1)STATIC AND DYNAMIC PARTIONING

Partition is  horizontally  dividing the data into number of slice in a equal and manageable manner.Every partition is stored as directory within data warehouse table.

Hive partition it supported for Multiple columns in a table .In Hive we can apply Hive Partition concept on Managed tables and External tables.If we not crated dynamic partition for hive, Hive also creates an automatic partition scheme when the table is created

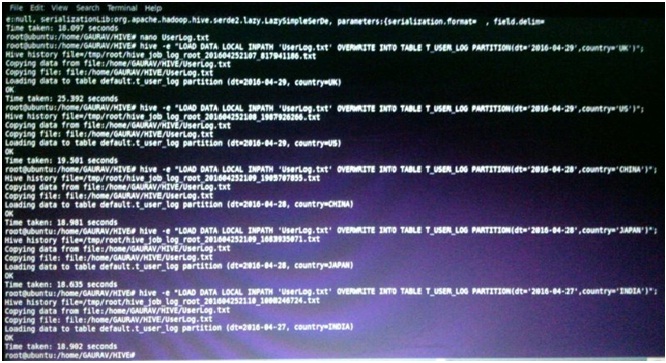
Example

create table cityreport(cityid string,creport string, ctover string)  
partitioned by (city string)  
row format delimited  
fields terminated by ‘|’  
stored as textfile;

Static Partitioning: In Static Partition, we know the partition column before itself. So far so good, now when we load data there it makes the difference.

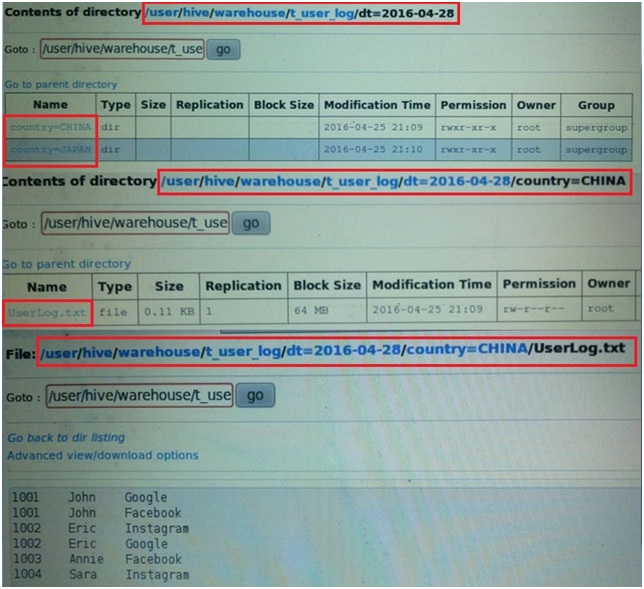
LOAD DATA LOCAL INPATH [path\_name] OVERWRITE INTO TABLE [table\_name] PARTITION(partition\_column='value'....). Here we have to give the partition column value explicitly whenever we want to create new partition as shown below:

LOAD SCRIPTS.

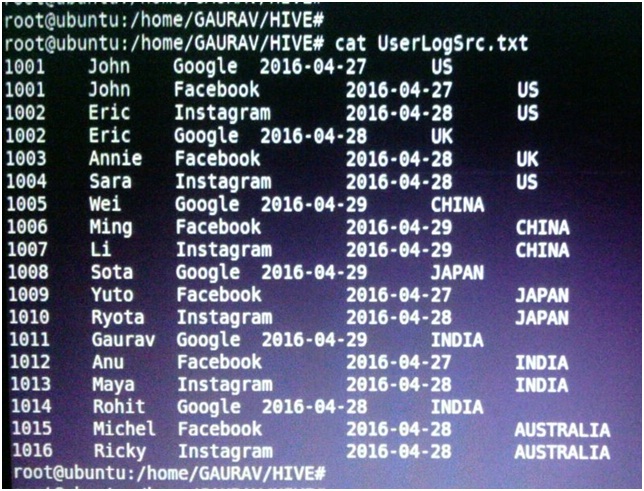


OUTPUT: Since our table T\_USER\_LOG is managed table so the data is loaded in hive warehouse path i.e /user/hive/warehouse/t\_user\_log.

