EXP 6: HIVE

Aim: Perform Hive commands.

HIVE: Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.
This is a brief tutorial that provides an introduction on how to use Apache Hive HiveQL with Hadoop Distributed File System. This tutorial can be your first step towards becoming a successful Hadoop Developer with Hive.

Hive vs SQL: SQL is a general purpose database language that has extensively been used for both transactional and analytical queries. Hive, on the other hand, is built with an analytical focus. What this means is Hive lacks update and delete functions but is superfast in reading and processing huge volumes of data faster than SQL. Hence, even though Hive SQL is SQL-like, lack of support for modifying or deleting data is a major difference.
Despite the working differences, once you enter the Hive world from SQL, similarity in language ensures smooth transition but it is important to note the differences in constructs and syntax, else you're in for frustrating times.
Now that we have an introduction to all the three data mining languages and their head to head comparisons, we close this article by analyzing different situations and listing the best fit option for each.

When is it best to use Apache Hive?
Big Data enterprises require fast analysis of data collected over a period of time. Hive is an excellent tool for analytical querying of historical data. It is to be noted that the data needs to be well organized, which would allow Hive to fully unleash its processing and analytical prowess. Querying real time data with Hive might not be the best idea, as it would be a time consuming job, defeating the original goal of fast processing (HBase is the answer for real time analytics, successfully utilized by Facebook).

HIVE ARCHITECTURE:

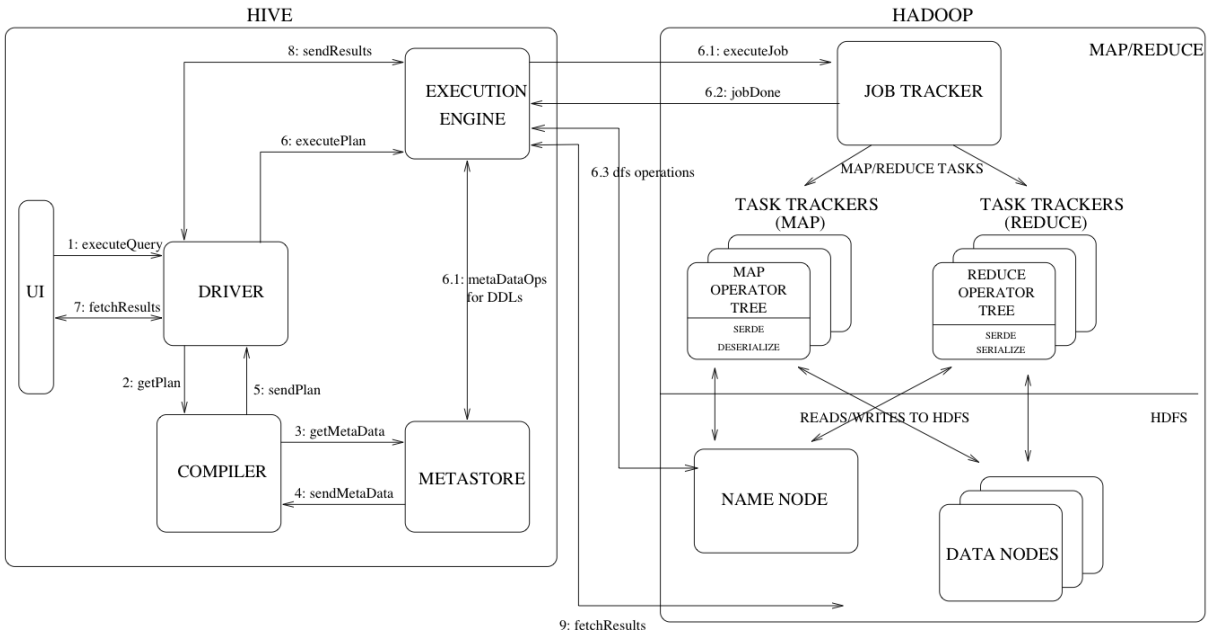

Figure 1 shows the major components of Hive and its interactions with Hadoop. As shown in that figure, the main components of Hive are:

- UI – The user interface for users to submit queries and other operations to the system. As of 2011 the system had a command line interface and a web based GUI was being developed.
- Driver – The component which receives the queries. This component implements the notion of session handles and provides execute and fetch APIs modelled on JDBC/ODBC interfaces.
- Compiler – The component that parses the query, does semantic analysis on the different query blocks and query expressions and eventually generates an execution plan with the help of the table and partition metadata looked up from the metastore.
- Metastore – The component that stores all the structure information of the various tables and partitions in the warehouse including column and column type information, the serializers and deserializers necessary to read and write data and the corresponding HDFS files where the data is stored.
- Execution Engine – The component which executes the execution plan created by the compiler. The plan is a DAG of stages. The execution engine manages the dependencies between these different stages of the plan and executes these stages on the appropriate system components.

  Figure 1 also shows how a typical query flows through the system. The UI calls the execute interface to the Driver (step 1 in Figure 1).

  The Driver creates a session handle for the query and sends the query to the compiler to generate an execution plan (step 2).

