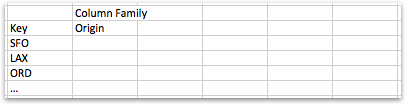
At the time of HBase table definition, you specify one or more “column families”. These are group headers for columns you might add earlier, and in the case of my origin airport table I might just use the column family name “dest”, so that the HBase table DDL looks like this:

Create 'geog\_origin','origin'

and the conceptual view of the table would look like this:



Now what’s neat about NoSQL-style databases like this (and Endeca Server is the same) is that you can define individual columns just by using them. For example, I could create columns for the airport name, airport city, airport state and airport code just by using their name in a data load, prefixing those column names with the named of a previously-defined column family. Using the HBase Shell, for example, I could issue the following PUT commands to insert the first row of data into this HBase table, like this:

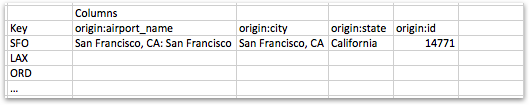
put 'geog\_origin’,’SFO','origin:airport\_name','San Francisco, CA: San Francisco'

put 'geog\_origin’,’SFO','origin:city’,’San Francisco, CA'

put 'geog\_origin’,’SFO',’origin':state','California'

put 'geog\_origin’,'SFO',’origin':id’,'14771'

Now my HBase table conceptually looks like this:



If I then want to use another column under the “origin” column family for LAX, I can just do so by using it in the next set of PUT commands, like this:

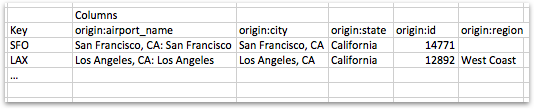
put 'geog\_origin','LAX’,origin:airport\_name','Los Angeles, CA: Los Angeles'

put 'geog\_origin','LAX','origin:city','Los Angeles, CA'

put 'geog\_origin','LAX','origin:state','California'

put 'geog\_origin','LAX','origin:region’,’West Coast'

put 'geog\_origin','LAX','origin:id','12892'



Each column within column families has its values individually set, retrieved and deleted using PUT, GET

20down voteaccepted

I like using Apache Pig for ingest into HBase because it is simple, straightforward, and flexible.

Here is a Pig script that would do the job for you, after you have created the table and the column family. To create the table and the column family, you'll do:

$ hbase shell

> create 'mydata', 'mycf'

Move the file to HDFS:

$ hadoop fs -put /home/file.txt /user/surendhar/file.txt

Then, write the pig script to store with [HBaseStorage](http://pig.apache.org/docs/r0.9.1/api/org/apache/pig/backend/hadoop/hbase/HBaseStorage.html) (you may have to look up how to [set up and run Pig](http://pig.apache.org/docs/r0.9.1/start.html)):

A = LOAD 'file.txt' USING PigStorage(',') as (strdata:chararray, intdata:long);

STORE A INTO 'hbase://mydata'

USING org.apache.pig.backend.hadoop.hbase.HBaseStorage( 'mycf:intdata');

Note that in the above script, the key is going to be strdata. If you want to create your own key from something, use a [FOREACH](http://pig.apache.org/docs/r0.9.1/basic.html#foreach) statement to generate the key. HBaseStorage assumes that the first thing in the previous relation (A::strdata in this case) is the key.

--

#### Short Description:

This is a quick tutorial on how to load data into HBase tables using Importtsv command

#### Article

One of the first cases we get to see with Hbase is loading it up with Data, most of the time we will have some sort of data in some format like CSV availalble and we would like to load it in Hbase, lets take a quick look on how does the procedure looks like:

lets examine our example data by looking at the simple structure that I have got for an industrial sensor

1. id, temp:in,temp:out,vibration,pressure:in,pressure:out
2. 5842, 50, 30, 4, 240, 340

First of all make sure Hbase is started on your Sandbox as following

**Creating the HBase Table**

* Login as Root to the HDP Sandbox and and switch to the Hbase User

1. root> su - hbase

* Go to the Hbase Shell by typing

1. hbase> hbase shell

* Create the example table by typing

1. hbase(main):001:0> create 'sensor','temp','vibration','pressure'

* lets make sure the table was created and examine the structure by typing

1. hbase(main):001:0> list

* now, exit the shell by typing 'exit' and lets load some data

**Loading the Data**

* lets put the hbase.csv file in HDFS, you may SCP it first to the cluster by using the following command

