FREE eBook

LEARNING hive

Free unaffiliated eBook created from
Stack Overflow contributors.

#hive

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: hive

It is an unofficial and free hive ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official hive.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with hive

Remarks

- Hive is a data warehouse tool built on top of Hadoop.
- It provides an SQL-like language to query data.
- We can run almost all the SQL queries in Hive, the only difference, is that, it runs a map-reduce job at the backend to fetch result from Hadoop Cluster. Because of this Hive sometimes take more time to fetch the result-set.

Examples

Word Count Example in Hive

Docs file (Input File)

Mary had a little lamb

its fleece was white as snow

and everywhere that Mary went

the lamb was sure to go.

Hive Query

```
CREATE TABLE FILES (line STRING);

LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE FILES;

CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
(SELECT explode(split(line, ' ')) AS word FROM FILES) w
GROUP BY word
ORDER BY word;
```

Output of word_counts table in Hive

Mary,2

had,1

a,1

little,1

lamb,2

its,1

fleece,1

was,2

white,1

as,1

snow,1

and,1

everywhere,1

that,1

went,1

the,1

sure,1

to,1

go,1

Installation of Hive(linux)

Start by downloading the latest stable release from https://hive.apache.org/downloads.html

-> Now untar the file with

$ tar -xvf hive-2.x.y-bin.tar.gz

-> Create a directory in the /usr/local/ with

$ sudo mkdir /usr/local/hive

-> Move the file to root with

$ mv ~/Downloads/hive-2.x.y /usr/local/hive

-> Edit environment variablesfor hadoop and hive in .bashrc

$ gedit ~/.bashrc

like this

export HIVE_HOME=/usr/local/hive/apache-hive-2.0.1-bin/

export PATH=$PATH:$HIVE_HOME/bin

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

export CLASSPATH=$CLASSPATH:/usr/local/hive/apache-hive-2.0.1-bin/lib/*:.

-> Now, start hadoop if it is not already running. And make sure that it is running and it is not in safe mode.

$ hadoop fs -mkdir /user/hive/warehouse

The directory "warehouse" is the location to store the table or data related to hive.

$ hadoop fs -mkdir /tmp

The temporary directory “tmp” is the temporary location to store the intermediate result of processing.

-> Set Permissions for read/write on those folders.

$ hadoop fs -chmod g+w /user/hive/warehouse

$ hadoop fs -chmod g+w /user/tmp

-> Now fire up HIVE with this command in console

$ hive

Hive Installation with External Metastore in Linux

Pre-requisites:

1. Java 7
2. Hadoop (Refer here for Hadoop Installation)
3. Mysql Server and Client

Installation:

Step 1: Download the latest Hive tarball from the downloads page.

Step 2: Extract the downloaded tarball (***Assumption:*** *The tarball is downloaded in $HOME*)

```
tar -xvf /home/username/apache-hive-x.y.z-bin.tar.gz
```

Step 3: Update the environment file (`~/.bashrc`)

```
export HIVE_HOME=/home/username/apache-hive-x.y.z-bin
export PATH=$HIVE_HOME/bin:$PATH
```

source the file to set the new environment variables.

```
source ~/.bashrc
```

Step 4: Download the JDBC connector for mysql from here and extract it.

```
tar -xvf mysql-connector-java-a.b.c.tar.gz
```

The extracted directory contains the connector jar file `mysql-connector-java-a.b.c.jar`. Copy it to the `lib` of `$HIVE_HOME`

```
cp mysql-connector-java-a.b.c.jar $HIVE_HOME/lib/
```

Configuration:

Create the hive configuration file `hive-site.xml` under `$HIVE_HOME/conf/` directory and add the following metastore related properties.

```
<configuration>
   <property>
      <name>javax.jdo.option.ConnectionURL</name>
      <value>jdbc:mysql://localhost/hive_meta</value>
      <description>JDBC connect string for a JDBC metastore</description>
   </property>

   <property>
      <name>javax.jdo.option.ConnectionDriverName</name>
      <value>com.mysql.jdbc.Driver</value>
      <description>Driver class name for a JDBC metastore</description>
   </property>

   <property>
       <name>javax.jdo.option.ConnectionUserName</name>
       <value>mysqluser</value>
       <description>username to use against metastore database</description>
   </property>

   <property>
       <name>javax.jdo.option.ConnectionPassword</name>
       <value>mysqlpass</value>
       <description>password to use against metastore database</description>
   </property>

   <property>
       <name>datanucleus.autoCreateSchema</name>
       <value>false</value>
   </property>

   <property>
       <name>datanucleus.fixedDatastore</name>
       <value>true</value>
   </property>
</configuration>
```

Update the values of MySQL "username" and "password" accordingly in the properties.

