**Hadoop cluster setup**

There are 3 clusters available in Hadoop:-

**1) Development cluster**

Total = 8 nodes

* Each Node has 32 cores => 8\*32 = 256 cores allowcated cores = 256-30 = 226 cores
* Each Node has 512GB RAM = 8\*512 = 4096GB => 4TB allowcated RAM =2.10 TB
* Each Node has 5TB Hard-Disk = 8\*5 = 40TB => allowcated Hard-Disk (I.e., HDFS) = 30TB

**2) QA (Quality Analysis) cluster**

In this cluster user acceptance test will be done => in this data modification will be done

Total = 18 nodes

* Each Node has 32 cores => 18\*32 = 576 cores, allowcated cores => 428 cores
* Each Node has 512GB RAM=18\*512=9216GB => 9.2TB, allowcated RAM = 4.5 TB
* Each Node has 5TB Hard-Disk =18\*5 = 90TB => allowcated Hard-Disk (I.e., HDFS)= 70TB

**3) Production cluster**

Total = 22 nodes

* Each Node has 32 cores => 22\*32 = 704 cores, allowcated cores => 550 cores
* Each Node has 512GB RAM = 22\*512 = 11264GB => 11.2TB, allowcated RAM = 6 TB
* Each Node has 5TB Hard-Disk = 22\*5 = 110TB => allowcated Hard-Disk (I.e., HDFS) = 80TB

**Explain about distcp in Hadoop**

Hadoop provides HDFS Distributed File copy (distcp) tool for copying large amounts of HDFS files within or in between HDFS clusters. It is implemented based on Mapreduce framework and thus it submits a map-only MapReduce job to parallelize the copy process. Usually this tool is useful for copying files between clusters from production to development environments. It supports some advanced command options while copying files. Below are the list of command options it supports

* $ hadoop distcp –help

The basic syntax for using this command is

$ hadoop distcp hdfs://namenodeX/src hdfs://namenodeY/dest

$ hadoop distcp hdfs://nameservice1/user/hive/warehouse/ap\_us\_stage.db/sales\_market hdfs://nameservice2/user/aravind/sales\_market

Some of the frequently useful command options are listed below. All these are not mandatory but just optional.

i) -atomic: This option is used to either commit all changes at a time or no changes should be committed. This makes sure that no partial copying is allowed. Either all files are copied entirely or no file is copied.

ii) -overwrite: By default distcp will skip copying the files that already exist in the destination directory but these can be overwritten unconditionally with this option.

iii) -update: If we need to copy only missing files or changed files, this options is very helpful and minimizes the copy time by copying only missing files/updated files instead of all the source files.

iv) -m <n> : This option lets user to specify the maximum number of mappers to be used.

v) -delete : Deletes the existing files in the destination directory but not in source directory.

Ex:- hadoop distcp –update -m 5 /test /input

**Advantages over hadoop fs -put command or hadoop fs -cp**:

hadoop fs -put command or hadoop fs -cp command can be used to copy the files from local file system into hadoop cluster and from one hadoop cluster to another respectively but here the process is sequential, i.e. only one process will be run to copy file by file. But the advantage of using hadoop distcp command will give us the flexibility to specify the number of parallel tasks should be run in the background to copy files between clusters.

Thus by parallel processing, hadoop’s distcp is the better option to copy bulk data/ huge number of files from one machine to another (or cluster to cluster) than using fs -put or fs -cp commands.

* Get the file from HDFS to the local file system using get command.

$ $HADOOP\_HOME/bin/hadoop fs -get /user/output/ /home/hadoop\_tp/

-rm ----> removes only a file which in the directory

Ex:- $ hadoop fs -rm /csv/sample.txt

-rm -R -------> removes both directory and file which is inside that particular directory

