Right Outer Join:
It returns all rows from the right table even if there is no matching row in the left table. Table returns all rows from the right table and matching rows from left table. Unmatched left table records will be NULL.

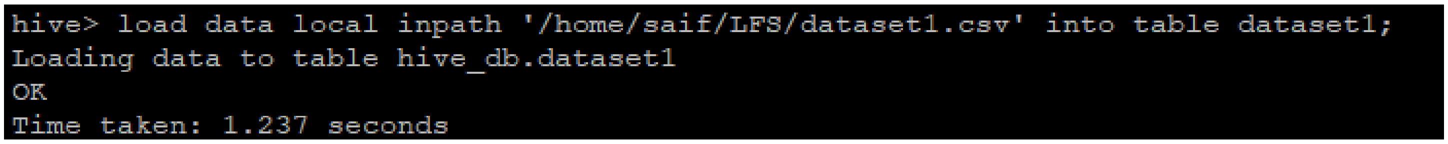
4) Full Outer Join
It returns all rows from the both tables that fulfill the JOIN condition. The unmatched rows from both tables will be returned as a NULL.

Creation of order_items table in Hive with HDFS Location & not Hive's default location:

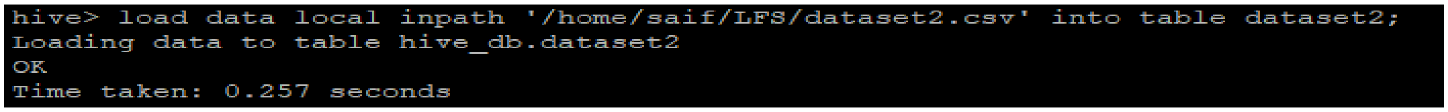
Map Side Joins:

- Map side join is a process where joins between two tables are performed in the Map phase without the involvement of Reduce phase.
- Map-side Joins allows a table to get loaded into memory ensuring a very fast join operation, performed entirely within a mapper and that too without having to use both map and reduce phases.

Load data in both tables:



```
hive> load data local inpath '/home/saif/LFS/dataset1.csv' into table dataset1;
Loading data to table hive_db.dataset1
OK
Time taken: 1.237 seconds
```



```
hive> load data local inpath '/home/saif/LFS/dataset2.csv' into table dataset2;
Loading data to table hive_db.dataset2
OK
Time taken: 0.257 seconds
```

1) Map Join:

a) By specifying the keyword, /*+ MAPJOIN (b) */ in the join statement.

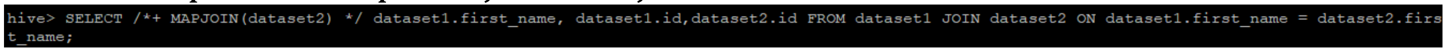
b) By setting the following property to true.

hive.auto.convert.join = true

For performing Map-side joins, there should be two files, one is of larger size and the other is of smaller size. You can set the small file size by using the following property:

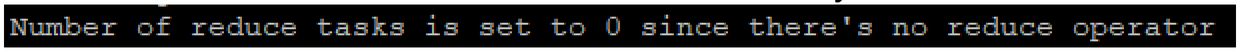
hive.mapjoin.smalltable.filesize = (default it will be 25MB)

Now, let us perform Map-side joins and join the two datasets based on their IDs.



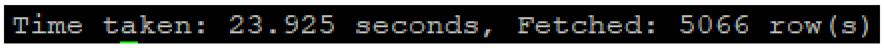
```
hive> SELECT /*+ MAPJOIN(dataset2) */ dataset1.first_name, dataset1.id,dataset2.id FROM dataset1 JOIN dataset2 ON dataset1.first_name = dataset2.firs
t_name;
```

The number of reducers will be set to 0 automatically.



```
Number of reduce tasks is set to 0 since there's no reduce operator
```

PFB Time Taken



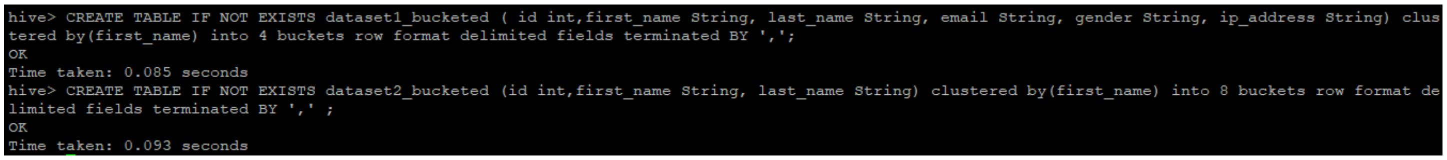
```
Time taken: 23.925 seconds, Fetched: 5066 row(s)
```

2) Bucket-Map Join:

The constraint for performing Bucket-Map join is:

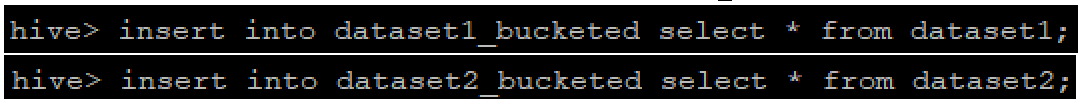
If tables being joined are bucketed on the join columns, and the number of buckets in one table is a multiple of the number of buckets in the other table, the buckets can be joined with each other.

To perform bucketing, we need to have bucketed tables. Let's create them.



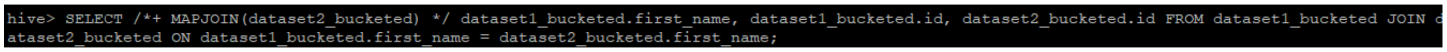
```
hive> CREATE TABLE IF NOT EXISTS dataset1_bucketed ( id int,first_name String, last_name String, email String, gender String, ip_address String) clus
tered by(first_name) into 4 buckets row format delimited fields terminated BY ',';
OK
Time taken: 0.085 seconds
hive> CREATE TABLE IF NOT EXISTS dataset2_bucketed (id int,first_name String, last_name String) clustered by(first_name) into 8 buckets row format de
limited fields terminated BY ',' ;
OK
Time taken: 0.093 seconds
```

Now we will insert the data into the dataset1_bucketed table.



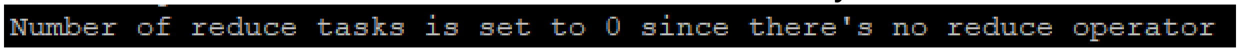
```
hive> insert into dataset1_bucketed select * from dataset1;
hive> insert into dataset2_bucketed select * from dataset2;
```

Here, we have two tables that are bucketed. We can now perform Bucket-map join between these two datasets.
Here, for the first table we have created 4 buckets and for the second table we have created 8 buckets on the same column. Now, we can perform Bucket-map join on these two tables.
For performing Bucket-Map join, we need to set this property in the Hive shell.
set hive.optimize.bucketmapjoin = true



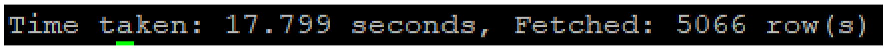
```
hive> SELECT /*+ MAPJOIN(dataset2_bucketed) */ dataset1_bucketed.first_name, dataset1_bucketed.id, dataset2_bucketed.id FROM dataset1_bucketed JOIN d
ataset2_bucketed ON dataset1_bucketed.first_name = dataset2_bucketed.first_name;
```

The number of reducers will be set to 0 automatically.



```
Number of reduce tasks is set to 0 since there's no reduce operator
```

PFB Time Taken



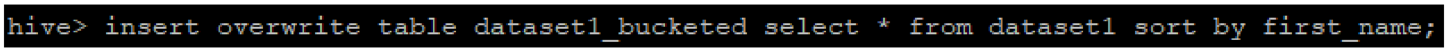
```
Time taken: 17.799 seconds, Fetched: 5066 row(s)
```

3) Sort Merge Bucket(SMB) Map Join:
If the tables being joined are sorted and bucketed on the join columns and have the same number of buckets, a sort-merge join can be performed. The corresponding buckets are joined with each other at the mapper.