1. macbook-ned> scp hbase.csv root@sandbox.hortonworks.com:/home/hbase

* now put in HDFS using the following command

1. hbase> hadoop dfs -copyFromLocal hbase.csv /tmp

* we shall now execute the Loadtsv statement as following

1. hbase> hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -Dimporttsv.separator=, -Dimporttsv.columns="HBASE\_ROW\_KEY,id,temp:in,temp:out,vibration,pressure:in,pressure:out" sensor hdfs://sandbox.hortonworks.com:/tmp/hbase.csv

* once the mapreduce job is completed, return back to hbase shell and execute

1. hbase(main):001:0> scan sensor

* you should now see the data in the table

Remarks

* Importtsv statement generates massive amount of logs, so make sure you have enough space in /var/logs, its always better to have it mounted on a seperate directories in real cluster to avoid operational stop becuase of logs filling the partition.

HBase – External Table

**HBase-Hive Integration:**

Creating an external table in hive for HBase table allows you to query HBase data o be queried in Hive without the need for duplicating data. You can just update or delete data from HBase table and you can view the modified table in Hive too.

**Example:**

Consider you have an hbase table with columns id, name and email.

Sample external table command for hive:

CREATE EXTERNAL TABLE hivehbasetable(key INT, id INT, username STRING, password STRING, email STRING) STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,id:id,name:username,name:password,email:email") TBLPROPERTIES("hbase.table.name" = "hbasetable");

**Hive: Internal Tables**

There are 2 types of tables in Hive, Internal and External. This case study describes creation of internal table, loading data in it, creating views, indexes and dropping table on weather data.

**Creating Internal Table**

Internal table are like normal database table where data can be stored and queried on. On dropping these tables the data stored in them also gets deleted and data is lost forever. So one should be careful while using internal tables as one drop command can destroy the whole data. Open new terminal and fire up hive by just typing hive. Create table on weather data.

CREATE TABLE weather (wban INT, date STRING, precip INT)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘,’

LOCATION ‘ /hive/data/weather’;

ROW FORMAT should have delimiters used to terminate the fields and lines like in the above example the fields are terminated with comma (“,”). The default location of Hive table is overwritten by using LOCATION. So the data now is stored in data/weather folder inside hive.

### ****Load the Data in Table****

Data can be loaded in 2 ways in Hive either from local file or from HDFS to Hive. To load the data from local to Hive use the following command in **NEW terminal**:

hadoop fs –copyFromLocal /home/user/data/weather/2012.txt hdfs:*//hname:10001/hive/data/weather*

Here the hdfs path was initially made in the create statement using LOCATION ‘ /hive/data/weather’.

Another way to load data is to load it from HDFS to hive using the following command:

LOAD DATA INPATH ‘hdfs:/data/2012.txt’ INTO TABLE weather;

### ****Views****

Views are used for creating virtual tables. They are faster than creating actual tables and they can work as table while using them in any other query. For example precipitation view can be made as follows and can be used in other query just like a table is used.

CREATE VIEW AS precipitation\_data

SELECT \* FROM weatherext WHERE precip IS NOT NULL AND precip>0;

Following query can be used to retrieve data from precipitation\_data

SELECT \* FROM precipitation\_data;

### ****Indexing****

Indexes are made on top of tables so that they speed up queries. Most popular column that are used very often in WHERE clause should be indexed to make the query run faster. Partition can be built on weather table’s date column in following way:

CREATE INDEX date\_index ON TABLE weather (date) AS ‘COMPACT’ WITH REBUILD;

After making this index any query that uses date column of weather table will be faster than running it before creating index.

**Drop table**

On dropping the table loaded by second method that is from HDFS to Hive, the data gets deleted and there is no copy of data on HDFS. This means that on creating internal table the data gets moved from HDFS to Hive. Table can be dropped using:

DROP TABLE weather;

**Hive: External Tables**

**Creating external table**

Open new terminal and fire up hive by just typing hive. Create table on weather data.

CREATE EXTERNAL TABLE weatherext ( wban INT, date STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘,’

LOCATION ‘ /hive/data/weatherext’;

ROW FORMAT should have delimiters used to terminate the fields and lines like in the above example the fields are terminated with comma (“,”). The default location of Hive table is overwritten by using LOCATION. So the data now is stored in data/weatherext folder inside hive.