Ex:- $ hadoop fs -rm -R /csv/sample.txt

**Word count in LFS**

jayreddy@ubuntu:~$ mkdir csv

jayreddy@ubuntu:~$ mv sample.txt csv

jayreddy@ubuntu:~$ cat csv/sample.txt

hadoop is a bigdata solution

Spark is a processing framework

jayreddy@ubuntu:~$ cd csv

jayreddy@ubuntu:~/csv$ wc sample.txt

2 10 61 sample.txt

**Word count in HDFS**

jayreddy@ubuntu:~$ hadoop fs -cat /csv/orders.txt|wc

500 1000 20832

jayreddy@ubuntu:~$ hadoop fs -cat /csv/orders.txt|wc –l = lines

500

jayreddy@ubuntu:~$ hadoop fs -cat /csv/orders.txt|wc –w = words

1000

jayreddy@ubuntu:~$ hadoop fs -cat /csv/orders.txt|wc –c = characters

20832

* jayreddy@ubuntu:~$ hadoop fs -count /csv

1 3 24555 /csv

No.of directories=1

No.of files = 3

Total size of the 3 files = 24555 bytes

* jayreddy@ubuntu:~$ hadoop fs -count /csv/orders.txt

0 1 20832 /csv/orders.txt

No.of directories=0

No.of files = 1

Total size of the orders.txt file = 20832 bytes

**Before chmod**

jayreddy@ubuntu:~$ hadoop fs -ls /csv

Found 3 items

-rw-r--r-- 1 jayreddy supergroup 20832 2018-05-09 09:31 /csv/orders.txt

-rw-r--r-- 1 jayreddy supergroup 3435 2018-05-09 21:50 /csv/orders.txt.gz

-rw-r--r-- 1 jayreddy supergroup 288 2018-05-09 00:37 /csv/sample.csv

r = read = 4, w = write = 2, - = nothing = 0

**After applying chmod**

jayreddy@ubuntu:~$ hadoop fs -chmod 777 /csv/orders.txt

jayreddy@ubuntu:~$ hadoop fs -ls /csv

Found 3 items

-rwxrwxrwx 1 jayreddy supergroup 20832 2018-05-09 09:31 /csv/orders.txt

-rw-r--r-- 1 jayreddy supergroup 3435 2018-05-09 21:50 /csv/orders.txt.gz

-rwxrwxrwx 1 jayreddy supergroup 288 2018-05-09 00:37 /csv/sample.csv

r = read = 4, w = write = 2, x = executable = 1

How to change owner

$ sudo chown root file.txt

How to change group

$ sudo chgrp root file.txt

**fsck**

jayreddy@ubuntu:~$ hadoop fsck /csv

Connecting to NameNode via [http://localhost:50070](http://localhost:50070/)

FSCK started by jayreddy (auth:SIMPLE) from /127.0.0.1 for path /csv at Thu May 10 00:42:19 PDT 2018

...Status: HEALTHY

Total size: 24555 B

Total dirs: 1

Total files: 3

Total symlinks: 0

Total blocks (validated): 3 (avg. block size 8185 B)

Minimally replicated blocks: 3 (100.0 %)

Over-replicated blocks: 0 (0.0 %)

Under-replicated blocks: 0 (0.0 %)

Mis-replicated blocks: 0 (0.0 %)

Default replication factor: 1

Average block replication: 1.0

Corrupt blocks: 0

Missing replicas: 0 (0.0 %)

Number of data-nodes: 1

Number of racks: 1

FSCK ended at Thu May 10 00:42:19 PDT 2018 in 2 milliseconds

The filesystem under path '/csv' is HEALTHY

**Hive SerDe**

**Csv SerDe**

The SerDe interface allows you to instruct Hive as to how a record should be processed. A SerDe is a combination of a Serializer and a Deserializer (hence, Ser-De). The Deserializer interface takes a string or binary representation of a record, and translates it into a Java object that Hive can manipulate. The Serializer, however, will take a Java object that Hive has been working with, and turn it into something that Hive can write to HDFS or another supported system. Commonly, Deserializers are used at query time to execute SELECT statements, and Serializers are used when writing data, such as through an INSERT-SELECTstatement.

Suppose, you have a CSV file – ‘sample.csv’ present in ‘/temp’ directory in your HDFS with the following entries: id name email gender ip\_address

1, Hugh Jackman,hughjackman@cam.ac.uk,Male,136.90.241.52

2, David Lawrence,dlawrence1@gmail.com,Male,101.177.15.130

3, Andy Hall,andyhall2@yahoo.com,Female,114.123.153.64

4, Samuel Jackson,samjackson231@sun.com,Male,89.60.227.31

5, Emily Rose,rose.emily4@surveymonkey.com,Female,119.92.21.19

So, I will implement the CSV SerDe as follows:

CREATE EXTERNAL TABLE sample (id int, name string, email string, gender string, ip\_address string)

ROW FORMAT SERDE ‘org.apache.hadoop.hive.serde2.OpenCSVSerde’

STORED AS TEXTFILE LOCATION ‘/csv’;

Now, you can perform any query on the table ‘sample’:

SELECT name FROM sample WHERE gender = ‘male’;

OK

Hugh Jackman

David Lawrence

Samuel Jackson

**Sqoop**

**--as - parquetfile**

$ sqoop import --connect “jdbc:mysql://localhost:3306/dbname” --username xxxxx --password yyyy --table <table\_name> --as - parquetfile --compress --compression-codec snappy -m n --target - dir /user/today\_prac2