Here we have 4 buckets for dataset1 and 8 buckets for dataset2. Now, we will create another table with 4 buckets for dataset2.

For performing the SMB-Map join, we need to set the following properties:
set hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
set hive.optimize.bucketmapjoin = true;
set hive.optimize.bucketmapjoin.sortedmerge = true;

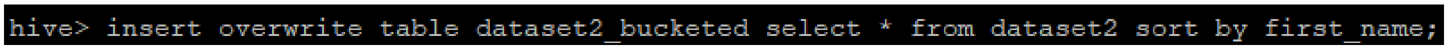
To perform this join, we need to have the data in the bucketed tables sorted by the join column. Now, we will re-insert the data into the bucketed tables by using sorting the records.



```
hive> insert overwrite table dataset1_bucketed select * from dataset1 sort by first_name;
```

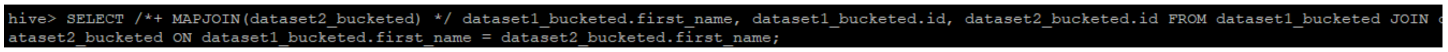
The above command will overwrite the data in the old table and insert the data as per the query and then the data in the dataset1_bucketed table is sorted by first_name.

We will now overwrite the data into the dataset2_bucketed table, using the following command:



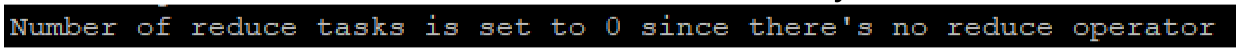
```
hive> insert overwrite table dataset2_bucketed select * from dataset2 sort by first_name;
```

Now, let us perform the SMB-Map join on the two tables with 4 buckets in one table and 8 buckets in one table.



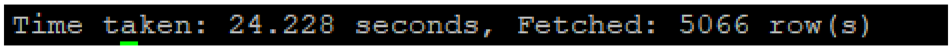
```
hive> SELECT /*+ MAPJOIN(dataset2_bucketed) */ dataset1_bucketed.first_name, dataset1_bucketed.id, dataset2_bucketed.id FROM dataset1_bucketed JOIN d
ataset2_bucketed ON dataset1_bucketed.first_name = dataset2_bucketed.first_name;
```

The number of reducers will be set to 0 automatically.



```
Number of reduce tasks is set to 0 since there's no reduce operator
```

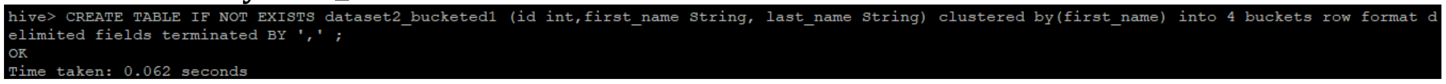
PFB Time Taken



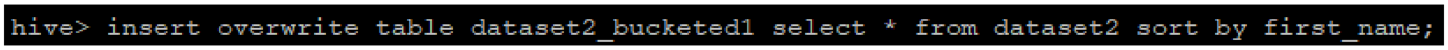
```
Time taken: 24.228 seconds, Fetched: 5066 row(s)
```

To perform SMB-Map join, we need to have the same number of buckets in both the tables with the bucketed column sorted.

Now, we will create another table for dataset2 having 4 buckets and will insert the data that is sorted by first_name.

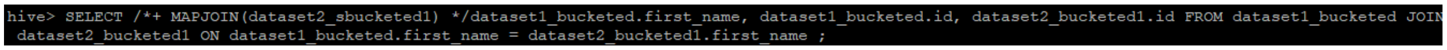


```
hive> CREATE TABLE IF NOT EXISTS dataset2_bucketed1 (id int,first_name String, last_name String) clustered by(first_name) into 4 buckets row format d
elimited fields terminated BY ',' ;
OK
Time taken: 0.062 seconds
```



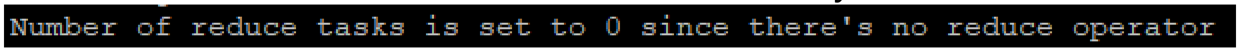
```
hive> insert overwrite table dataset2_bucketed1 select * from dataset2 sort by first_name;
```

Now, we have two tables with 4 buckets and the joined column sorted. Let us perform the join query again.



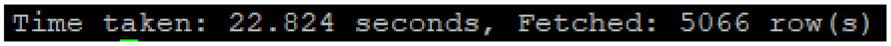
```
hive> SELECT /*+ MAPJOIN(dataset2_sbucketed1) */dataset1_bucketed.first_name, dataset1_bucketed.id, dataset2_bucketed1.id FROM dataset1_bucketed JOIN
 dataset2_bucketed1 ON dataset1_bucketed.first_name = dataset2_bucketed1.first_name ;
```

The number of reducers will be set to 0 automatically.



```
Number of reduce tasks is set to 0 since there's no reduce operator
```

PFB Time Taken



```
Time taken: 22.824 seconds, Fetched: 5066 row(s)
```

As the dataset sizes are too small, this analysis results may vary based on the dataset size.

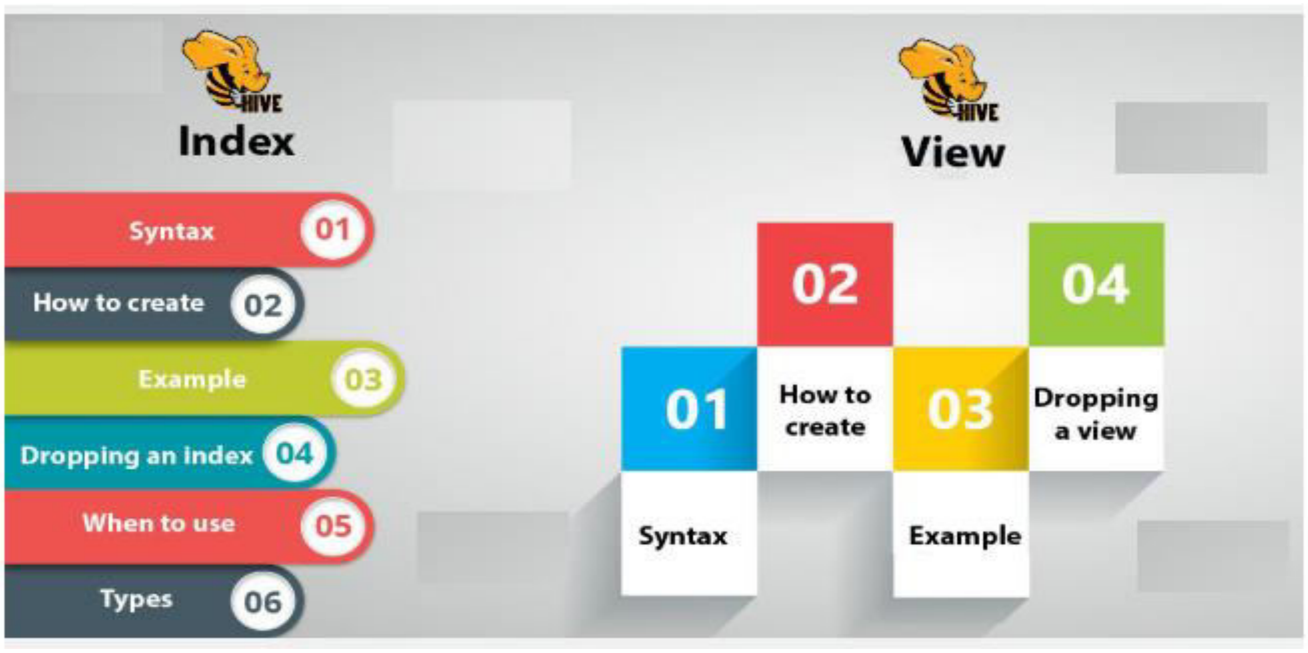
This analysis is performed under ideal conditions for all the joins. So in this case, in order to perform a Map join, there are no constraints.