Create the Metastore Schema:

The metastore schema scripts are available under `$HIVE_HOME/scripts/metastore/upgrade/mysql/`

Login to MySQL and source the schema,

```
mysql -u username -ppassword

mysql> create database hive_meta;
mysql> use hive_meta;
mysql> source hive-schema-x.y.z.mysql.sql;
mysql> exit;
```

Starting Metastore:

```
hive --service metastore
```

To run it in background,

```
nohup hive --service metastore &
```

Starting HiveServer2: (Use if required)

```
hiveserver2
```

To run it in background,

```
nohup hiveserver2 metastore &
```

Note: These executables are available under `$HIVE_HOME/bin/`

Connect:

Use either `hive`, `beeline` or Hue to connect with Hive.

Hive CLI is deprecated, using Beeline or Hue is recommended.

Additional Configurations for Hue:

Update this value in `$HUE_HOME/desktop/conf/hue.ini`

```
[beeswax]
    hive_conf_dir=/home/username/apache-hive-x.y.z-bin/conf
```

Read Getting started with hive online: https://riptutorial.com/hive/topic/1099/getting-started-with-hive

Chapter 2: Create Database and Table Statement

Syntax

- CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name

  [(col_name data_type [COMMENT col_comment], ...)] [COMMENT table_comment] [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)] [CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS] [SKEWED BY (col_name, col_name, ...) -- (Note: Available in Hive 0.10.0 and later)] ON ((col_value, col_value, ...), (col_value, col_value, ...), ...) [STORED AS DIRECTORIES] [[ROW FORMAT row_format] [STORED AS file_format] | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]] [LOCATION hdfs_path] [TBLPROPERTIES (property_name=property_value, ...)]
  [AS select_statement];

- CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name LIKE existing_table_or_view_name [LOCATION hdfs_path];

- data_type : primitive_type,array_type,map_type,struct_type,union_type

- primitive_type: TINYINT, SMALLINT, INT , BIGINT, BOOLEAN, FLOAT, DOUBLE, STRING, BINARY, TIMESTAMP, DECIMAL, DECIMAL(precision, scale), DATE, VARCHAR, CHAR

- array_type: ARRAY < data_type >

- map_type: MAP < primitive_type, data_type >

- struct_type: STRUCT < col_name : data_type [COMMENT col_comment], ...>

- union_type: UNIONTYPE < data_type, data_type, ... >

- row_format: DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char] [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char] [NULL DEFINED AS char]
  , SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, ...)]

- file_format: : SEQUENCEFILE , TEXTFILE , RCFILE , ORC , PARQUET , AVRO , INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

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

Remarks

When working with tables and databases in HIVE. Below points can be usefull.

- We can switch database using `use database;` command
- To know the current working database we can get using `SELECT current_database()`
- To see the DDL used for create table statement we can use `SHOW CREATE TABLE tablename`
- To see all columns of table use `DESCRIBE tablename` to show extended details like location serde used and others `DESCRIBE FORMATTED tablename`. DESCRIBE can also be abbrevated as DESC.

Examples

Create Table

Creating a **managed** table with partition and stored as a sequence file. The data format in the files is assumed to be field-delimited by `Ctrl-A (^A)` and row-delimited by newline. The below table is created in hive warehouse directory specified in value for the key `hive.metastore.warehouse.dir` in the Hive config file `hive-site.xml`.

```
CREATE TABLE view
(time INT,
id BIGINT,
url STRING,
referrer_url STRING,
add STRING COMMENT 'IP of the User')
COMMENT 'This is view table'
PARTITIONED BY(date STRING, region STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
STORED AS SEQUENCEFILE;
```

Creating a **external** table with partitions and stored as a sequence file. The data format in the files is assumed to be field-delimited by `ctrl-A` and row-delimited by newline. The below table is created in the location specified and it comes handy when we already have data. One of the advantages of using an external table is that we can drop the table without deleting the data. For instance, if we create a table and realize that the schema is wrong, we can safely drop the table and recreate with the new schema without worrying about the data.Other advantage is that if we are using other tools like pig on same files, we can continue using them even after we delete the table.

```
CREATE EXTERNAL TABLE view
(time INT,
id BIGINT,
url STRING,
referrer_url STRING,
add STRING COMMENT 'IP of the User')
COMMENT 'This is view table'
PARTITIONED BY(date STRING, region STRING)
ROW FORMAT DELIMITED
```

```
FIELDS TERMINATED BY '\001'
STORED AS SEQUENCEFILE
LOCATION '<hdfs_location>';
```

Creating a table using select query and populating results from query, these statements are known as **CTAS(Create Table As Select)**.