O/P :-

/user/today\_prac2/

.metadata

15566aaaaabbbbbb15626266.parquet

15566aaaaabbbbbb15626266.parquet

15566aaaaabbbbbb15626266.parquet

15566aaaaabbbbbb15626266.parquet

.metadata

* descriptor.properties

#Dataset descriptor for today\_prac2

#Tue May 08 00:39:07 PDT 2018

Location = hdfs\://localhost\:8020/user/today\_prac2

Version = 1

compressionType = snappy

format = parquet

* schema.avsc

{

"type" : "record",

"name" : "orders",

"doc" : "Sqoop import of orders",

"fields" : [ {

"name" : "order\_id",

"type" : [ "null", "int" ],

"default" : null,

"columnName" : "order\_id",

"sqlType" : "4"

},

* schemas/1.avsc

{

"type" : "record",

"name" : "orders",

"doc" : "Sqoop import of orders",

"fields" : [ {

"name" : "order\_id",

"type" : [ "null", "int" ],

"default" : null,

"columnName" : "order\_id",

"sqlType" : "4"

},

**Hive Table creation by using schema**

Hive> CREATE TABLE AVRO\_TEST

row format SerDe ‘org.apache.hadoop.hive.serde2.avro.AvroSerDe’

stored as avro

TBLPROPERTIES(‘avro.schema.url’ =‘hdfs://localhost:8020/user/today\_prac2/.metadata/schema.avsc’);

Hive> CREATE EXTERNAL TABLE PARQUET\_TEST like AVRO\_TEST stored as parquet location '/user/today\_prac2';

**PARQUET SERDE**

Hive> CREATE EXTERNAL TABLE ikea\_netharland123 (number int, name string...)

ROW FORMAT SERDE ‘org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe’

STORED AS INPUTFORMAT

‘org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat’

OUTPUTFORMAT

‘org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat’

Location ‘/user/hive/….’;

**Hive Important Concepts**

* If you want to know the Resource Manager (RM) is running which is related to YARN and Map Reduce, Node Manager also we can use the command

$ ps – ef|grep - i manager

* If you want to know status of NameNode, DataNode and SecondaryNameNode run Under user HDFS

$ ps –fu hdfs

* If you want to know ResourceManager and NodeManager running status

$ ps –fu yarn

* To know the size of the file

Hive> dfs –du –s –h /user/hive/warehouse/sqoop\_import.db/order\_items

**ORDER BY clause:**

* The “ORDER BY” syntax in HQL is similar to the syntax of “ORDER BY” in SQL language.
* ORDER BY is the clause we can use with “SELECT” statement in Hive Queries which guarantees total ordering of data.
* It ensures Global sorting in Hive.
* ORDER BY clause use columns on hive tables for grouping of particular column values mentioned with ORDER BY.
* If the mentioned ORDER BY field is a “String” then it will display the results in “lexicographical order”.
* “At the backend it has to be passed on to a Single Reducer”.

Query:- SELECT \*FROM employees\_guru ORDER BY department;

**SORT BY clause**

* SORT BY clause performs on column names of hive tables to sort the Output. We can mention DESC for sorting the results set in Descending order and ASC for sorting results set in Ascending order.
* In this “SORT BY” It will sort the rows before feeding to the reducer.
* Always SORT BY depends on column types

a) If column type is numeric it will sort in numeric order

b) If column type is String it will sort in “lexicographical order”.

Query:- SELECT \* FROM employees\_guru SORT BY id DESC;

**DISTRIBUTE BY clause**

* DISTRIBUTE BY clause used on tables present in hive.
* Hive uses the columns in DISTRIBUTE BY to distribute the rows “among the reducers”.
* All distribute by columns will go to the same reducers.
* It ensures each of ‘N’ reducers gets non-overlapping ranges of column.
* It doesn’t sort the output of each reducer.

Query:- SELECT id, name FROM employees\_guru DISTRIBUTE BY id;

**CLUSTER BY clause**

* CLUSTER BY clause used as an alternative for both “DISTRIBUTE BY AND SORT BY” clauses in Hive HQL.
* CLUSTER BY clause used on tables present in Hive
* Hive uses the columns in CLUSTER BY to Distribute the rows among the reducers
* CLUSTER BY columns will go to the multiple reducers it ensures sorting orders of values present in multiple reducers.