But for performing a Bucket-Map join, we need to have the tables bucketed on the join column and the number of buckets can be multiples of each other.

To perform the Sort-Merge-Bucket Map join, we need to have two tables with the same number of buckets on the join column and the records are to be sorted on the join column.

Hive Views:

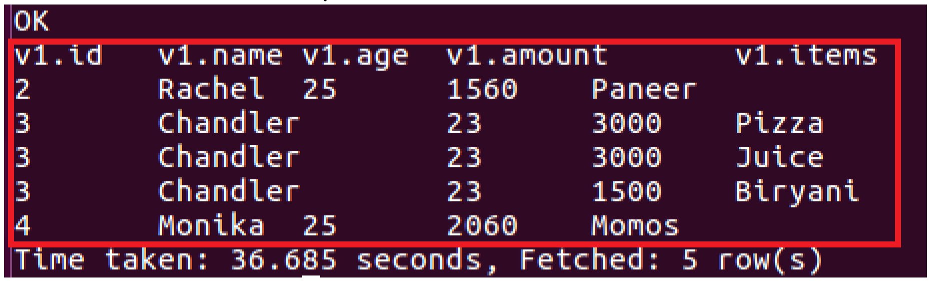
Here we will cover how to create Hive Index and hive Views, manage views and Indexing of hive, hive index types, hive index performance, and hive view performance.



You can use **Hive create view** to create a virtual table based on the result-set of a complex SQL statement that may have multiple table joins. The CREATE VIEW statement lets you create a shorthand abbreviation for a more complex and complicated query.

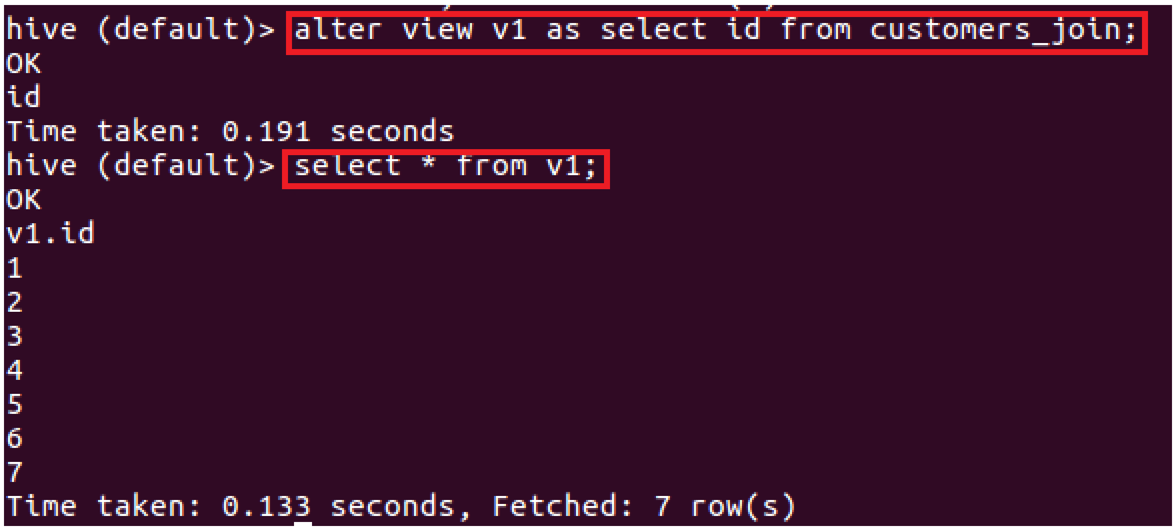
Hive Create View Syntax:
CREATE VIEW [IF NOT EXISTS] view_name [(column_list)]
AS select_statement;

SELECT * FROM V1;



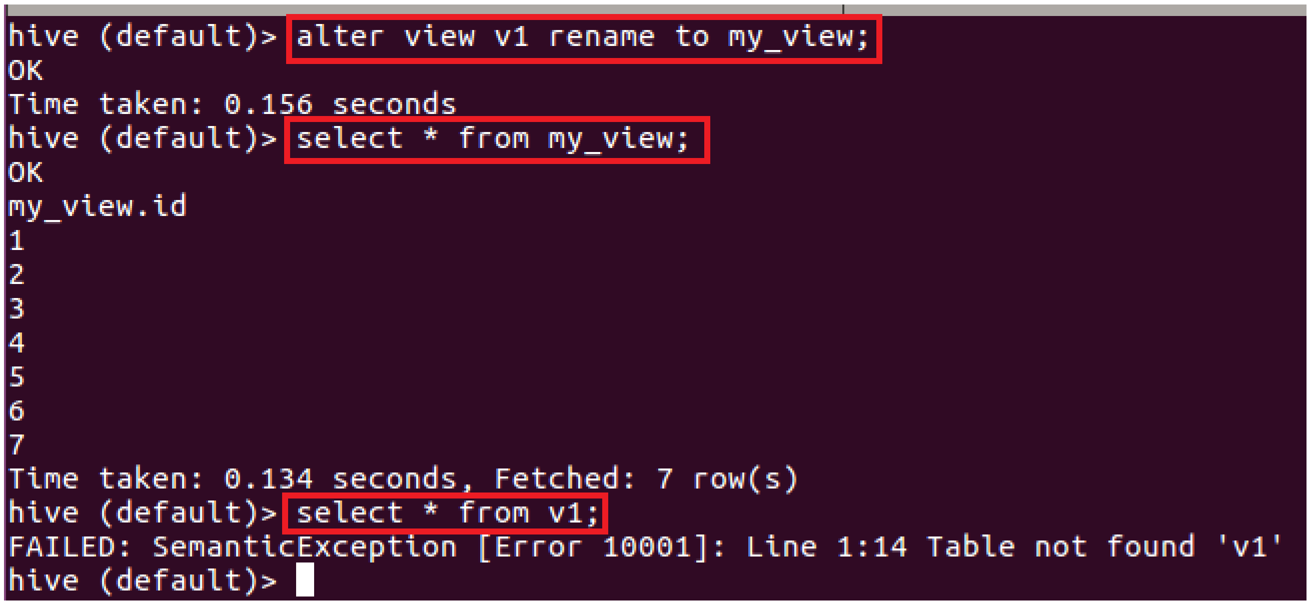
```
OK
v1.id   v1.name v1.age  v1.amount       v1.items
2       Rachel  25      1560    Paneer
3       Chandler        23      3000    Pizza
3       Chandler        23      3000    Juice
3       Chandler        23      1500    Biryani
4       Monika  25      2060    Momos
Time taken: 36.685 seconds, Fetched: 5 row(s)
```