There are two parts in CTAS, the SELECT part can be any SELECT statement supported by HiveQL. The CREATE part of the CTAS takes the resulting schema from the SELECT part and creates the target table with other table properties such as the SerDe and storage format.

CTAS has these restrictions:

- The target table cannot be a partitioned table.
- The target table cannot be an external table.
- The target table cannot be a list bucketing table.

```
CREATE TABLE new_key_value_store
ROW FORMAT SERDE "org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe"
STORED AS RCFile
AS
SELECT * FROM page_view
SORT BY url, add;
```

Create Table Like:

The **LIKE form of CREATE TABLE** allows you to copy an existing table definition exactly (without copying its data). In contrast to CTAS, the statement below creates a new table whose definition exactly matches the existing table in all particulars other than table name. The new table contains no rows.

```
CREATE TABLE empty_page_views
LIKE page_views;
```

Create Database

Creating a database in a particular location. If we dont specify any location for database its created in warehouse directory.

```
CREATE DATABASE IF NOT EXISTS db_name
COMMENT 'TEST DATABASE'
LOCATION /PATH/HDFS/DATABASE/;
```

Hive ACID table creation.

ACID tables are supported since hive 0.14 version. Below table supports UPDATE/DELETE/INSERT

Below configuration changes required in hive-site.xml.

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

Currently only orc file is format supported.

Table create statement.

```
create table Sample_Table(
col1 Int,
col2 String,
col3 String)
clustered by (col3) into 3 buckets
stored as orc
TBLPROPERTIES ('transactional'='true');
```

HIVE_HBASE Integration

Hive-Hbase integration is supported since below versions. Hive: 0.11.0 HBase: 0.94.2 Hadoop: 0.20.2

```
CREATE TABLE hbase_hive
(id string,
 col1 string,
 col2 string,
 col3 int)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES
("hbase.columns.mapping" = ":key,cf1:col1,cf1:col2,cf1:col3")
TBLPROPERTIES ("hbase.table.name" = "hive_hbase");
```

Note: 1st column should be the key column.

Create table using existing table properties.

```
CREATE TABLE new_table_name LIKE existing_table_name;
```

Read Create Database and Table Statement online: https://riptutorial.com/hive/topic/3328/create-database-and-table-statement

Chapter 3: Export Data in Hive

Examples

Export feature in hive

Exporting data from employees table to /tmp/ca_employees

INSERT OVERWRITE LOCAL DIRECTORY '/tmp/ca_employees' SELECT name, salary, address FROM employees WHERE se.state = 'CA';

Exporting data from employees table to multiple local directories based on specific condition

The below query shows how a single construct can be used to export data to multiple directories based on specific criteria

```
FROM employees se INSERT OVERWRITE DIRECTORY '/tmp/or_employees' SELECT * WHERE se.cty = 'US'
and se.st = 'OR'
INSERT OVERWRITE DIRECTORY '/tmp/ca_employees' SELECT * WHERE se.cty = 'US' and se.st = 'CA'
INSERT OVERWRITE DIRECTORY '/tmp/il_employees' SELECT * WHERE se.cty = 'US' and se.st = 'IL';
```

Read Export Data in Hive online: https://riptutorial.com/hive/topic/6530/export-data-in-hive

Chapter 4: File formats in HIVE

Examples

SEQUENCEFILE

Store data in SEQUENCEFILE if the data needs to be compressed. You can import text files compressed with Gzip or Bzip2 directly into a table stored as TextFile. The compression will be detected automatically and the file will be decompressed on-the-fly during query execution.

```
CREATE TABLE raw_sequence (line STRING)
STORED AS SEQUENCEFILE;
```

ORC

The Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data. ORC file can contain lightweight indexes and bloom filters.