Ex:- CLUSTER BY clause mentioned on the column of name “ID” of table name ‘X’ the output, when executing this query will gives results to multiple reducers at the backend, But at frontend it is an alternative clause for both “SORT BY” and ”DISTRIBUTE BY”

So if you want to store results into multiple reducers we can go with CLUSTER BY.

Query:- SELECT id, name FROM employees\_guru CLUSTER BY id;

**Internal/Managed Tables:**

* Internal/Managed table is tightly coupled in nature. In this type of table, first we have to create table and load the data.
* We can call this Managed table as “Data on Schema”.
* Table data is managed by Hive by moving Data into its warehouse directory configured by hive.metastore.warehouse.dir by default “/user/hive/warehouse”.
* By dropping this table, both data and metadata (schema) will be deleted i.e., these tables are owned by Hive (internal process is -move).

**When to choose Internal/Managed table?**

* If the processing data available in local file system.
* If we want Hive to manage the complete lifecycle of data including deletion.

**External Table:**

* External table is loosely coupled in nature. Data will be available in HDFS. The table is going to create on HDFS Data.
* These tables are not managed or owned by Hive. And tables data will not be copied into hive warehouse directory but maintained at external location.
* We can call this one as “Schema on Data”.
* At the time of dropping the table, it drops only Schema; the data will be still available in HDFS as before (Internal process is copy).
* External tables Provides an option to create multiple schemas for the data stored in HDFS instead of deleting the data every time whenever schema updates.
* “LOCATION” keyword is mandatory to create an external table otherwise table will be managed by Hive only even if you create with “External” keyword.

**When to choose External table?**

* If Processing Data Available in HDFS.
* Useful when the files are being used outside of Hive. Provides convenience to share the tables data with other tools like Pig, HBase and Map Reduce.

Note:- If we are not Specifying the ‘location’ keyword at the time of Table creation, we can load the Data Manually.

* I created managed table and then I load the data which is on HDFS location, after I drop the table then I recognize that both data and on HDFS and data in hive warehouse was deleted (Note:- when I was loading data which is on HDFS into hive table by using “load data inpath ‘/user/data’ into table Emp” move operation will take place hence data will be moved from HDFS location to Hive warehouse location)

Why we need concept of Meta-Data in Hive?

Because HDFS (Hive on top of HDFS) stores the Data only in the form of files or directories It does not store Data in the form of Tables, hence to store Data in the form tables on HDFS we required metadata of tables hence we are maintaining metastore concept in hive.

What kind of applications is supported by Hive?

Applications Supported by Hive are:-

Log Processing, Text Mining, Document Indexing, Google Analytics,

Sentiment Analysis, Predictive Modeling, Hypothesis Testing

List different modes for metastore deployment in Hive?

Hive provides following three modes for metastore deployment.

a) Embedded metastore

By default metastore service runs in the same JVM as the Hive service. In this mode both metastore service and hive service runs in the same JVM by using embedded Derby Database. In this only one embedded Derby database can access the database files on disk at any one time, so only one Hive session could be open at a time.

b) Local Metastore

This mode allows us to have multiple Hive sessions i.e. multiple users can use the metastore at the same time. In this mode Hive metastore service runs in the same process as the main Hive Server process, but the metastore database runs in a separate process.

c) Remote Metastore

In this mode metastore runs on its own separate JVM not in the Hive service JVM. If other processes want to communicate with the metastore server they can communicate using Thrift Network APIs. We can also have one more metastore servers in this case to provide more availability. This also brings better manageability/security because the database tier can be completely firewalled off. And the clients no longer need share database credentials with each Hive user to access the metastore database.

Explain how the client can interact with Hive?

The Client can interact with the Hive in the below three ways:-

* Hive Thrift Client: The Hive server is exposed as thrift service. Hence it is possible to interact with HIVE with any programming language that supports thrift.
* JDBC Driver: Hive uses pure Type 4 JDBC driver to connect to the server which is defined in org.apache.hadoop.HIVE.JDBC.HiveDriver class. . Pure Java applications may use this driver in order to connect to application using separate host and port. The BeeLine CLI uses JDBC Driver to connect to the HIVE Server.
* ODBC Driver: An ODBC Driver allows application that support ODBC to connect to the HIVE server. By default Apache does not ships the ODBC Driver but it is freely available by many vendors.

What are the limitations of Apache Hive?

Hive has following limitations:

1) It does not allow row level inserts, updates or deletes. It only provides the option of dropping of table when it comes to deleting something because behind the scenes hive works with files on HDFS.

2) Hive takes less time to load the data because of its property "scheme on read" but it takes longer time to query the data because data has to be verified against the schema at the time of querying.