Hive ALTER VIEW:
With use of **Hive ALTER VIEW** statement, you can change the query in the AS clause or rename the view to other name as per your requirements.



```
hive (default)> alter view v1 as select id from customers_join;
OK
id
Time taken: 0.191 seconds
hive (default)> select * from v1;
OK
v1.id
1
2
3
4
5
6
7
Time taken: 0.133 seconds, Fetched: 7 row(s)
```

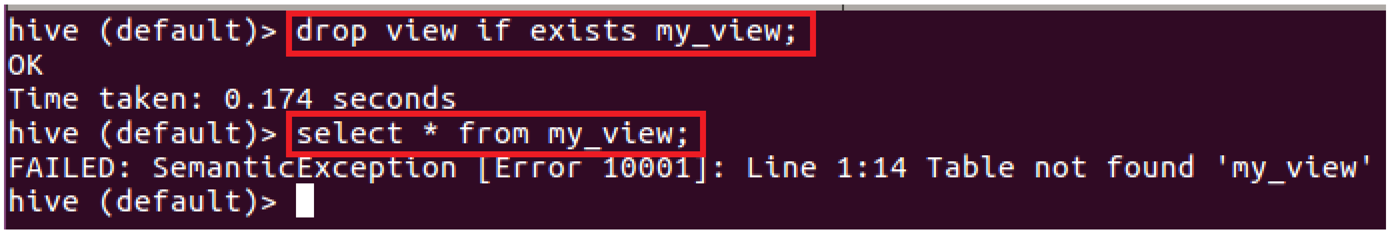
Rename:



```
hive (default)> alter view v1 rename to my_view;
OK
Time taken: 0.156 seconds
hive (default)> select * from my_view;
OK
my_view.id
1
2
3
4
5
6
7
Time taken: 0.134 seconds, Fetched: 7 row(s)
hive (default)> select * from v1;
FAILED: SemanticException [Error 10001]: Line 1:14 Table not found 'v1'
hive (default)>
```

Hive DROP VIEW:
You can use **Hive DROP VIEW** statement to remove view from Hive metastore. This statement removes the specified view, which was originally created by the CREATE VIEW statement.

Hive DROP VIEW Syntax:
DROP VIEW [IF EXISTS] [db_name.]view_name;



```
hive (default)> drop view if exists my_view;
OK
Time taken: 0.174 seconds
hive (default)> select * from my_view;
FAILED: SemanticException [Error 10001]: Line 1:14 Table not found 'my_view'
hive (default)>
```

HIVE Index:

An Index act as reference to records.
Used to speed up searching the data. Will search for only portion of data and not whole dataset.
Partition is done at HDFS level & Indexing is done at table level.
Since Hive works with large dataset indexing can be beneficial.

The main goal of **creating INDEX on Hive table** is to improve the data retrieval speed and optimize query performance. For example, let us say you are executing Hive query with filter condition WHERE col1 = 100, without index hive will load entire table or partition to process records and with index on col1 would load part of HDFS file to process records.

Hive CREATE INDEX Syntax:
You can create INDEX on particular column of the table by using **CREATE INDEX** statement. Below is the syntax:

```
CREATE INDEX index_name
 ON TABLE base_table_name (col_name, ...)
 AS index_type
 [WITH DEFERRED REBUILD]
 [IDXPROPERTIES (property_name=property_value, ...)]
 [IN TABLE index_table_name]
 [ [ ROW FORMAT ...] STORED AS ...
 | STORED BY ... ]
 [LOCATION hdfs_path]
 [TBLPROPERTIES (...)]
 [COMMENT "index comment"];
```

CREATE INDEX:

Hive ALTER INDEX:
ALTER INDEX ... REBUILD builds an index that was created using the **WITH DEFERRED REBUILD** clause, or rebuilds a previously built index on the table. You should provide PARTITION details if the table is partitioned.

Hive ALTER INDEX Syntax:
ALTER INDEX index_name ON table_name [PARTITION partition_spec] REBUILD;

ALTER INDEX my_index ON customers_join REBUILD;

Hive DROP INDEX:
DROP INDEX statement drops the index and delete index table.

Hive DROP INDEX Syntax:
DROP INDEX [IF EXISTS] index_name ON table_name;

When to use Hive Indexing?
Under the following circumstances, we can use Indexing in Hive:

- While the dataset is very large.
- Whenever the query execution is more amount of time than you expected.
- While we need a speedy query execution.
- While we build a data model.

Hive Index is maintained in a separate table. Hence, it won't affect the data inside the table, which contains the data. There is one more advantage of it. That is for indexing in Hive is that index can also be partitioned depending on the size of the data we have.

User Defined functions (UDF):

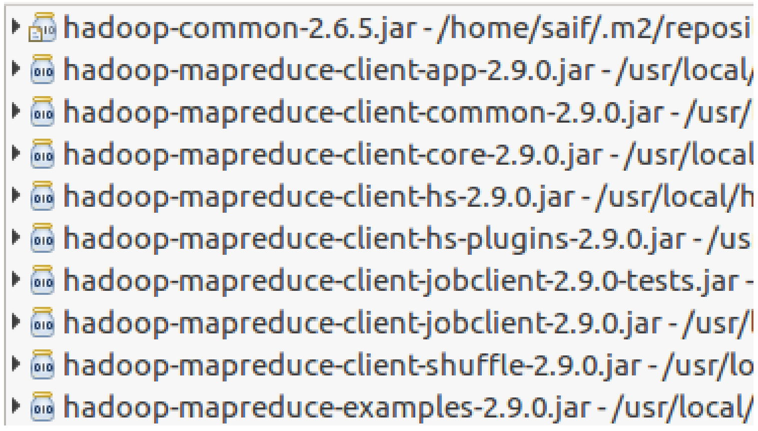
We can create our own functions when built in functions does not serve the purpose. Functions are written mostly in java. There are some rules to be adhered to while writing UDF's.

Steps to create a UDF:
1) Create a JAVA program in any platform.
2) Save or Convert the program into jar file.
3) Add that jar file into HIVE.
4) Create function of the jar file added.
5) Use those functions in HIVE query.

Steps to create UDF in Eclipse:
1) Create a new java project & give project name.

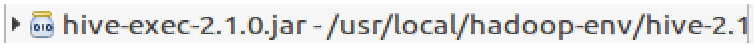
2) Right click on project, click Build Path, configure Build Path, Go to Libraries & Add External JAR's for basic mapreduce program to run on hadoop.