**Load the data in table**

Load the data from HDFS to Hive using the following command:

LOAD DATA INPATH ‘hdfs:/data/2012.txt’ INTO TABLE weatherext;

**Partitioning of table**

Hive stores tables in partitions. Partitions are used to divide the table into related parts. Partitions make data querying more efficient. For example in the above weather table the data can be partitioned on the basis of year and month and when query is fired on weather table this partition can be used as one of the column.

CREATE EXTERNAL TABLE IF NOT EXSISTS weatherext ( wban INT, date STRING)

PARTITIONED BY (year INT, month STRING)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ‘,’

LOCATION ‘ /hive/data/weatherext’;

Loading data in partitioned tables is different than non-partitioned one. There is little manual work of mentioning the partition data. Data can be loaded in partition, year 2012 and month 01 and 02 as follows:

LOAD DATA INPATH ‘hdfs:/data/2012.txt’ INTO TABLE weatherext PARTITION (year=2012, month=’01’);

LOAD DATA INPATH ‘hdfs:/data/2012.txt’ INTO TABLE weatherext PARTITION (year=2012, month=’02’);

This creates the partitioned table and makes different folder for each partition which helps in querying data.

**Querying of partitioned table**

Partitioned tables can use partition parameters as one of the column for querying. To retrieve all the data for month of ‘02’ following query can be used on weather table.

SELECT \* FROM weatherext WHERE month = ‘02’;

**Drop table**

On dropping the external table, the data does not get deleted from HDFS. Thus it is evident that the external table are just pointers on HDFS data. Table can be dropped using:

DROP TABLE weather;

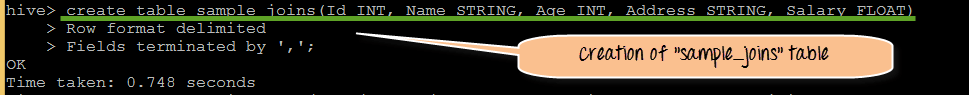
# Hive Join & SubQuery Tutorial with Examples

## Join queries:

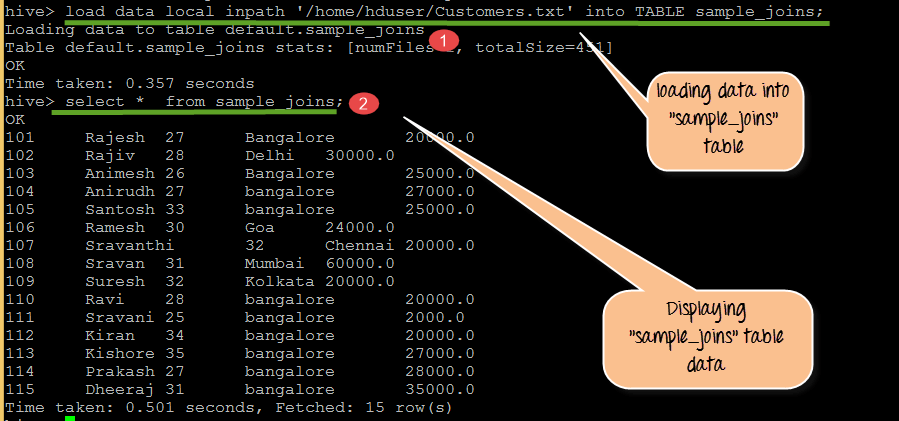
Join queries can perform on two tables present in Hive. For understanding Join Concepts in clear here we are creating two tables overhere,

* Sample\_joins( Related to Customers Details )
* Sample\_joins1( Related to orders details done by Employees)

**Step 1)**Creation of table "sample\_joins" with Column names ID, Name, Age, address and salary of the employees

[](https://www.guru99.com/images/Hive/120415_1305_HiveQueries7.png)