  The compiler gets the necessary metadata from the metastore (steps 3 and 4).

  This metadata is used to typecheck the expressions in the query tree as well as to prune partitions based on query predicates. The plan generated by the compiler (step 5) is a DAG of stages with each stage being either a map/reduce job, a metadata operation or an operation on HDFS.

  For map/reduce stages, the plan contains map operator trees (operator trees that are executed on the mappers) and a reduce operator tree (for operations that need reducers). The execution engine submits these stages to appropriate components (steps 6, 6.1, 6.2 and 6.3). In each task (mapper/reducer) the deserializer associated with the table or intermediate outputs is used to read the rows from HDFS files and these are passed through the associated operator tree. Once the output is generated, it is written to a temporary HDFS file though the serializer (this happens in the mapper in case the operation does not need a reduce).

  The temporary files are used to provide data to subsequent map/reduce stages of the plan. For DML operations the final temporary file is moved to the table's location. This scheme is used to ensure that dirty data is not read (file rename being an atomic operation in HDFS). For queries, the contents of the temporary file are read by the execution engine directly from HDFS as part of the fetch call from the Driver (steps 7, 8 and 9).

Hive Data Model

Data in Hive is organized into:

- **Tables** – These are analogous to Tables in Relational Databases. Tables can be filtered, projected, joined and unioned. Additionally all the data of a table is stored in a directory in HDFS. Hive also supports the notion of external tables wherein a table can be created on prexisting files or directories in HDFS by providing the appropriate location to the table creation DDL. The rows in a table are organized into typed columns similar to Relational Databases.
- **Partitions** – Each Table can have one or more partition keys which determine how the data is stored, for example a table T with a date partition column ds had files with data for a particular date stored in the <table location>/ds=<date> directory in HDFS. Partitions allow the system to prune data to be inspected based on query predicates, for example a query that is interested in rows from T that satisfy the predicate T.ds = '2008-09-01' would only have to look at files in <table location>/ds=2008-09-01/ directory in HDFS.
- **Buckets** – Data in each partition may in turn be divided into Buckets based on the hash of a column in the table. Each bucket is stored as a file in the partition directory. Bucketing allows the system to

efficiently evaluate queries that depend on a sample of data (these are queries that use the SAMPLE clause on the table).
Apart from primitive column types (integers, floating point numbers, generic strings, dates and booleans), Hive also supports arrays and maps. Additionally, users can compose their own types programmatically from any of the primitives, collections or other user-defined types. The typing system is closely tied to the SerDe (Serailization/Deserialization) and object inspector interfaces. User can create their own types by implementing their own object inspectors, and using these object inspectors they can create their own SerDes to serialize and deserialize their data into HDFS files). These two interfaces provide the necessary hooks to extend the capabilities of Hive when it comes to understanding other data formats and richer types. Builtin object inspectors like ListObjectInspector, StructObjectInspector and MapObjectInspector provide the necessary primitives to compose richer types in an extensible manner. For maps (associative arrays) and arrays useful builtin functions like size and index operators are provided. The dotted notation is used to navigate nested types, for example a.b.c = 1 looks at field c of field b of type a and compares that with 1.

Hive Query Language
HiveQL is an SQL-like query language for Hive. It mostly mimics SQL syntax for creation of tables, loading data into tables and querying the tables. HiveQL also allows users to embed their custom map-reduce scripts. These scripts can be written in any language using a simple row-based streaming interface – read rows from standard input and write out rows to standard output. This flexibility comes at a cost of a performance hit caused by converting rows from and to strings. However, we have seen that users do not mind this given that they can implement their scripts in the language of their choice. Another feature unique to HiveQL is multi-table insert. In this construct, users can perform multiple queries on the same input data using a single HiveQL query. Hive optimizes these queries to share the scan of the input data, thus increasing the throughput of these queries several orders of magnitude.

Hive Data Types
Hive supports different data types to be used in table columns. The data types supported by Hive can be broadly classified in Primitive and Complex data types.

A. Primitive data types:
The primitive data types supported by Hive are listed below:

a. Numeric Types
- TINYINT (1-byte signed integer, from -128 to 127)
- SMALLINT (2-byte signed integer, from -32,768 to 32,767)
- INT (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)
- BIGINT (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- FLOAT (4-byte single precision floating point number)
- DOUBLE (8-byte double precision floating point number)
- DECIMAL (Hive 0.13.0 introduced user definable precision and scale)

b. Date/Time Types
- TIMESTAMP
- DATE

c. String Types
- STRING
- VARCHAR
- CHAR

d. Misc Types

- BOOLEAN
- BINARY

Apart from these primitive data types Hive offers some complex data types which are listed below:

B. Complex data types

a. Complex Types

- arrays: ARRAY<data_type>
- maps: MAP<primitive_type, data_type>
- structs: STRUCT<col_name : data_type [COMMENT col_comment], ...>
- union: UNIONTYPE<data_type, data_type, ...>