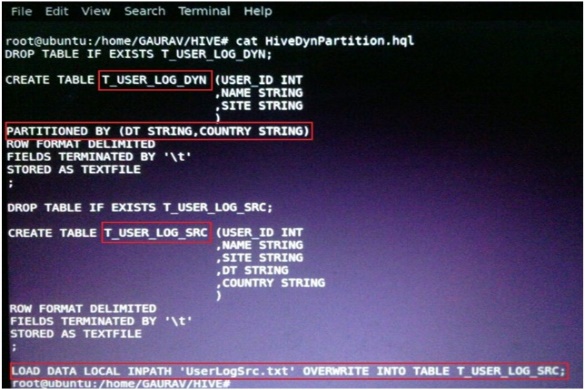




2) DYNAMIC PARTITIONING: Let us see now the load script of Dynamic Partitioning. We will create new table T\_USER\_LOG\_DYN for dynamic partition and also as we told earlier that we will load this table using a new table, let's create another table T\_USER\_LOG\_SRC. Below is the data of source table:



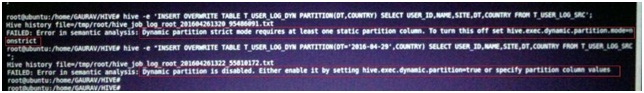
Let's us check the hive script for table DDL.



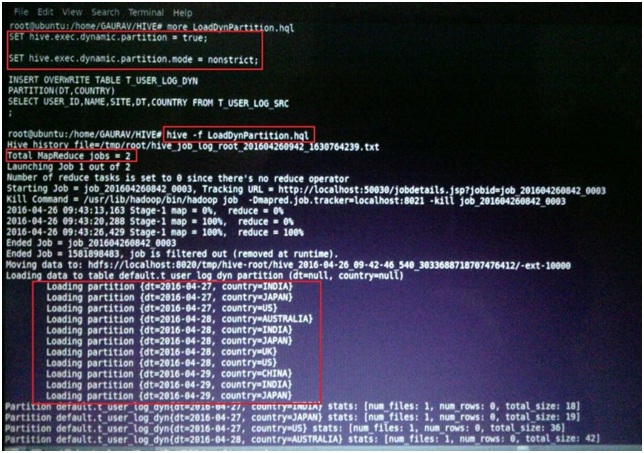
We will see first the negative scenario and then the final load scripts. We have to set two hive properties as below:

1.SET hive.exec.dynamic.partition= true;

2. SET hive.exec.dynamic.partition.mode= nonstrict

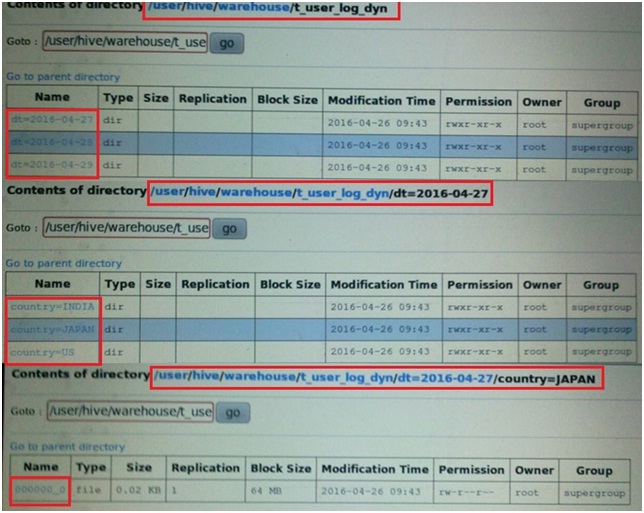


LOAD SCRIPT: Below is the screenshot for loading the table T\_USER\_LOG\_DYN using the data from T\_USER\_LOG\_SRC and creating dynamic partitions. LoadDynPartition.hql is my script to load the table dynamically as shown below:



hive -f LoadDynPartition.hql is used to execute the hive script.