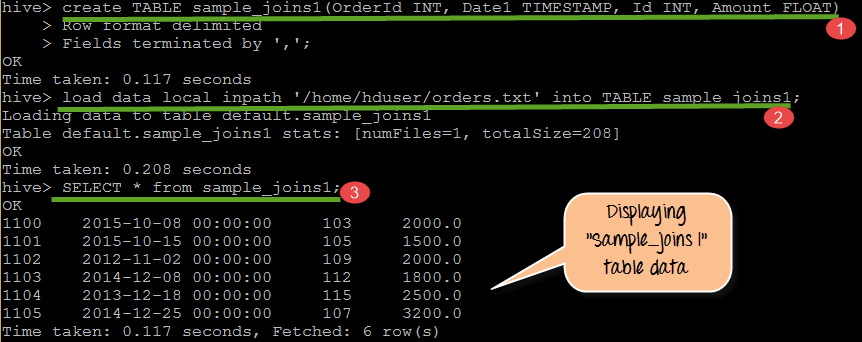
**Step 2)**Loading and Displaying Data

sample\_joins [](https://www.guru99.com/images/Hive/120415_1305_HiveQueries8.png)

From the above screen shot

1. Loading data into sample\_joins from Customers.txt
2. Displaying table contents

**Step 3)**Creation of sample\_joins1 table and loading, displaying data

[](https://www.guru99.com/images/Hive/120415_1305_HiveQueries9.png)

From the above screenshot, we can observe the following

1. Creation of table sample\_joins1 with columns Orderid, Date1, Id, Amount
2. Loading data into sample\_joins1 from orders.txt
3. Displaying records present in sample\_joins1

Moving forward we will see different types of joins that can be performed on tables we have created but before that you have to consider following points for joins.

**Some points to observe in Joins:**

* Only Equality joins are allowed In Joins
* More than two tables can be joined in the same query
* LEFT, RIGHT, FULL OUTER joins exist in order to provide more control over ON Clause for which there is no match
* Joins are not Commutative
* Joins are left-associative irrespective of whether they are LEFT or RIGHT joins

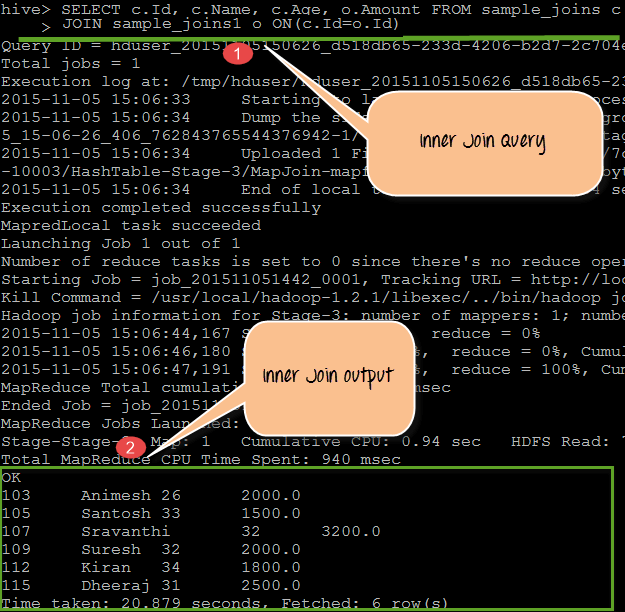
## Different type of joins

Joins are of 4 types, these are

* Inner join
* Left outer Join
* Right Outer Join
* Full Outer Join

**Inner Join:**

The Records common to the both tables will be retrieved by this Inner Join.

[](https://www.guru99.com/images/Hive/120415_1305_HiveQueries10.png)

From the above screenshot, we can observe the following

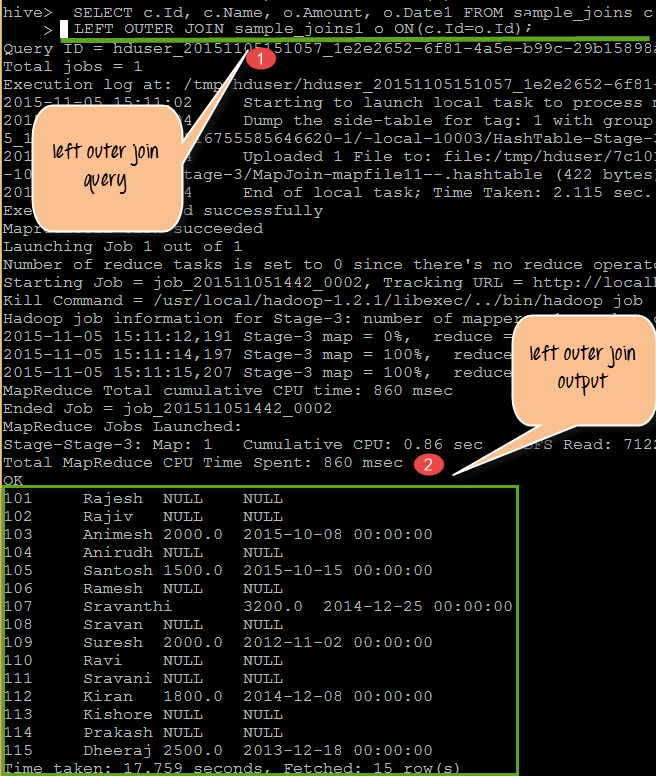
1. Here we are performing join query using JOIN keyword between the tables sample\_joins and sample\_joins1 with matching condition as (c.Id= o.Id).
2. The output displaying common records present in both the table by checking the condition mentioned in the query