See: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC

ORC is a recommended format for storing data within HortonWorks distribution.

```
CREATE TABLE tab_orc (col1 STRING,
                      col2 STRING,
                      col3 STRING)
STORED AS ORC
TBLPROPERTIES (
               "orc.compress"="SNAPPY",
               "orc.bloom.filter.columns"="col1",
               "orc.create.index" = "true"
              )
```

To modify a table so that new partitions of the table are stored as ORC files:

```
ALTER TABLE T SET FILEFORMAT ORC;
```

As of Hive 0.14, users can request an efficient merge of small ORC files together by issuing a `CONCATENATE` command on their table or partition. The files will be merged at the stripe level without reserializatoin.

```
ALTER TABLE T [PARTITION partition_spec] CONCATENATE;
```

PARQUET

Parquet columnar storage format in Hive 0.13.0 and later. Parquet is built from the ground up with

complex nested data structures in mind, and uses the record shredding and assembly algorithm described in the Dremel paper. We believe this approach is superior to simple flattening of nested name spaces.

Parquet is built to support very efficient compression and encoding schemes. Multiple projects have demonstrated the performance impact of applying the right compression and encoding scheme to the data. Parquet allows compression schemes to be specified on a per-column level, and is future-proofed to allow adding more encodings as they are invented and implemented.

Parquet is recommended File Format with Impala Tables in Cloudera distributions.

See: http://parquet.apache.org/documentation/latest/

```
CREATE TABLE parquet_table_name (x INT, y STRING) STORED AS PARQUET;
```

AVRO

Avro files are been supported in Hive 0.14.0 and later.

Avro is a remote procedure call and data serialization framework developed within Apache's Hadoop project. It uses JSON for defining data types and protocols, and serializes data in a compact binary format. Its primary use is in Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services.

Specification of AVRO format: https://avro.apache.org/docs/1.7.7/spec.html

```
CREATE TABLE kst
PARTITIONED BY (ds string)
STORED AS AVRO
TBLPROPERTIES (
  'avro.schema.url'='http://schema_provider/kst.avsc');
```

We can also use below syntax without using schema file.

```
CREATE TABLE kst (field1 string, field2 int)
PARTITIONED BY (ds string)
STORED AS AVRO;
```

In the examples above `STORED AS AVRO` clause is equivalent to:

```
ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
```

Text File

TextFile is the default file format, unless the configuration parameter hive.default.fileformat has a different setting. We can create a table on hive using the field names in our delimited text file. Lets say for example, our csv file contains three fields (id, name, salary) and we want to create a table in hive called "employees". We will use the below code to create the table in hive.

```
CREATE TABLE employees (id int, name string, salary double) ROW FORMAT DELIMITED FIELDS
TERMINATED BY `,';
```

Now we can load a text file into our table:

```
LOAD DATA LOCAL INPATH '/home/ourcsvfile.csv' OVERWRITE INTO TABLE employees;
```

Displaying the contents of our table on hive to check if the data was successfully loaded:

```
SELECT * FROM employees;
```

Read File formats in HIVE online: https://riptutorial.com/hive/topic/4513/file-formats-in-hive

Chapter 5: Hive Table Creation Through Sqoop

Introduction

If we have a Hive meta-store associated with our HDFS cluster, Sqoop can import the data into Hive by generating and executing a CREATE TABLE statement to define the data's layout in Hive. Importing data into Hive is as simple as adding the --hive-import option to your Sqoop command line.

Remarks

Importing data directly from RDBMS to HIVE can solve lots of time. Also we can run a freeform query(a join or some simple query) and populate it in a table of our choice directly into Hive.

--hive-import tells Sqoop that the final destination is Hive and not HDFS.

--hive-table option helps in importing the data to the table in hive chosen by us, otherwise it will be named as the source table being imported from RDBMS.

Examples

Hive import with Destination table name in hive

```
$ sqoop import --connect jdbc:mysql://10.0.0.100/hadooptest
--username hadoopuser -P
--table table_name --hive-import --hive-table hive_table_name
```

Read Hive Table Creation Through Sqoop online: https://riptutorial.com/hive/topic/10685/hive-table-creation-through-sqoop

Chapter 6: Hive User Defined Functions (UDF's)

Examples

Hive UDF creation

To create a UDF, we need to extend UDF (`org.apache.hadoop.hive.ql.exec.UDF`) class and implement evaluate method.