OUTPUT: Once the script is run, the file will be created as below:



2) PARTITIONING AND BUCKETING

Partition is  horizontally  dividing the data into number of slice in a equal and manageable manner.Every partition is stored as directory within data warehouse table.

Hive partition it supported for Multiple columns in a table .In Hive we can apply Hive Partition concept on Managed tables and External tables.If we not crated dynamic partition for hive, Hive also creates an automatic partition scheme when the table is created

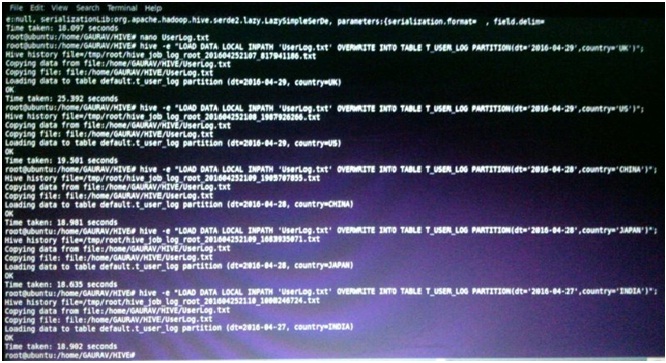
Example Hive Partition

create table cityreport(cityid string,creport string, ctover string)  
partitioned by (city string)  
row format delimited  
fields terminated by ‘|’  
stored as textfile;

Static Partitioning: In Static Partition, we know the partition column before itself. So far so good, now when we load data there it makes the difference.

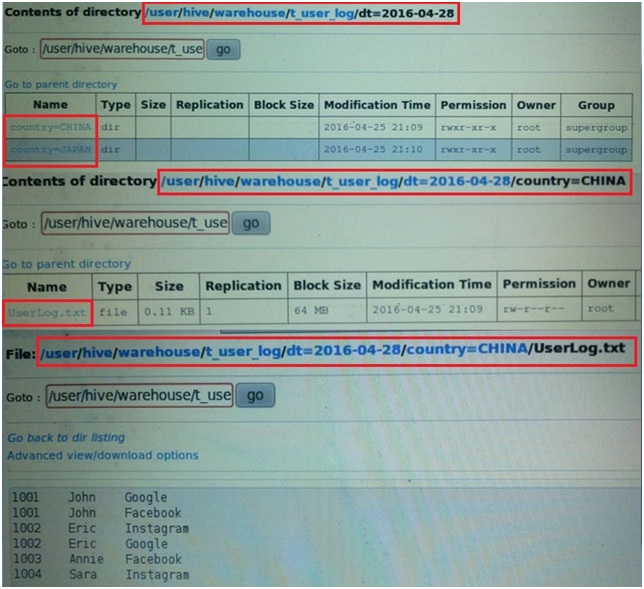
LOAD DATA LOCAL INPATH [path\_name] OVERWRITE INTO TABLE [table\_name] PARTITION(partition\_column='value'....). Here we have to give the partition column value explicitly whenever we want to create new partition as shown below:

LOAD SCRIPTS.

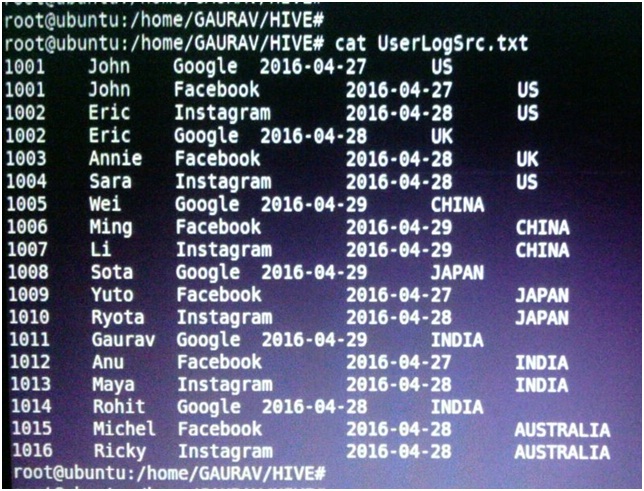


OUTPUT: Since our table T\_USER\_LOG is managed table so the data is loaded in hive warehouse path i.e /user/hive/warehouse/t\_user\_log.