Query:

SELECT c.Id, c.Name, c.Age, o.Amount FROM sample\_joins c JOIN sample\_joins1 o ON(c.Id=o.Id);

**Left Outer Join:**

* Hive query language LEFT OUTER JOIN returns all the rows from the left table even though there are no matches in right table
* If ON Clause matches zero records in the right table, the joins still return a record in the result with NULL in each column from the right table

[](https://www.guru99.com/images/Hive/120415_1305_HiveQueries11.png)

From the above screenshot, we can observe the following

1. Here we are performing join query using "LEFT OUTER JOIN" keyword between the tables sample\_joins and sample\_joins1 with matching condition as (c.Id= o.Id).

**For example** here we are using employee id as a reference, it checks whether id is common in right as well as left the table or not. It acts as matching condition.

1. The output displaying common records present in both the table by checking the condition mentioned in the query.

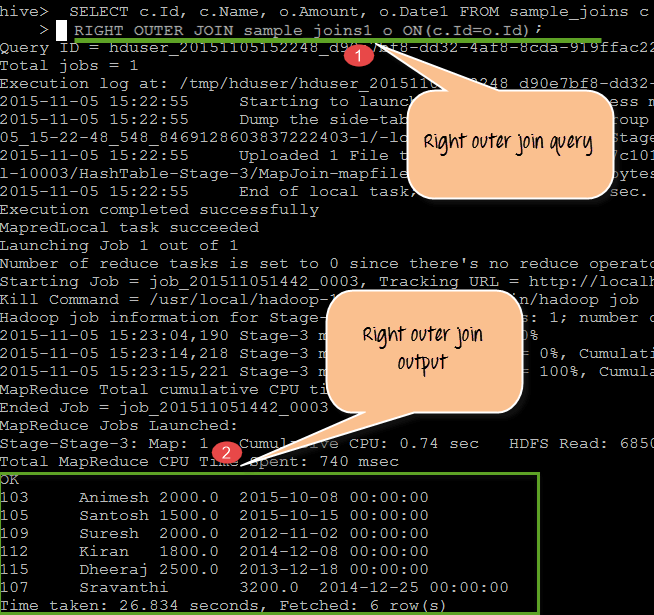
NULL values in the above output are columns with no values from Right table that is sample\_joins1

Query:

SELECT c.Id, c.Name, o.Amount, o.Date1 FROM sample\_joins c LEFT OUTER JOIN sample\_joins1 o ON(c.Id=o.Id)

**Right outer Join:**

* Hive query language RIGHT OUTER JOIN returns all the rows from the Right table even though there are no matches in left table
* If ON Clause matches zero records in the left table, the joins still return a record in the result with NULL in each column from the left table
* RIGHT joins always return records from a Right table and matched records from the left table. If the left table is having no values corresponding to the column, it will return NULL values in that place.

[](https://www.guru99.com/images/Hive/120415_1305_HiveQueries12.png)

From the above screenshot, we can observe the following

1. Here we are performing join query using "RIGHT OUTER JOIN" keyword between the tables sample\_joins and sample\_joins1 with matching condition as (c.Id= o.Id).
2. The output displaying common records present in both the table by checking the condition mentioned in the query