3) Previously it did not support the transaction processing because it had no support for ACID properties but recently ACID properties has been added to version hive 0.14 but it leads to performance degradation.

4) Read only views are allowed but materialized views are not allowed.

5) It does not support triggers.

6) No difference between NULL and null values

Hive Static Partitioning

Hive> CREATE TABLE epart (empid int, name string, salary int, sex string, deptno int)

Partitioned by (s string)

Row format delimited

Fields terminated by ‘,’

Stored as textfile;

Note: Instead of ‘s’ if you put ‘sex’ then we will get error because already sex column is there in the create table statement. In a table with two names the column should not be exist. Column name should be Unique so instead of Sex I am using ‘s’

Now how-many columns are there in the table?

5+1=6 here 5 are physical columns 1 is logical column the logical column is for partition

Note: Partitions are not created when you create the table (at the time creation) Partitions are created when you load the data (at the time of loading data)

Loading of Data into static partitioned table

If the table has already have existed data, ex: empl then

Hive> INSERT OVERWRITE TABLE epart

Partition (s= ’f’)

SELECT \* FROM empl where sex = ‘f’

Hive Static Partitioning by using multiple columns

Hive> CREATE TABLE epart3 (empid int, name string, salary int, sex string, deptno int)

Partitioned by (s string, d int)

Row format delimited

Fields terminated by ‘,’

Stored as textfile;

Loading of Data into static partitioned table for multiple partitioned columns

Hive> INSERT OVERWRITE TABLE epart3

Partition (s= ’m’, d=11)

SELECT \* FROM empl where sex = ‘m’ and deptno = 11;

Hive Dynamic Partitioning

Hive> CREATE TABLE dpart (empid int, empname string, salary int, sex string, deptno int)

Partitioned by (d int, s string)

Row format delimited

Fields terminated by ‘,’

Stored as textfile;

Note: Bydefault this dynamic loading feature is disabled we have to enable that one

Hive> set hive.exec.dynamic.partition = true;

When you set it is true the dynamic partition feature is enabled, but what it will do is “except primary partition column remaining partition column will become dynamic”.

I want to make first one also dynamic so

Hive> set hive.exec.dynamic.partition.mode = nonstrict;

Loading of Data into Dynamic Partitioned Table

Hive> INSERT OVERWRITE TABLE dpart

Partition (d, s)

SELECT \* FROM empl;

Note: now this load will be failed why because ‘dpart’ is expecting 5+2=7 columns but ‘empl’ is having only ‘5’ columns

Expectation ‘7’ but supply is ‘5’ that is why this load will fail.

Error= number of columns mismatched

* But in case of static loading this failure was not happened because “you are fixing the partition column while loading ”.But in case of Dynamic loading we are not fixing the partition columns that’s why don’t use ‘\*’ in dynamic loading.
* Static Partition in Hive
* Insert input data files individually into a partition table is Static Partition
* Usually when loading files (big files) into Hive Tables static partitions are preferred
* Static Partition saves your time in loading data compared to dynamic partition
* You “statically” add a partition in table and move the file into the partition of the table.
* We can alter the partition in static partition
* Dynamic Partition in Hive
* single insert to partition table is known as dynamic partition
* Usually dynamic partition load the data from non partitioned table
* Dynamic Partition takes more time in loading data compared to static partition
* When you have large data stored in a table then Dynamic partition is suitable.
* If you want to partition number of column but you don’t know how many columns then also dynamic partition is suitable

Correct Query

Hive> INSERT OVERWRITE TABLE dpart

Partition (d, s)

SELECT empid, empname, salary, sex, deptno, deptno, sex FROM empl;

**How to alter a Hive Table**

Alter table statements enable you to change the structure of an existing table. You can add columns/partitions, change serde, add table and serde properties, or rename the table itself. Similarly alter table partition statements allow you change the properties of a specific partition in the named column

1) ALTER TABLE <table\_name> RENAME TO <new\_name>

2) ALTER TABLE <table\_name> ADD COLUMNS (COL\_SPEC [, COL\_SPEC…)

3) ALTER TABLE <table\_name> DROP [COLUMN] column\_name

4) ALTER TABLE <table\_name> CHANGE existed\_column\_name new\_column\_name new\_type

5) ALTER TABLE <table\_name> REPLACE columns (col\_spec [, col\_spec…])

**Examples**

1) ALTER TABLE Employee RENAME TO Emp;

2) ADD COLUMNS statement

The following query adds a column named dept to the Employee table

Hive> ALTER TABLE Employee ADD columns (dept string, salary int,…,…)

3) Change Statement

ALTER TABLE Employee CHANGE name Emp\_Name string

ALTER TABLE Employee CHANGE salary Emp\_Salary Double

4) Replace Statement

The following query deletes all columns from the Employee table and replaces it with new columns

Hive> ALTER TABLE Employee REPLACE COLUMNS (eid int Emp\_Id int, ename string Emp\_Name string)

Map-side-join:-

Map-side-join is a process where joins between two tables are performed in the Map phase without the involvement of Reduce phase.