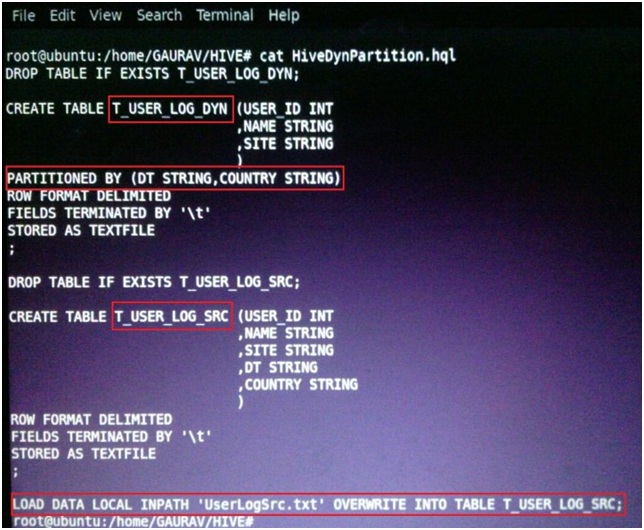




DYNAMIC PARTITIONING: Let us see now the load script of Dynamic Partitioning. We will create new table T\_USER\_LOG\_DYN for dynamic partition and also as we told earlier that we will load this table using a new table, let's create another table T\_USER\_LOG\_SRC. Below is the data of sourse table:



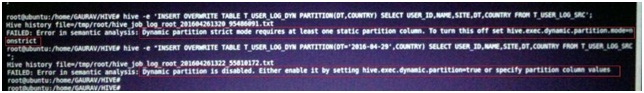
Let's us check the hive script for table DDL.



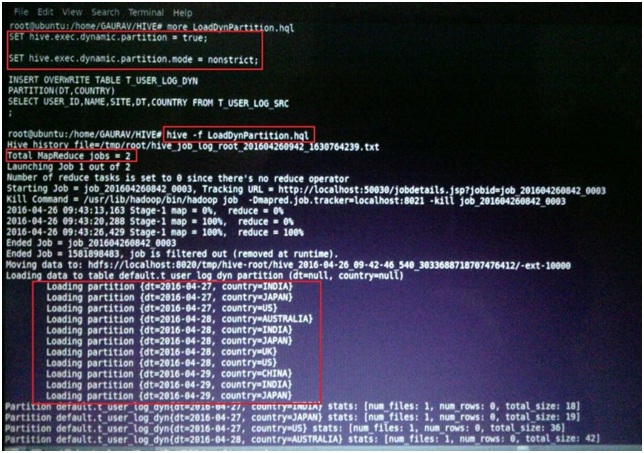
We will see first the negative scenario and then the final load scripts. We have to set two hive properties as below:

1.SET hive.exec.dynamic.partition= true;

2. SET hive.exec.dynamic.partition.mode= nonstrict

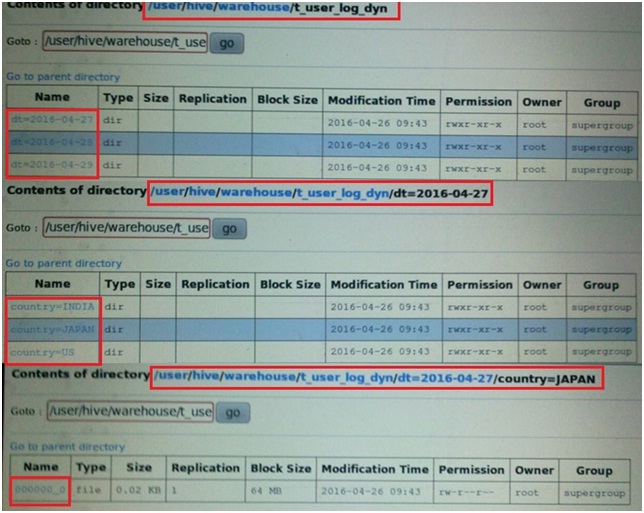


LOAD SCRIPT: Below is the screenshot for loading the table T\_USER\_LOG\_DYN using the data from T\_USER\_LOG\_SRC and creating dynamic partitions. LoadDynPartition.hql is my script to load the table dynamically as shown below:



hive -f LoadDynPartition.hql is used to execute the hive script.