Once UDF is complied and JAR is build, we need to add jar to hive context to create a temporary/permanent function.

```
import org.apache.hadoop.hive.ql.exec.UDF;

class UDFExample extends UDF {

  public String evaluate(String input) {

    return new String("Hello " + input);
  }
}

hive> ADD JAR <JAR NAME>.jar;
hive> CREATE TEMPORARY FUNCTION helloworld as 'package.name.UDFExample';
hive> select helloworld(name) from test;
```

Hive UDF to trim the given string.

```
package MyHiveUDFs;

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

public class Strip extends UDF {

private Text result = new Text();
 public Text evaluate(Text str) {
 if(str == null) {
 return null;
 }
 result.set(StringUtils.strip(str.toString()));
 return result;
 }
}
```

export the above to jar file

Go to the Hive CLI and Add the UDF JAR

```
hive> ADD jar /home/cloudera/Hive/hive_udf_trim.jar;
```

Verify JAR is in Hive CLI Classpath

```
hive> list jars;
/home/cloudera/Hive/hive_udf_trim.jar
```

Create Temporary Function

```
hive> CREATE TEMPORARY FUNCTION STRIP AS 'MyHiveUDFs.Strip';
```

UDF Output

```
 hive> select strip('   hiveUDF ') from dummy;
 OK
 hiveUDF
```

Read Hive User Defined Functions (UDF's) online: https://riptutorial.com/hive/topic/4949/hive-user-defined-functions--udf-s-

Chapter 7: Indexing

Examples

Structure

```
CREATE INDEX index_name
ON TABLE base_table_name (col_name, ...)
AS 'index.handler.class.name'
[WITH DEFERRED REBUILD]
[IDXPROPERTIES (property_name=property_value, ...)]
[IN TABLE index_table_name]
[PARTITIONED BY (col_name, ...)]
[
 [ ROW FORMAT ...] STORED AS ...
 | STORED BY ...
]
[LOCATION hdfs_path]
[TBLPROPERTIES (...)]
```

Example:

```
CREATE INDEX inedx_salary ON TABLE employee(salary) AS
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler' WITH DEFERRED REBUILD;
```

Alter Index

ALTER INDEX index_name ON table_name [PARTITION (...)] REBUILD

Drop Index

```
DROP INDEX <index_name> ON <table_name>
```

If WITH DEFERRED REBUILD is specified on CREATE INDEX, then the newly created index is initially empty (regardless of whether the table contains any data).

The ALTER INDEX REBUILD command can be used to build the index structure for all partitions or a single partition.

Read Indexing online: https://riptutorial.com/hive/topic/6365/indexing

Chapter 8: Insert Statement

Syntax

- **Standard syntax:**
- INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select_statement1 FROM from_statement;
- INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement;
- INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] (z,y) select_statement1 FROM from_statement;
- **Hive extension (multiple inserts):**
- FROM from_statement
  INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select_statement1
  [INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select_statement2]
  [INSERT INTO TABLE tablename2 [PARTITION ...] select_statement2] ...;
- FROM from_statement
  INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1
  [INSERT INTO TABLE tablename2 [PARTITION ...] select_statement2]
  [INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select_statement2] ...;
- **Hive extension (dynamic partition inserts):**
- INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select_statement FROM from_statement;
- INSERT INTO TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select_statement FROM from_statement;

Remarks

insert overwrite
An insert overwrite statement deletes any existing files in the target table or partition before adding new files based off of the select statement used. Note that when there are structure changes to a table or to the DML used to load the table that sometimes the old files are not deleted. When loading to a table using dynamic partitioning only partitions defined by the select statement will be

overwritten. Any preexisting partitions in the target will remain and will not be deleted.

insert into
An insert into statement appends new data into a target table based off of the select statement used.

Examples

insert overwrite

```
insert overwrite table yourTargetTable select * from yourSourceTable;
```

Insert into table

INSERT INTO will append to the table or partition, keeping the existing data intact.

```
    INSERT INTO table yourTargetTable SELECT * FROM yourSourceTable;
```

If a table is partitioned then we can insert into that particular partition in static fashion as shown below.

```
    INSERT INTO TABLE yourTargetTable PARTITION (state=CA, city=LIVERMORE)
    select * FROM yourSourceTable;
```

If a table is partitioned then we can insert into that particular partition in dynamic fashion as shown below. To perfom dynamic partition inserts we must set below below properties.

```
    Dynamic Partition inserts are disabled by default. These are the relevant configuration
properties for dynamic partition inserts:
    SET hive.exec.dynamic.partition=true;
    SET hive.exec.dynamic.partition.mode=non-strict

    INSERT INTO TABLE yourTargetTable PARTITION (state=CA, city=LIVERMORE) (date,time)
    select * FROM yourSourceTable;
```

Multiple Inserts into from a table.