COMMANDS:

```
[cloudera@quickstart ~]$ vi emp.tab

[cloudera@quickstart ~]$ hdfs dfs -mkdir /user/cloudera/npzt

[cloudera@quickstart ~]$ hdfs dfs -put emp.tab /user/cloudera/npzt

[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/
Found 3 items
-rw-r--r--   1 cloudera cloudera       46 2019-09-28 00:17 /user/cloudera/group.csv
-rw-r--r--   1 cloudera cloudera       30 2019-09-28 00:23 /user/cloudera/group1.csv
drwxr-xr-x   - cloudera cloudera        0 2019-09-28 22:13 /user/cloudera/npzt

[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/npzt
Found 1 items
-rw-r--r--   1 cloudera cloudera       88 2019-09-28 22:13 /user/cloudera/npzt/emp.tab

[cloudera@quickstart ~]$ vi emp.tab

[cloudera@quickstart ~]$ hdfs dfs -cat /user/cloudera/npzt/emp.tab
1       zoya    20000   java
2       taniya  30000   android
3       pranali 40000   python
4       nazmeen 30000   python3

[cloudera@quickstart ~]$ hive
Logging initialized using configuration in file:/etc/hive/conf.dist/hive-log4j.properties
WARNING: Hive CLI is deprecated and migration to Beeline is recommended.

hive> CREATE TABLE IF NOT EXISTS employee( eid int,name String,salary String,destination String)
    > COMMENT 'Employee details'
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '\t'
    > LINES TERMINATED BY '\n'
    > STORED AS TEXTFILE;
OK
Time taken: 7.202 seconds
```

//error(you need superuser permission and superuser shouls hive or hdfs)

hive**> LOAD DATA INPATH '/user/cloudera/npzt'**
> OVERWRITE INTO TABLE employee;
Loading data to table default.employee
chgrp: changing ownership of 'hdfs://quickstart.cloudera:8020/user/hive/warehouse/employee': User does not belong to supergroup
Table default.employee stats: [numFiles=1, numRows=0, totalSize=88, rawDataSize=0]
OK
Time taken: 5.346 seconds

//ctrl+d correct

[cloudera@quickstart ~]$ **sudo su hdfs**
bash-4.1$ **hive**

Logging initialized using configuration in file:/etc/hive/conf.dist/hive-log4j.properties
WARNING: Hive CLI is deprecated and migration to Beeline is recommended.

// if already loaded and agian we are trying to load then it gives error as ** FAILED:

SemanticException Line 1:17 Invalid path ''/user/cloudera/npzt'': No files matching path hdfs://quickstart.cloudera:8020/user/cloudera/npzt **

hive> **select * from employee;**
OK
1	zoya	20000	java
2	taniya	30000	android
3	pranali	40000	python
4	nazmeen		30000	python3
Time taken: 3.398 seconds, Fetched: 4 row(s)

hive> **select * from employee where destination='java';**
OK
1	zoya	20000	java
Time taken: 1.118 seconds, Fetched: 1 row(s)

hive> **select sum(salary) from employee;**
Query ID = cloudera_20190928225656_4de1747c-bb30-4193-8f81-8da337ba1bdf
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_1569733481333_0001, Tracking URL = http://quickstart.cloudera:8088/proxy/application_1569733481333_0001/
Kill Command = /usr/lib/hadoop/bin/hadoop job -kill job_1569733481333_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2019-09-28 22:57:33,949 Stage-1 map = 0%, reduce = 0%
2019-09-28 22:58:11,652 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 5.85 sec
2019-09-28 22:58:42,591 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 12.09 sec
MapReduce Total cumulative CPU time: 12 seconds 90 msec

```
Ended Job = job_1569733481333_0001
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 12.09 sec   HDFS Read: 7917 HDFS Write: 9 SUCCESS
Total MapReduce CPU Time Spent: 12 seconds 90 msec
OK
120000.0
Time taken: 142.197 seconds, Fetched: 1 row(s)
```

// To see the structure of database and to find the path where data is stored bcz hive doesn't has storage, storage is hdfs.

```
hive> show create table employee;
OK
CREATE TABLE `employee`(
  `eid` int,
  `name` string,
  `salary` string,
  `destination` string)
COMMENT 'Employee details'
ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'field.delim'='\t',
  'line.delim'='\n',
  'serialization.format'='\t')
STORED AS INPUTFORMAT
  'org.apache.hadoop.mapred.TextInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
LOCATION
  'hdfs://quickstart.cloudera:8020/user/hive/warehouse/employee'
TBLPROPERTIES (
  'COLUMN_STATS_ACCURATE'='true',
  'numFiles'='1',
  'numRows'='0',
  'rawDataSize'='0',
  'totalSize'='88',
  'transient_lastDdlTime'='1569735482')
Time taken: 1.255 seconds, Fetched: 25 row(s)
```

//to check whether file is loaded in warehouse or not back to the main directory.

```
[cloudera@quickstart ~]$ hdfs dfs -ls /user/hive/warehouse/employee
Found 1 items
-rwxrwxrwx   1 cloudera cloudera         88 2019-09-28 22:13 /user/hive/warehouse/employee/emp.tab
```