OUTPUT: Once the script is run, the file will be created as below:



## Advantages with Hive Partition

* Distribute execution load horizontally
* Faster execution of queries in case of partition with low volume of data. e.g. Get the population from “**Vatican city**” returns very fast instead of searching entire population of world.
* No need to search entire table columns for a single record.

### Disadvantages with Hive Partition

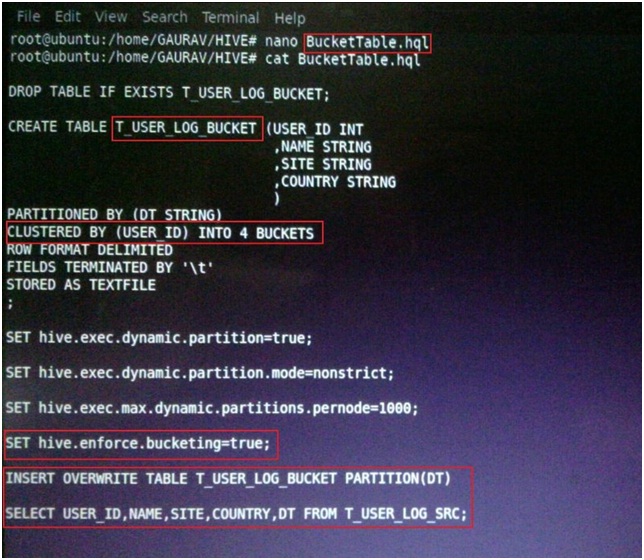
* there is a possibility for creating too many folders in HDFS that is extra burden for Namenode metadata.

Effective for low volume data for a given partition. But some queries like group by on high volume of data still take long time to execute. e.g. Grouping of population of China will take long time compared to grouping of population in Vatican city. Partition is not solving responsiveness problem in case of data skewing towards a particular partition value.

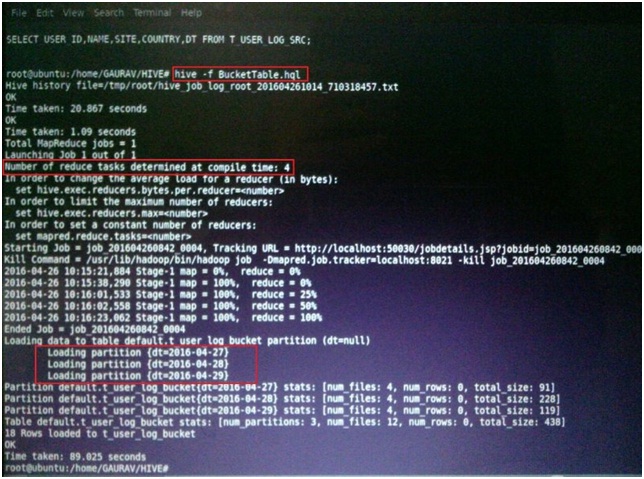
* so there is no guarantee for query optimization for all the times.

BUCKETING in HIVE: When we write data in bucketed table in hive, it places the data in distinct buckets as files. Hive uses some hashing algorithm to generate a number in range of 1 to N buckets [as mentioned in DDL] and based on the result of hashing, data is placed in a particular buckets as a file. Let's create a hive bucketed table T\_USER\_LOG\_BUCKET with a partition column as DT and having 4 buckets. We specify bucketing column in CLUSTERED BY (column\_name) clause in hive table DDL as shown below in hive script files:

LOAD SCRIPT:

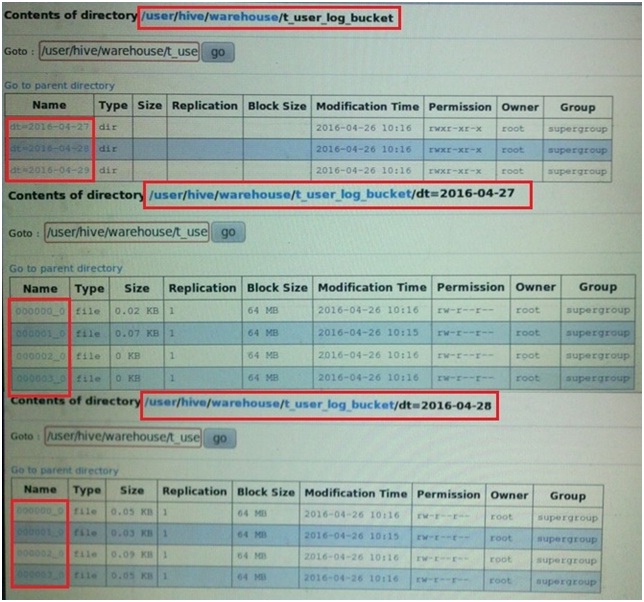


Below screenshots shows the hive script execution to create and load Bucketed table:



As we know that the number of reduce task determine the number of output file so here we have defined table as four buckets so the number of reduce taks is 4 as highlighted in above screenshot.

OUTPUT: Here the output is partitioned on DT column and each partition will contain the four buckets as files. So all three partitions will have four files each as shown in below screenshots



Advantage of Bucketing:

Sampling: When we want to test a table which has huge amount of data or when we want to draw some patterns or when we want some aggregations [where accuracy is not out top priority] then we need to sample i.e we need to run the query on smaller set of data of the entire table data with evenly distributed sample. There are two types of sampling:

1.Bucket Sampling : e.g SELECT \* FROM T\_USER\_LOG\_BUCKET TABLESAMPLE (BUCKET 1 OUT OF 4 AT USER\_ID).... It will select the data from the first buckets of each partition from T\_USER\_LOG\_BUCKET table otherwise in normal select we can't select data in such distributed and evenly manner.

 2.Block Sampling: e.g SELECT \* FROM T\_USER\_LOG\_BUCKET TABLESAMPLE (20 PERCENT);

 Optimize Join Operation: We can optimize the join by bucketing the similar column values in one bucket so that during bucket to bucket join, hive can minimize the processing steps and reduce the data needed to parse and compare for join.

## Advantages with Hive Bucketing

* Due to equal volumes of data in each partition, joins at Map side will be quicker.
* Faster query response like partitioning

### Disadvantages with Hive Bucketing

* You can define number of buckets during table creation but loading of equal volume of data has to be done manually by programmers.

3) GROUPING IN HIVE

**JOINS**

JOIN is a clause that is used for combining specific fields from two tables by using values common to each one. It is used to combine records from two or more tables in the database. It is more or less similar to SQL JOIN.

Syntax

join\_table:

table\_reference JOIN table\_factor [join\_condition]

| table\_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table\_reference

join\_condition

| table\_reference LEFT SEMI JOIN table\_reference join\_condition

| table\_reference CROSS JOIN table\_reference [join\_condition]

Example

We will use the following two tables in this chapter. Consider the following table named CUSTOMERS..

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Consider another table ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

There are different types of joins given as follows:

JOIN

LEFT OUTER JOIN

RIGHT OUTER JOIN

FULL OUTER JOIN

JOIN

JOIN clause is used to combine and retrieve the records from multiple tables. JOIN is same as OUTER JOIN in SQL. A JOIN condition is to be raised using the primary keys and foreign keys of the tables.

The following query executes JOIN on the CUSTOMER and ORDER tables, and retrieves the records:

hive> SELECT c.ID, c.NAME, c.AGE, o.AMOUNT

FROM CUSTOMERS c JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

LEFT OUTER JOIN

The HiveQL LEFT OUTER JOIN returns all the rows from the left table, even if there are no matches in the right table. This means, if the ON clause matches 0 (zero) records in the right table, the JOIN still returns a row in the result, but with NULL in each column from the right table.