Map-side Joins allows a table to get loaded into memory ensuring a very fast join operation, performed entirely within a mapper and that too without having to use both map and reduce phases. In case your queries frequently run with small table joins, you might see a very substantial decrease in the time taken to compute the queries after usage of map-side joins.

<https://drive.google.com/open?id=0ByJLBTmJojjzSWhfXzI5NGh4RGM(input> data find from chrome)

CREATE TABLE IF NOT EXISTS dataset1 (eid int, first\_name String, last\_name String, email String, gender String, ip\_address String)

row format delimited

fields terminated BY ','

tblproperties ("skip.header.line.count"="1");

CREATE TABLE IF NOT EXISTS dataset2 (eid int, first\_name String, last\_name String) row format delimited fields terminated BY ',' tblproperties ("skip.header.line.count"="1");

Map Join

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

2. By setting the following property to true.

Set hive.auto.convert.join=true

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

SELECT /\*+ MAPJOIN (dataset2) \*/ dataset1.first\_name, dataset1.eid, dataset2.eid FROM dataset1 JOIN dataset2 ON dataset1.first\_name = dataset2.first\_name;

Bucket-Map join:-

A bucket map join is used when the tables are large and all the tables used in the join are bucketed on the join columns. In this type of join, one table should have buckets in multiples of the number of buckets in another table. For example, if one table has 2 buckets then the other table must have either 2 buckets or a multiple of 2 buckets (2, 4, 6, and so on). If the preceding condition is satisfied then the joining can be done at the mapper side only, otherwise a normal inner join is performed. This means that only the required buckets are fetched on the mapper side and not the complete table. That is, only the matching buckets of all small tables are replicated onto each mapper. Doing this, the efficiency of the query is improved drastically. In a bucket map join, data is not sorted. Hive does not support a bucket map join by default. The following property needs to be set to “true” for the query to work as a bucket map join:

Set hive.optimize.bucketmapjoin = true;

In this type of join, not only tables need to be bucketed but also data...

The constraint for performing Bucket-Map join is:

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

Create Bucketing tables:-

1) CREATE TABLE IF NOT EXISTS dataset1\_bucketed (eid int, first\_name String, last\_name String, email String, gender String, ip\_address String) clustered by (first\_name) into 4 buckets row format delimited fields terminated BY ',’;

2) CREATE TABLE IF NOT EXISTS dataset2\_bucketed (eid int, first\_name String, last\_name String) clustered by (first\_name) into 8 buckets row format delimited fields terminated BY ',’;

Inserting data into Bucketed tables:-

1) Insert into table dataset1\_bucketed select \* from dataset1;

2) Insert into table dataset2\_bucketed select \* from dataset2;

Here, we have two tables that are bucketed. We can now perform Bucket-map join between these two datasets.

Here, for the first table we have created 4 buckets and for the second table we have created 8 buckets on the same column. Now, we can perform Bucket-map join on these two tables.

For performing Bucket-Map join, we need to set this property in the Hive shell.

Set hive.optimize.bucketmapjoin = true

SELECT /\*+MAPJOIN(dataset2\_bucketed)\*/ dataset1\_bucketed.first\_name, dataset1\_bucketed.eid, dataset2\_bucketed.eid FROM dataset1\_bucketed JOIN dataset2\_bucketed ON dataset1\_bucketed.first\_name= dataset2\_bucketed.first\_name;

Sort Merge Bucket (SMB) Map Join:-

In this recipe, you will learn how to use a Sort-Merge-Bucket-Map join in Hive. A sort merge bucket map join is an advanced version of a bucket map join. If the data in the tables is sorted and bucketed on the join columns at the same time then a bucket sort merge map join comes into the picture. In this type of join, all the tables must have an equal number of buckets as each mapper will read a bucket from each table and will perform a sort merge bucket map join.