Path: /usr/local/hadoop-env/hadoop-2.9.0/share/hadoop/mapreduce



```
hadoop-common-2.6.5.jar - /home/saif/.m2/reposi
hadoop-mapreduce-client-app-2.9.0.jar - /usr/local/
hadoop-mapreduce-client-common-2.9.0.jar - /usr/
hadoop-mapreduce-client-core-2.9.0.jar - /usr/local
hadoop-mapreduce-client-hs-2.9.0.jar - /usr/local/h
hadoop-mapreduce-client-hs-plugins-2.9.0.jar - /us
hadoop-mapreduce-client-jobclient-2.9.0-tests.jar -
hadoop-mapreduce-client-jobclient-2.9.0.jar - /usr/l
hadoop-mapreduce-client-shuffle-2.9.0.jar - /usr/lo
hadoop-mapreduce-examples-2.9.0.jar - /usr/local/
```

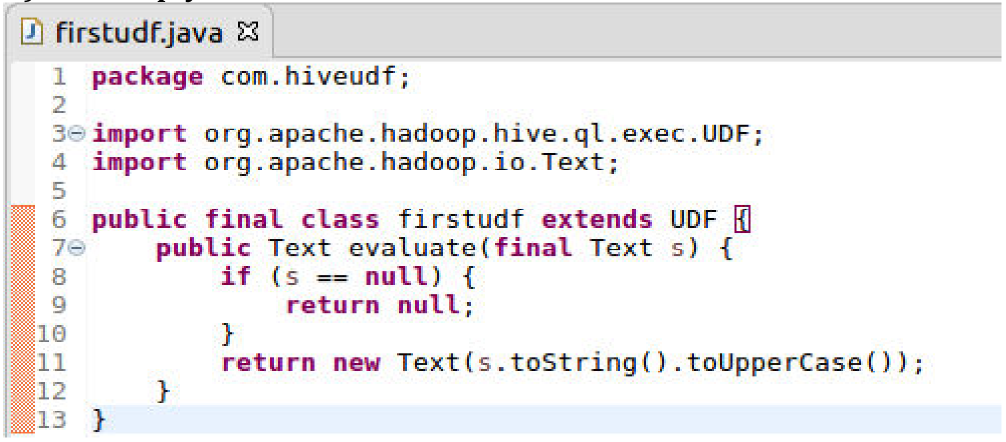
3) Then add Hive jar file which will have all definition which inherits all classes & functions
Path: /usr/local/hadoop-env/hive-2.1.0-bin/lib



```
hive-exec-2.1.0.jar - /usr/local/hadoop-env/hive-2.1
```

4) Right click on src & create a new package and create a class within that.

5) Develop your code:



firstudf.java

```java
package com.hiveudf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public final class firstudf extends UDF {
    public Text evaluate(final Text s) {
        if (s == null) {
            return null;
        }
        return new Text(s.toString().toUpperCase());
    }
}
```

6) Save it & create a jar file.

7) Add jar file to your hive by below command:

8) Create a function of jar file by specifying package.class name:

9) Use the function in your Hive query as below:

ACID Transactional features in HIVE:

Points to Consider:
1) Only ORC storage format is supported presently.
2) Table must have CLUSTERED BY column
3) Table properties must have: “transactional”=”true”
4) External tables cannot be transactional.
5) Transactional tables cannot be read by non-ACID session.
6) Table cannot be loaded using “LOAD DATA…” command.
7) Once table is created as transactional, it cannot be converted to non-ACID afterwards.

Following properties must be set at Client Side to use transactional tables:

```
set hive.support.concurrency=true;
set hive.enforece.bucketing=true;
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.compactor.initiator.on=true;
set hive.compactor.worker.threads=1;
set hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
```

Create an ORC table:

As of Hive 0.14, if a table has an OutputFormat that implements AcidOutputFormat and the system is configured to use a transaction manager that implements ACID, then INSERT OVERWRITE will be disabled for that table. This is to avoid users unintentionally overwriting transaction history. The same functionality can be achieved by using TRUNCATE TABLE (for non-partitioned tables) or DROP PARTITION followed by INSERT INTO. OR Don’t create an External table.

Write Data into your ORC table from external table:

View your ORC Table:

Now you can Insert, Update & Delete on your ORC transactional table:

Insert:

Update:

Delete:

Errors:
1) [Error 10302]: Updating values of bucketing columns is not supported.
Since ACID table must be bucketed to enable transactional property, you cannot run UPDATE command to set BUCKETED column. In the table DDL, you can see "CLUSTERED BY (pres_bs) INTO 4 BUCKETS". Now we cannot run UPDATE command with set pres_bs='any_value'.
Solution: Change bucketed column if you really have to run UPDATE on that column.

2) [Error 10292]: Updating values of partition columns is not supported.
Similar to previous constraint, if you have created partitioned table then you cannot run UPDATE command to set PARTITION column to new value. If it is non-transactional, then anyways you would be re-writing entire partition. So won't get this error in non-ACID tables.
Solution: Change partitioned column if you really have to run UPDATE on that column.

3) [Error 10295]: INSERT OVERWRITE not allowed on table with OutputFormat that implements AcidOutputFormat while transaction manager that supports ACID is in use.
You cannot execute INSERT OVERWRITE command if you are using hive ACID tables.
Solution: Run Truncate & INSERT in place of INSERT OVERWRITE query.
4) [Error 10265]: This command is not allowed on an ACID table usa_prez_tx with a non-ACID transaction manager. Failed command: null
You cannot use ACID table to load other tables, non-ACID in non-ACID session [hive.txn.manager=org.apache.hadoop.hive.ql.lockmgr.DummyTxnManager]

Hive Variables in hive shell:
Hive conf variables are limited to session only.

Setting hive variable for city & using in where clause:

Setting hive variable for table and id & using both city & table_name, city & id.

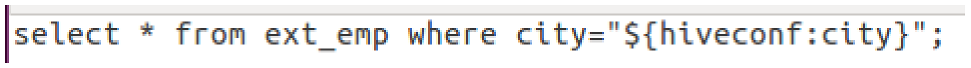
Hive Variables in bash shell:

Assigning variable & using in query:

Executing multiple statements at session level during runtime:

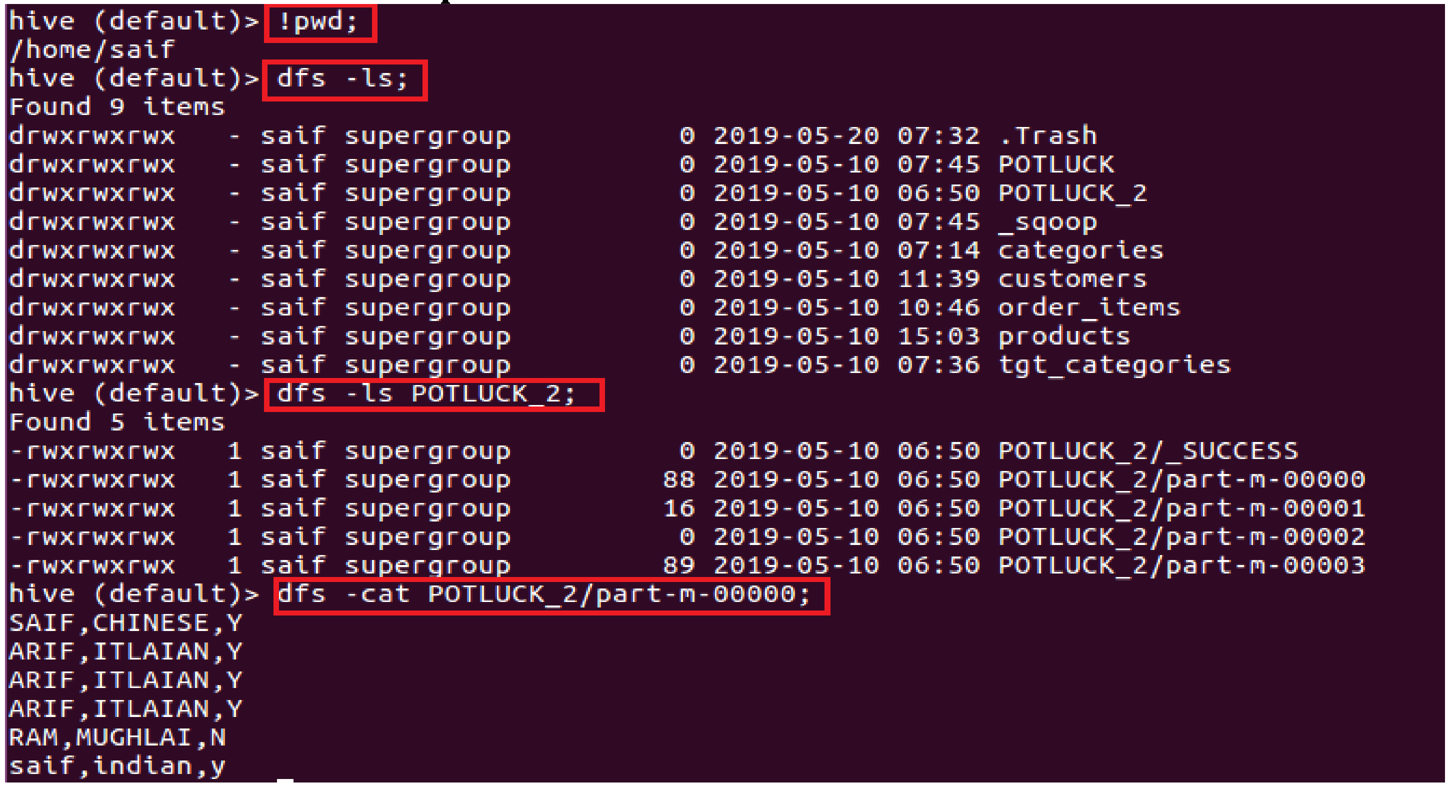
Executing statements from .hql file:

File:



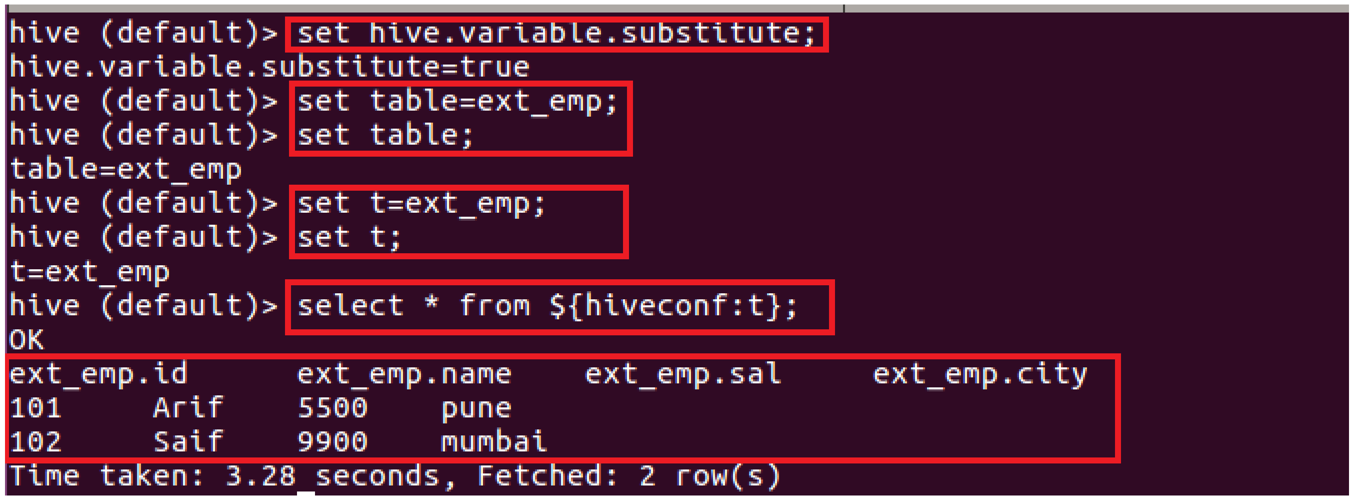
```
select * from ext_emp where city="${hiveconf:city}";
```

How to run Unix & Hadoop commands in Hive:



```
hive (default)> !pwd;
/home/saif
hive (default)> dfs -ls;
Found 9 items
drwxrwxrwx   - saif supergroup          0 2019-05-20 07:32 .Trash
drwxrwxrwx   - saif supergroup          0 2019-05-10 07:45 POTLUCK
drwxrwxrwx   - saif supergroup          0 2019-05-10 06:50 POTLUCK_2
drwxrwxrwx   - saif supergroup          0 2019-05-10 07:45 _sqoop
drwxrwxrwx   - saif supergroup          0 2019-05-10 07:14 categories
drwxrwxrwx   - saif supergroup          0 2019-05-10 11:39 customers
drwxrwxrwx   - saif supergroup          0 2019-05-10 10:46 order_items
drwxrwxrwx   - saif supergroup          0 2019-05-10 15:03 products
drwxrwxrwx   - saif supergroup          0 2019-05-10 07:36 tgt_categories
hive (default)> dfs -ls POTLUCK_2;
Found 5 items
-rwxrwxrwx   1 saif supergroup          0 2019-05-10 06:50 POTLUCK_2/_SUCCESS
-rwxrwxrwx   1 saif supergroup         88 2019-05-10 06:50 POTLUCK_2/part-m-00000
-rwxrwxrwx   1 saif supergroup         16 2019-05-10 06:50 POTLUCK_2/part-m-00001
-rwxrwxrwx   1 saif supergroup          0 2019-05-10 06:50 POTLUCK_2/part-m-00002
-rwxrwxrwx   1 saif supergroup         89 2019-05-10 06:50 POTLUCK_2/part-m-00003
hive (default)> dfs -cat POTLUCK_2/part-m-00000;
SAIF,CHINESE,Y
ARIF,ITLAIAN,Y
ARIF,ITLAIAN,Y
ARIF,ITLAIAN,Y
RAM,MUGHLAI,N
saif,indian,y
```

Variable substitution:



```
hive (default)> set hive.variable.substitute;
hive.variable.substitute=true
hive (default)> set table=ext_emp;
hive (default)> set table;
table=ext_emp
hive (default)> set t=ext_emp;
hive (default)> set t;
t=ext_emp
hive (default)> select * from ${hiveconf:t};
OK
ext_emp.id      ext_emp.name    ext_emp.sal     ext_emp.city
101     Arif    5500    pune
102     Saif    9900    mumbai
Time taken: 3.28 seconds, Fetched: 2 row(s)
```

File Compressions:

Compression is implemented in Hadoop as Hive, MapReduce, or any other processing component that results in several Network bandwidths between the nodes for I/O and for storage (not to mention the redundant storage to help fault tolerance). To reduce the amount of disk space that the Hive queries use, you should enable the Hive compression codecs.