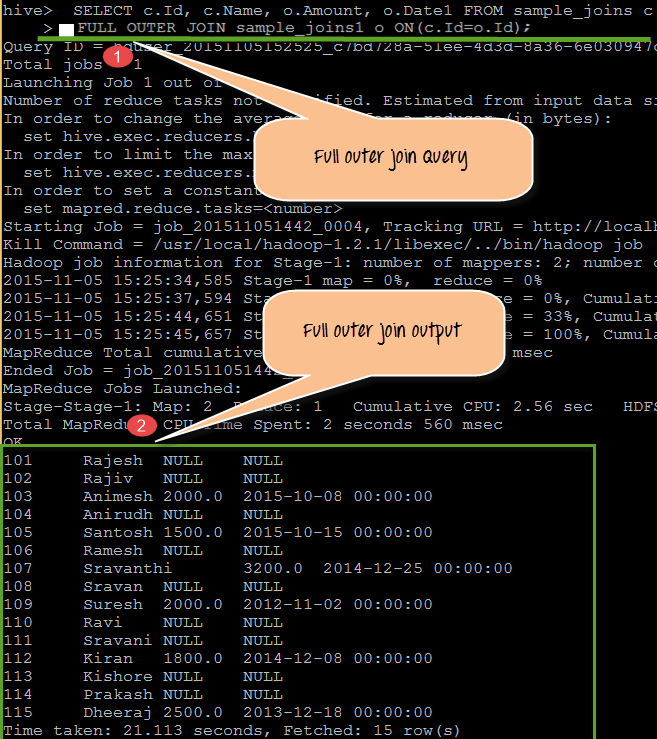
Query**:**

SELECT c.Id, c.Name, o.Amount, o.Date1 FROM sample\_joins c RIGHT OUTER JOIN sample\_joins1 o ON(c.Id=o.Id)

**Full outer join:**

It combines records of both the tables sample\_joins and sample\_joins1 based on the JOIN Condition given in query.

It returns all the records from both tables and fills in NULL Values for the columns missing values matched on either side.

[](https://www.guru99.com/images/Hive/120415_1305_HiveQueries13.png)

From the above screen shot we can observe the following:

1. Here we are performing join query using "FULL OUTER JOIN" keyword between the tables sample\_joins and sample\_joins1 with matching condition as (c.Id= o.Id).
2. The output displaying all the records present in both the table by checking the condition mentioned in the query. Null values in output here indicates the missing values from the columns of both tables.

Query

SELECT c.Id, c.Name, o.Amount, o.Date1 FROM sample\_joins c FULL OUTER JOIN sample\_joins1 o ON(c.Id=o.Id)

## Sub queries:

A Query present within a Query is known as a sub query. The main query will depend on the values returned by the subqueries.

Subqueries can be classified into two types

* Subqueries in FROM clause
* Subqueries in WHERE clause

**When to use:**

* To get a particular value combined from two column values from different tables
* Dependency of one table values on other tables
* Comparative checking of one column values from other tables

**Syntax:**

Subquery in FROM clause

SELECT <column names 1, 2…n>From (SubQuery) <TableName\_Main >

Subquery in WHERE clause

SELECT <column names 1, 2…n> From<TableName\_Main>WHERE col1 IN (SubQuery);

**Example:**

SELECT col1 FROM (SELECT a+b AS col1 FROM t1) t2

Here t1 and t2 are table names. The colored one is Subquery performed on table t1. Here a and b are columns that are added in a subquery and assigned to col1. Col1 is the column value present in Main table. This column "col1" present in the subquery is equivalent to the main table query in column col1.

## Embedding custom scripts:

Hive provides feasibility of writing user specific scripts for the client requirements. The users can able to write their own map and reduce scripts for the requirements. These are called Embedded Custom scripts. The coding logic is defined in the custom scripts and we can use that script in the ETL time.

**When to choose Embedded Scripts:**

* In client specific requirements developers has to write and deploy scripts in Hive
* Where Hive inbuilt functions are not going to work for specific domain requirements

For this in Hive it uses TRANSFORM clause to embedded both map and reducer scripts.

In this Embedded custom scripts, we have to observe the following points

* Columns will be transformed to string and delimited by TAB before giving it to the user script
* Standard output of the user script will be treated as TAB- separated string columns

Sample Embedded Script,

FROM (

FROM pv\_users

MAP pv\_users.userid, pv\_users.date

USING 'map\_script'

AS dt, uid

CLUSTER BY dt) map\_output

INSERT OVERWRITE TABLE pv\_users\_reduced

REDUCE map\_output.dt, map\_output.uid

USING 'reduce\_script'

AS date, count;

From the above script, we can observe the following

This is only the sample script for understanding

* pv\_users is the users table which is having fields like userid and date as mentioned in map\_script
* Reducer script defined on date and count of the pv\_users tables