It is mandatory for the data to be sorted in this join condition. The following parameter needs to be set to true for sorting the data or data can be sorted manually:

Set hive.enforce.sorting = true;

If the tables being joined are sorted and bucketized on the join columns and have the same number of buckets, a sort-merge join can be performed. The corresponding buckets are joined with each other at the mapper. Here we have 4 buckets for dataset1 and 8 buckets for dataset2. Now, we will create another table with 4 buckets for dataset2.

For performing the SMB-Map join, we need to set the following properties:

1) Set hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;

2) Set hive.optimize.bucketmapjoin = true;

3) Set hive.optimize.bucketmapjoin.sortedmerge = true;

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

* insert overwrite table dataset1\_bucketed select \* from dataset1 sort by first\_name;

The above command will overwrite the data in the old table and insert the data as per the query.

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

We will now overwrite the data into the dataset2\_bucketed table, using the following command:

* insert overwrite table dataset2\_bucketed select \* from dataset2 sort by first\_name;

Now, let us perform the join between tables having 4 buckets and 8 buckets.

* SELECT /\*+ MAPJOIN (dataset2\_bucketed) \*/ dataset1\_bucketed.first\_name, dataset1\_bucketed.eid, dataset2\_bucketed.eid FROM dataset1\_bucketed JOIN dataset2\_bucketed ON dataset1\_bucketed.first\_name = dataset2\_bucketed.first\_name;

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

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

CREATE TABLE IF NOT EXISTS dataset2\_sbucketed1 (eid int, first\_name String, last\_name String) clustered by (first\_name) into 4 buckets row format delimited fields terminated BY ','

Insert overwrite table dataset2\_bucketed1 select \* from dataset2 sort by first\_name;

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

SELECT /\*+ MAPJOIN (dataset2\_sbucketed1) \*/ dataset1\_bucketed.first\_name, dataset1\_bucketed.eid, dataset2\_sbucketed1.eid FROM dataset1\_bucketed JOIN dataset2\_sbucketed1 ON dataset1\_bucketed.first\_name = dataset2\_sbucketed1.first\_name;

In the above screen shot, you can see that it has taken 22.69 seconds. As the dataset sizes are too small, this analysis results may vary based on the dataset size.

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

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

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

Skewed Join:-

In this recipe, you will learn how to use a skew join in Hive.

A skew join is used when there is a table with skew data in the joining column. A skew table is a table that is having values that are present in large numbers in the table compared to other data. Skew data is stored in a separate file while the rest of the data is stored in a separate file.

If there is a need to perform a join on a column of a table that is appearing quite often in the table, the data for that particular column will go to a single reducer, which will become a bottleneck while performing the join. To reduce this, a skew join is used.

The following parameter needs to be set for a skew join:

Set hive.optimize.skewjoin=true;

Set hive.skewjoin.key=100000;

In the last blogs, I discussed Common Join and Map Join. In this blog, I am going to discuss Skewed Join. Remember the blog of Common Join, I mentioned one of major issues in Common Join is the join performs poorly when data is skewed. The query is waiting for the longest running reducers on the skewed keys while majority of reducers complete the join operation.

Skewed Join is exactly targeting this problem. At runtime, it scans the data and detects the keys with a large skew, which is controlled by parameter hive.skewjoin.key. Instead of processing those keys, it stores them temporarily in an HDFS directory. Then in a map-reduce job, process those skewed keys. The same key need not be skewed for all the tables, and so the follow-up map-reduce job (for the skewed keys) would be much faster, since it would be a map-join.

For example, let’s say we have a join with Table A and B. Both Table A and B has skewed data “mytest” in the joining column. Assuming Table B has fewer rows with skewed data in Table A. The first step is to scan B and save all rows with the key “mytest” in an in-memory hash table. Then run a set of mappers to read Table A to perform the followings:

If it has skewed key “mytest”, then it will use hashed version of B for the join.

For all other keys, send the rows to a reducer that performs the join. The same reducer will get rows from the mappers that scanning Table B.

We can see that Table B is scanned twice during Skewed Join. The skewed keys in Table A are read and processed by the mapper, and perform map-side join. The rows with skewed keys in Table A have never sent to the reducer. For the rest of keys in Table A, they use the regular common join approach.

To use Skewed Join, you need to understand your data and query. Set parameter hive.optimize.skewjoin to true. Parameter hive.skewjoin.key is optional and it is 100000 by default.

How to Identify the Join

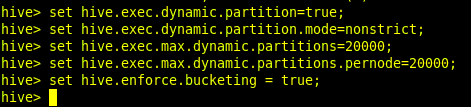
When using EXPLAINS command, you will see handleSkewJoin: true below Join Operator and Reduce Operator Tree.