Advantages:
1) Occupies less space.
2) Uses less bandwidth in network.

Files can be compressed in any phase of job:
1) Compressing Input File.
2) Compressing Map Output.
3) Compressing Reducer Output.

To compress your Map Output the property is:

Default Compression Codec:

Different Codec are:

Compression Format	Tool	Algorithm	File Extension	Splittable
Gzip	gzip	DEFLATE	.gz	No
bzip2	bizp2	bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	Yes if indexed
Snappy	N/A	snappy	.snappy	No

You can change default codec by below commands:

Let us perceive how to activate the compression codecs in a Hive system. There are 2 places where you can modify compression codecs in Hive, one is through the intermediate process, and an alternative is while writing the output of a Hive query to the HDFS location.
Users can forever enable or disable this within the Hive session for every query. These properties are set within the hive.site.xml or within the Hive session via the Hive command line interface.

- 4mc com.hadoop.compression.fourmc.FourMcCodec
- gzip org.apache.hadoop.io.compress.GzipCodec
- lzo com.hadoop.compression.lzo.LzopCodec
- Snappy org.apache.hadoop.io.compress.SnappyCodec
- bzip2 org.apache.hadoop.io.compress.BZip2Codec
- lz4 org.apache.hadoop.io.compress.Lz4Codec

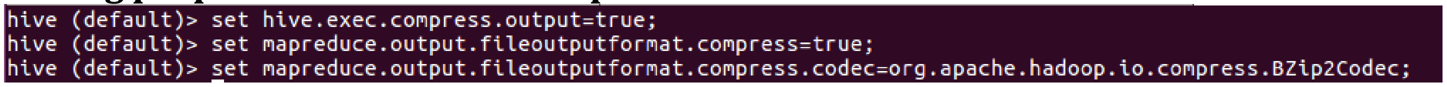
```
set hive.exec.compress.output=true;
set mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;
```

Users can also set the following properties in hive-site.xml and map-site.xml to get permanent effects.

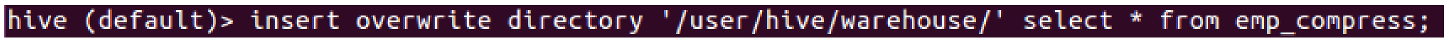
hive.exec.compress.intermediate is set to false, which implies the files created in intermediate map steps aren't compressed. Set this to true so that the I/O and Network take up less bandwidth.
hive.exec.compress.output is false, this parameter is observed if the ultimate output to HDFS is compressed.
The parameters mapred.map.output.compression.codec and mapred.output.compression.codec within the config file (/etc/hive/conf/mapred-site.xml).
MapReduce is one sort of application which will run on a Hadoop platform, thus, all the apps using the MapReduce framework goes in here.

Create table & load data into it:

Setting properties in HIVE for compression codecs:

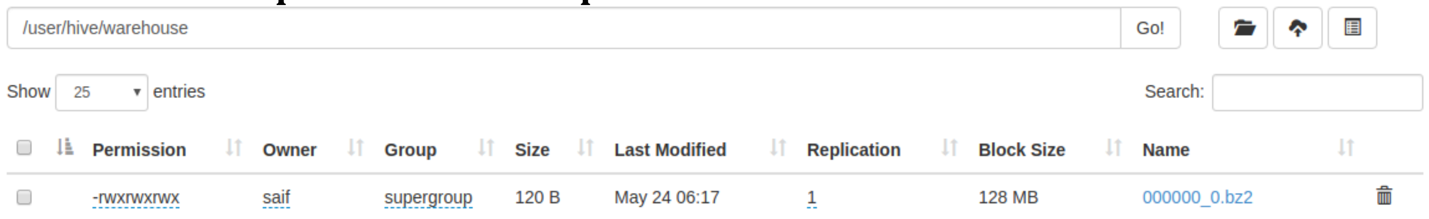


```
hive (default)> set hive.exec.compress.output=true;
hive (default)> set mapreduce.output.fileoutputformat.compress=true;
hive (default)> set mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.BZip2Codec;
```



```
hive (default)> insert overwrite directory '/user/hive/warehouse/' select * from emp_compress;
```

Check if a file is present in the output dir:

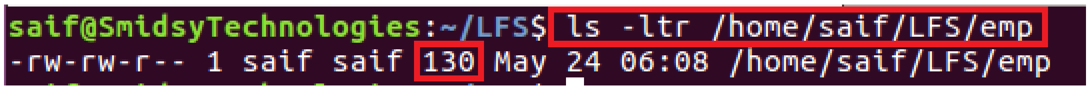


/user/hive/warehouse Go!

Show 25 entries Search:

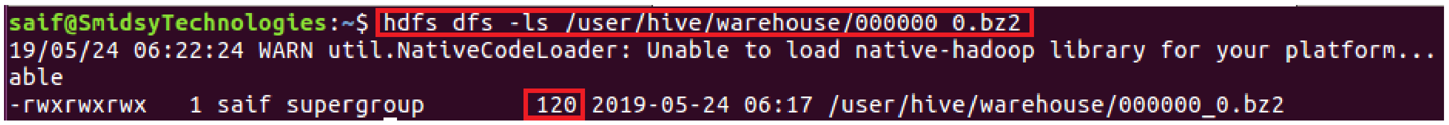
	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
	-rwxrwxrwx	saif	supergroup	120 B	May 24 06:17	1	128 MB	000000_0.bz2

Comparing the size of the original file and compressed the file.
LFS:



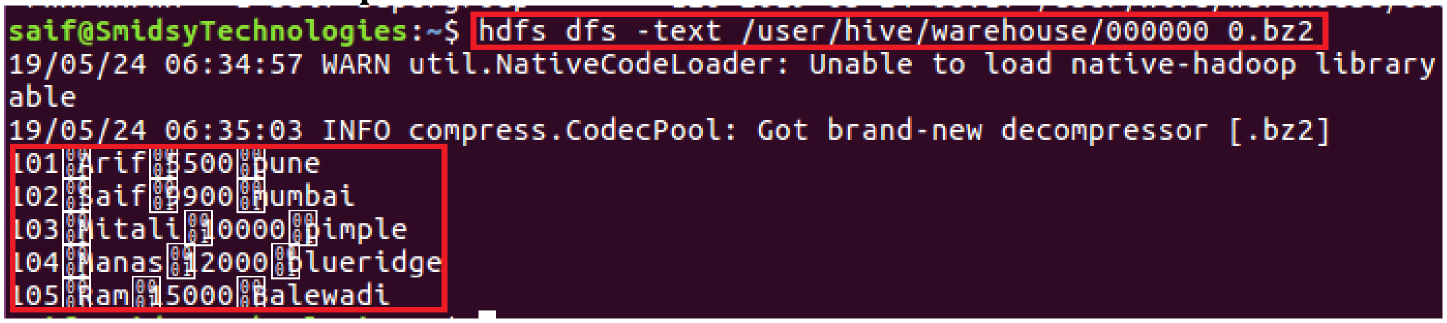
```
saif@SmidsyTechnologies:~/LFS$ ls -ltr /home/saif/LFS/emp
-rw-rw-r-- 1 saif saif 130 May 24 06:08 /home/saif/LFS/emp
```

HFS:



```
saif@SmidsyTechnologies:~$ hdfs dfs -ls /user/hive/warehouse/000000_0.bz2
19/05/24 06:22:24 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform...
able
-rwxrwxrwx   1 saif supergroup        120 2019-05-24 06:17 /user/hive/warehouse/000000_0.bz2
```

View the data of compressed file in HFS:



```
saif@SmidsyTechnologies:~$ hdfs dfs -text /user/hive/warehouse/000000_0.bz2
19/05/24 06:34:57 WARN util.NativeCodeLoader: Unable to load native-hadoop library
able
19/05/24 06:35:03 INFO compress.CodecPool: Got brand-new decompressor [.bz2]
101Arif5500pune
102saif900mumbai
103Mitali10000pimple
104Manas12000blueridge
105Ram15000Balewadi
```

Windowing Functions in Hive:

Windowing allows you to create a window on a set of data further allowing aggregation surrounding that data. Windowing in Hive is introduced from Hive 0.11.