Hive extension (multiple inserts):

```
    FROM table_name

    INSERT OVERWRITE TABLE table_one SELECT table_name.column_one,table_name.column_two

    INSERT OVERWRITE TABLE table_two SELECT table_name.column_two WHERE table_name.column_one
== 'something'
```

Read Insert Statement online: https://riptutorial.com/hive/topic/1744/insert-statement

Chapter 9: SELECT Statement

Syntax

- SELECT [ALL | DISTINCT] select_expr, select_expr, select_expr, ….
- FROM table_reference
- [WHERE where_condition]
- [GROUP BY col_list]
- [HAVING having condition]
- [ORDER BY col_list]
- [LIMIT n]

Examples

Select All Rows

`SELECT` is used to retrieve rows of data from a table. You can specify which columns will be retrieved:

```
SELECT Name, Position
FROM Employees;
```

Or just use * to get all columns:

```
SELECT *
FROM Employees;
```

Select Specific Rows

This query will return all columns from the table `sales` where the values in the column `amount` is greater than 10 and the data in the `region` column in "US".

```
SELECT * FROM sales WHERE amount > 10 AND region = "US"
```

You can use *regular expressions* to select the columns you want to obtain. The following statement will get the data from column `name` and all the columns starting with the prefix `address`.

```
SELECT name, address.* FROM Employees
```

You can also use the keyword `LIKE` (combined with the character '%') to match strings that begin with or end with a particular substring. The following query will return all the rows where the column `city` begins with "New"

```
SELECT name, city FROM Employees WHERE city LIKE 'New%'
```

You can use the keyword `RLIKE` to use Java *regular expressions*. The following query will return rows which column `name` contains the words "smith" or "son".

```
SELECT name, address FROM Employee WHERE name RLIKE '.*(smith|son).*'
```

You can apply functions to the returned data. The following sentence will return all name in upper case.

```
SELECT upper(name) FROM Employees
```

You can use different mathematical functions , collection functions, type conversion functions, date functions, conditional functions or string functions.

In order to limit the number of rows given in result, you can use the `LIMIT` keyword. The following statement will return only ten rows.

```
SELECT * FROM Employees LIMIT 10
```

Select: Project selected columns

Sample table (say Employee) structure

Column Name	Datatype
ID	INT
F_Name	STRING
L_Name	STRING
Phone	STRING
Address	STRING

Project all the columns

Use wild card `*` to project all the columns. e.g.

```
Select * from Employee
```

Project selected columns (say ID, Name)

Use name of columns in the projection list. e.g.

```
Select ID, Name from Employee
```

Discard 1 column from Projection list

Display all columns except 1 column. e.g.

```
Select `(ID)?+.+` from Employee
```

Discard columns matching pattern

Reject all columns which matches the pattern. e.g. Reject all the columns ending with `NAME`

```
Select `(.*NAME$)?+.+` from Employee
```

Read SELECT Statement online: https://riptutorial.com/hive/topic/4133/select-statement

Chapter 10: Table Creation Script with sample data

Examples

Date and Timestamp types

```
CREATE TABLE all_datetime_types(
  c_date date,
  c_timestamp timestamp
);
```

Minimum and maximum data values:

```
insert into all_datetime_types values ('0001-01-01','0001-01-01 00:00:00.000000001');
insert into all_datetime_types values ('9999-12-31','9999-12-31 23:59:59.999999999');
```

Text Types

```
CREATE TABLE all_text_types(
  c_char char(255),
  c_varchar varchar(65535),
  c_string string
);
```

Sample data:

```
insert into all_text_type values ('some ****&&&%%% char value ','some $$$$#####@@@@ varchar
value','some !!~~~++ string value' );
```

Numeric Types

```
CREATE TABLE all_numeric_types(
  c_tinyint tinyint,
  c_smallint smallint,
  c_int int,
  c_bigint bigint,
  c_decimal decimal(38,3)
);
```

Minimum and maximum data values:

```
insert into all_numeric_types values (-128,-32768,-2147483648,-9223372036854775808,-
99999999999999999999999999999999999.999);
insert into all_numeric_types values
(127,32767,2147483647,9223372036854775807,99999999999999999999999999999999999.999);
```

Floating Point Numeric Types

```
CREATE TABLE all_floating_numeric_types(
  c_float float,
  c_double double
);
```

Minimum and maximum data values:

```
insert into all_floating_numeric_types values (-3.4028235E38,-1.7976931348623157E308);
insert into all_floating_numeric_types values (-1.4E-45,-4.9E-324);
insert into all_floating_numeric_types values (1.4E-45,4.9E-324);
insert into all_floating_numeric_types values (3.4028235E38,1.7976931348623157E308);
```

Boolean and Binary Types

```
CREATE TABLE all_binary_types(
  c_boolean boolean,
  c_binary binary
);
```

Sample data:

```
insert into all_binary_types values (0,1234);
insert into all_binary_types values (1,4321);
```

Note:

- For boolean, internally it stored as true or false.
- For binary, it will store base64 encoded value.