A LEFT JOIN returns all the values from the left table, plus the matched values from the right table, or NULL in case of no matching JOIN predicate.

The following query demonstrates LEFT OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

LEFT OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

RIGHT OUTER JOIN

The HiveQL RIGHT OUTER JOIN returns all the rows from the right table, even if there are no matches in the left table. If the ON clause matches 0 (zero) records in the left table, the JOIN still returns a row in the result, but with NULL in each column from the left table.

A RIGHT JOIN returns all the values from the right table, plus the matched values from the left table, or NULL in case of no matching join predicate.

The following query demonstrates RIGHT OUTER JOIN between the CUSTOMER and ORDER tables.

notranslate"> hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE FROM CUSTOMERS c RIGHT OUTER JOIN ORDERS o ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

FULL OUTER JOIN

The HiveQL FULL OUTER JOIN combines the records of both the left and the right outer tables that fulfil the JOIN condition. The joined table contains either all the records from both the tables, or fills in NULL values for missing matches on either side.

The following query demonstrates FULL OUTER JOIN between CUSTOMER and ORDER tables:

hive> SELECT c.ID, c.NAME, o.AMOUNT, o.DATE

FROM CUSTOMERS c

FULL OUTER JOIN ORDERS o

ON (c.ID = o.CUSTOMER\_ID);

On successful execution of the query, you get to see the following response:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

**GROUP BY CLAUSE**

The GROUP BY clause is used to group all the records in a result set using a particular collection column. It is used to query a group of records.

Syntax

The syntax of GROUP BY clause is as follows:

SELECT [ALL | DISTINCT] select\_expr, select\_expr, ...

FROM table\_reference

[WHERE where\_condition]

[GROUP BY col\_list]

[HAVING having\_condition]

[ORDER BY col\_list]]

[LIMIT number];

Example

Let us take an example of SELECT…GROUP BY clause. Assume employee table as given below, with Id, Name, Salary, Designation, and Dept fields. Generate a query to retrieve the number of employees in each department.

+------+--------------+-------------+-------------------+--------+

| ID | Name | Salary | Designation | Dept |

+------+--------------+-------------+-------------------+--------+

|1201 | Gopal | 45000 | Technical manager | TP |

|1202 | Manisha | 45000 | Proofreader | PR |

|1203 | Masthanvali | 40000 | Technical writer | TP |

|1204 | Krian | 45000 | Proofreader | PR |

|1205 | Kranthi | 30000 | Op Admin | Admin |

+------+--------------+-------------+-------------------+--------+

The following query retrieves the employee details using the above scenario.

hive> SELECT Dept,count(\*) FROM employee GROUP BY DEPT;

On successful execution of the query, you get to see the following response:

+------+--------------+

| Dept | Count(\*) |

+------+--------------+

|Admin | 1 |

|PR | 2 |

|TP | 3 |

+------+--------------+

JDBC Program

Given below is the JDBC program to apply the Group By clause for the given example.

import java.sql.SQLException;

import java.sql.Connection;

import java.sql.ResultSet;

import java.sql.Statement;

import java.sql.DriverManager;

public class HiveQLGroupBy {

private static String driverName = "org.apache.hadoop.hive.jdbc.HiveDriver";

public static void main(String[] args) throws SQLException {

// Register driver and create driver instance

Class.forName(driverName);

// get connection

Connection con = DriverManager.

getConnection("jdbc:hive://localhost:10000/userdb", "", "");

// create statement

Statement stmt = con.createStatement();

// execute statement

Resultset res = stmt.executeQuery(“SELECT Dept,count(\*) ” + “FROM employee GROUP BY DEPT; ”);

System.out.println(" Dept \t count(\*)");

while (res.next()) {

System.out.println(res.getString(1) + " " + res.getInt(2));

}

con.close();

}

}

Save the program in a file named HiveQLGroupBy.java. Use the following commands to compile and execute this program.

$ javac HiveQLGroupBy.java

$ java HiveQLGroupBy

Output:

Dept Count(\*)

Admin 1

PR 2

TP 3