Windowing in Hive includes the following functions

1) Lead: This function returns the values from the following rows. You can specify an integer offset which designates the row position else it will take the default integer offset as 1.

- The number of rows to lead can optionally be specified. If the number of rows to lead is not specified, the lead is one row.
- Returns null when the lead for the current row extends beyond the end of the window.

```
hive (default)> select emp_id, job, lead(deptno,1) over(order by deptno) as next_row from def_emp;
```

2) Lag: This function returns the values of the previous row. You can specify an integer offset which designates the row position else it will take the default integer offset as 1.

- The number of rows to lag can optionally be specified. If the number of rows to lag is not specified, the lag is one row.
- Returns null when the lag for the current row extends before the beginning of the window.

```
hive (default)> select emp_id, job, lag(deptno,1) over(order by deptno) as next_row from def_emp;
```

3) FIRST_VALUE: It returns the value of the first row from that window.

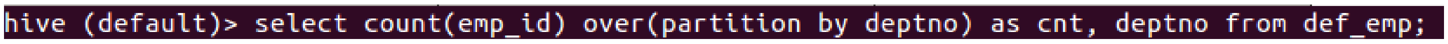
```
hive (default)> select deptno,first_value(sal) over(partition by deptno) as first_row from def_emp;
```

4) LAST_VALUE: It is the reverse of FIRST_VALUE. It returns the value of the last row from that window.

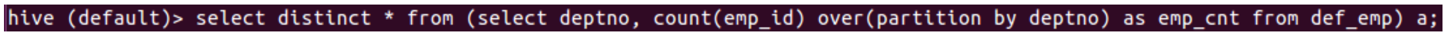
```
hive (default)> select deptno,last_value(sal) over(partition by deptno) as first_row from def_emp;
```

5) The OVER clause: OVER with standard aggregates:

a) COUNT: It returns the count of all the values for the expression written in the over clause.

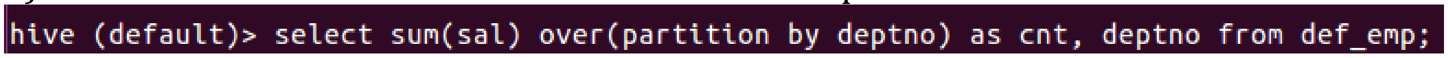


```
hive (default)> select count(emp_id) over(partition by deptno) as cnt, deptno from def_emp;
```



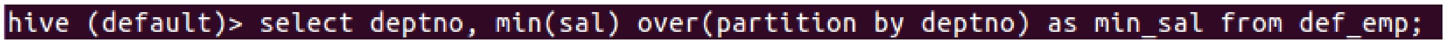
```
hive (default)> select distinct * from (select deptno, count(emp_id) over(partition by deptno) as emp_cnt from def_emp) a;
```

b) SUM: It returns the sum of all the values for the expression written in the over clause.



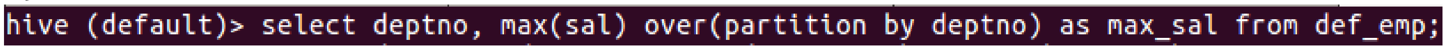
```
hive (default)> select sum(sal) over(partition by deptno) as cnt, deptno from def_emp;
```

c) MIN: It returns the minimum value of the column for the rows in that over clause.



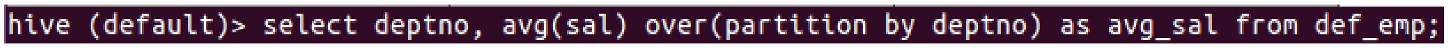
```
hive (default)> select deptno, min(sal) over(partition by deptno) as min_sal from def_emp;
```

d) MAX: It returns the maximum value of the column for the rows in that over clause.



```
hive (default)> select deptno, max(sal) over(partition by deptno) as max_sal from def_emp;
```

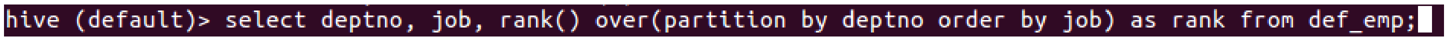
e) AVG: It returns the average value of the column for the rows that over clause returns.



```
hive (default)> select deptno, avg(sal) over(partition by deptno) as avg_sal from def_emp;
```

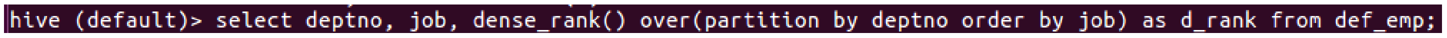
6) OVER with PARTITION BY and ORDER BY with one or more partitioning and/or ordering columns:

a) RANK: The rank function will return the rank of the values as per the result set of the over clause. If two values are same then it will give the same rank to those 2 values and then for the next value, the sub-sequent rank will be skipped.



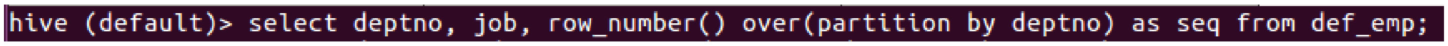
```
hive (default)> select deptno, job, rank() over(partition by deptno order by job) as rank from def_emp;
```

b) DENSE_RANK: It is same as the rank () function but the difference is if any duplicate value is present then the rank will not be skipped for the subsequent rows. Each unique value will get the ranks in a sequence.



```
hive (default)> select deptno, job, dense_rank() over(partition by deptno order by job) as d_rank from def_emp;
```

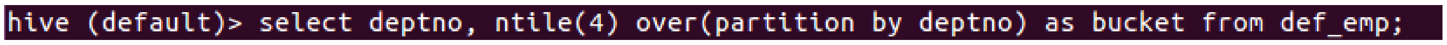
c) ROW_NUMBER: Row number will return the continuous sequence of numbers for all the rows of the result set of the over clause.



```
hive (default)> select deptno, job, row_number() over(partition by deptno) as seq from def_emp;
```

d) PERCENT_RANK: It returns the percentage rank of each row within the result set of over clause. percent_rank is calculated in accordance with the rank of the row and the calculation is as follows (rank-1)/(total_rows_in_group – 1). If the result set has only one row then the percent_rank will be 0.

e) NTILE: It returns the bucket number of the particular value. For suppose if you say ntile(5) then it will create 5 buckets based on the result set of the over clause after that it will place the first 20% of the records in the 1st bucket and so on till 5th bucket.



```
hive (default)> select deptno, ntile(4) over(partition by deptno) as bucket from def_emp;
```