Complex Types

ARRAY

```
CREATE TABLE array_data_type(
  c_array array<string>)
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  COLLECTION ITEMS TERMINATED BY '&';
```

Create `data.csv` with data:

```
arr1&arr2
arr2&arr4
```

Put `data.csv` in `/tmp` folderand load this data in Hive

```
LOAD DATA LOCAL INPATH '/tmp/data.csv' INTO TABLE array_data_type;
```

Or you can put this CSV in HDFS say at `/tmp`. Load data from CSV at HDFS using

```
LOAD DATA INPATH '/tmp/data.csv' INTO TABLE array_data_type;
```

MAP

```
CREATE TABLE map_data_type(
  c_map map<int,string>)
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  COLLECTION ITEMS TERMINATED BY '&'
  MAP KEYS TERMINATED BY '#';
```

`data.csv` file:

```
101#map1&102#map2
103#map3&104#map4
```

Load data into hive:

```
LOAD DATA LOCAL INPATH '/tmp/data.csv' INTO TABLE map_data_type;
```

STRUCT

```
CREATE TABLE struct_data_type(
  c_struct struct<c1:smallint,c2:varchar(30)>)
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  COLLECTION ITEMS TERMINATED BY '&';
```

`data.csv` file:

```
101&struct1
102&struct2
```

Load data into hive:

```
LOAD DATA LOCAL INPATH '/tmp/data.csv' INTO TABLE struct_data_type;
```

UNIONTYPE

```
CREATE TABLE uniontype_data_type(
  c_uniontype uniontype<int, double, array<string>)
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  COLLECTION ITEMS TERMINATED BY '&';
```

`data.csv` file:

```
0&10
1&10.23
2&arr1&arr2
```

Load data into hive:

```
LOAD DATA LOCAL INPATH '/tmp/data.csv' INTO TABLE uniontype_data_type;
```

Read Table Creation Script with sample data online: https://riptutorial.com/hive/topic/5067/table-creation-script-with-sample-data

Chapter 11: User Defined Aggregate Functions (UDAF)

Examples

UDAF mean example

- Create a Java class which extends `org.apache.hadoop.hive.ql.exec.hive.UDAF` Create an inner class which implements `UDAFEvaluator`
- Implement five methods
    - `init()` – This method initializes the evaluator and resets its internal state. We are using new Column() in the code below to indicate that no values have been aggregated yet.
    - `iterate()` – This method is called every time there is a new value to be aggregated. The evaluator should update its internal state with the result of performing the aggregation (we are doing sum – see below). We return true to indicate that the input was valid.
    - `terminatePartial()` – This method is called when Hive wants a result for the partial aggregation. The method must return an object that encapsulates the state of the aggregation.
    - `merge()` – This method is called when Hive decides to combine one partial aggregation with another.
    - `terminate()` – This method is called when the final result of the aggregation is needed.

```
public class MeanUDAF extends UDAF {
// Define Logging
static final Log LOG = LogFactory.getLog(MeanUDAF.class.getName());
public static class MeanUDAFEvaluator implements UDAFEvaluator {
/**
 * Use Column class to serialize intermediate computation
 * This is our groupByColumn
 */
public static class Column {
 double sum = 0;
 int count = 0;
 }
private Column col = null;
public MeanUDAFEvaluator() {
 super();
 init();
 }
// A - Initalize evaluator - indicating that no values have been
// aggregated yet.
public void init() {
 LOG.debug("Initialize evaluator");
 col = new Column();
 }
// B- Iterate every time there is a new value to be aggregated
 public boolean iterate(double value) throws HiveException {
```

```
 LOG.debug("Iterating over each value for aggregation");
 if (col == null)
 throw new HiveException("Item is not initialized");
 col.sum = col.sum + value;
 col.count = col.count + 1;
 return true;
 }
// C - Called when Hive wants partially aggregated results.
 public Column terminatePartial() {
 LOG.debug("Return partially aggregated results");
 return col;
 }
 // D - Called when Hive decides to combine one partial aggregation with another
 public boolean merge(Column other) {
 LOG.debug("merging by combining partial aggregation");
 if(other == null) {
 return true;
 }
 col.sum += other.sum;
 col.count += other.count;
 return true;
}
 // E - Called when the final result of the aggregation needed.
 public double terminate(){
 LOG.debug("At the end of last record of the group - returning final result");
 return col.sum/col.count;
 }
 }
}


hive> CREATE TEMPORARY FUNCTION <FUNCTION NAME> AS 'JAR PATH.jar';
hive> select id, mean_udf(amount) from table group by id;
```

Read User Defined Aggregate Functions (UDAF) online:
https://riptutorial.com/hive/topic/5137/user-defined-aggregate-functions--udaf-

Chapter 12: User Defined Table Functions (UDTF's)

Examples

UDTF Example and Usage

User defined table functions represented by **org.apache.hadoop.hive.ql.udf.generic.GenericUDTF** interface. This function allows to output multiple rows and multiple columns for a single input.

We have to overwrite below methods :

```
 1.we specify input and output parameters
abstract StructObjectInspector initialize(ObjectInspector[] args)
                                                throws UDFArgumentException;

 2.we process an input record and write out any resulting records
abstract void process(Object[] record) throws HiveException;

3.function is Called to notify the UDTF that there are no more rows to process.
    Clean up code or additional output can be produced here.
abstract void close() throws HiveException;




import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import
org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;

public class NameParserGenericUDTF extends GenericUDTF {
      private PrimitiveObjectInspector stringOI = null;

       //Defining input argument as string.
      @Override
      public StructObjectInspector initialize(ObjectInspector[] args) throws
UDFArgumentException {
        if (args.length != 1) {
          throw new UDFArgumentException("NameParserGenericUDTF() takes exactly one
argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
```

```
            && ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() !=
PrimitiveObjectInspector.PrimitiveCategory.STRING) {
          throw new UDFArgumentException("NameParserGenericUDTF() takes a string as a
parameter");
        }

        // input
        stringOI = (PrimitiveObjectInspector) args[0];

        // output
        List<String> fieldNames = new ArrayList<String>(2);
        List<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>(2);
        fieldNames.add("name");
        fieldNames.add("surname");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
      }

      public ArrayList<Object[]> processInputRecord(String name){
            ArrayList<Object[]> result = new ArrayList<Object[]>();

            // ignoring null or empty input
            if (name == null || name.isEmpty()) {
              return result;
            }

            String[] tokens = name.split("\\s+");

            if (tokens.length == 2){
                result.add(new Object[] { tokens[0], tokens[1] });
            }else if (tokens.length == 4 && tokens[1].equals("and")){
                result.add(new Object[] { tokens[0], tokens[3] });
                result.add(new Object[] { tokens[2], tokens[3] });
            }

            return result;
      }

      @Override
      public void process(Object[] record) throws HiveException {
        final String name = stringOI.getPrimitiveJavaObject(record[0]).toString();
        ArrayList<Object[]> results = processInputRecord(name);

        Iterator<Object[]> it = results.iterator();

        while (it.hasNext()){
            Object[] r = it.next();
            forward(r);
        }
      }

      @Override
      public void close() throws HiveException {
        // do nothing
      }
}
```

Package the code into jar and need to add jar to hive context.

```
hive> CREATE TEMPORARY FUNCTION process_names as 'jar.path.NameParserGenericUDTF';

Here we will pass input as full name and break it into first and last name.

hive> SELECT
   t.name,
   t.surname
FROM people
    lateral view process_names(name) t as name, surname;

Teena Carter
John Brownewr
```

Read User Defined Table Functions (UDTF's) online: https://riptutorial.com/hive/topic/6502/user-defined-table-functions--udtf-s-

Credits

S. No	Chapters	Contributors
1	Getting started with hive	Bhavesh, Community, franklinsijo, johnnyaug, NeoWelkin
2	Create Database and Table Statement	CodingInCircles, dev ツ, goks, Panther, Venkata Karthik
3	Export Data in Hive	Prem Singh Bist
4	File formats in HIVE	agentv, Alex, Community, johnnyaug, leftjoin, Mzzzzzz, NeoWelkin, tomek, Venkata Karthik
5	Hive Table Creation Through Sqoop	NeoWelkin
6	Hive User Defined Functions (UDF's)	Ashok, Panther, Venkata Karthik
7	Indexing	Prem Singh Bist
8	Insert Statement	Jared, johnnyaug, Venkata Karthik
9	SELECT Statement	Ambrish, dev, Jaime Caffarel, johnnyaug
10	Table Creation Script with sample data	dev ツ
11	User Defined Aggregate Functions (UDAF)	dev ツ, Venkata Karthik
12	User Defined Table Functions (UDTF's)	Venkata Karthik