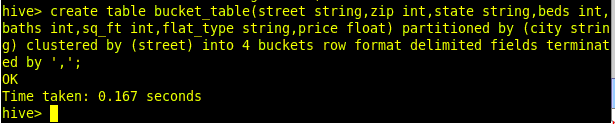
Example

* Set hive.optimize.skewjoin = true;
* Set hive.skewjoin.key=500000;
* Set hive.skewjoin.mapjoin.map.tasks=10000;
* Set hive.skewjoin.mapjoin.min.split=33554432;

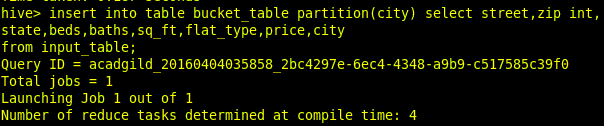
Bucketing in Hive:-

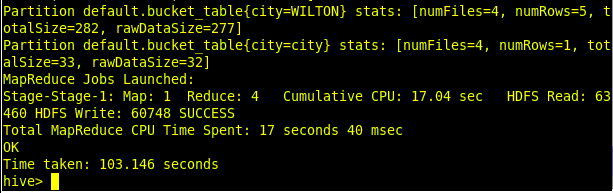
To populate the bucketed table, we have to set hive.enforce.bucketing property to ‘true’, so that the Hive knows to create the number of buckets declared in the table definition.





Query to Retrieve Data from Bucketed Table:-





How To: Convert External table to internal table or vice-versa

Description:

According to Qubole, the External Tables and Internal Tables (non-External) differ only in the aspect that dropping the table drops meta-data in External tables, whereas it drops both meta-data and data in the case of internal tables.

How To:

Use the following command to convert an external table to an internal table:

Use <db name>;

ALTER TABLE <tablename> SET TBLPROPERTIES ('EXTERNAL'='FALSE');

Use the following command to convert an internal table to an external table:

Use <db name>;

ALTER TABLE <tablename> SET TBLPROPERTIES ('EXTERNAL'='TRUE');

Q) When we do Partitioning and when we do Bucketing in hive?

PARTITIONING will be used when there are few unique values in the Column - which you want to load with your required WHERE clause

BUCKETING will be used if there are multiple unique values in your Where clause

Hive Performance Tuning:-

1. Enable compression in Hive:

By enabling compression at various phases (i.e. on final output, intermediate data), we achieve the performance improvement in Hive Queries.

Hive> set io.compression.codecs;

io.compression.codecs=

org.apache.hadoop.io.compress.GzipCodec,

org.apache.hadoop.io.compress.DefaultCodec,

org.apache.hadoop.io.compress.BZip2Codec,

org.apache.hadoop.io.compress.SnappyCodec

Enable Compression on Intermediate Data

A complex Hive query is usually converted to a series of multi-stage MapReduce jobs after submission, and these jobs will be chained up by the Hive engine to complete the entire query. So “intermediate output” here refers to the output from the previous MapReduce job, which will be used to feed the next MapReduce job as input data.

We can enable compression on Hive Intermediate output by setting the property hive.exec.compress.intermediate either from Hive Shell using set command or at site level in hive-site.xml file.

<Property>

<name>hive.exec.compress.intermediate</name>

<Value>true</value>

<Description>

This controls whether intermediate files produced by Hive between multiple map-reduce jobs are compressed. The compression codec and other options are determined from Hadoop config variables mapred.output.compress\*

</description>

</property>

<Property>

<name>hive.intermediate.compression.codec</name>

<value>org.apache.hadoop.io.compress.SnappyCodec</value>

<description/>

</property>

<Property>

<name>hive.intermediate.compression.type</name>

<Value>BLOCK</value>

<description/>

</property>

Hive> set hive.exec.compress.intermediate=true;

Hive> set hive.intermediate.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;

Hive> set hive.intermediate.compression.type=BLOCK;

2. Optimize Joins

Auto Map Joins

Skew Joins

Enable Bucketed Map Joins

3. Avoid Global Sorting in Hive:

Global Sorting in Hive can be achieved with ORDER BY clause but this comes with a drawback. ORDER BY produces a result by setting the number of reducers to one, making it very inefficient for large datasets. When a globally sorted result is not required, then we can use SORT BY clause. SORT BY produces a sorted file per reducer. If we need to control which reducer a particular row goes to, we can use DISTRIBUTE BY clause, for example,

SELECT id, name, salary, dept FROM employee

DISTRIBUTE BY dept

SORT BY id ASC, name DESC;

4. Use Spark Execution Engine

5. Optimize LIMIT operator

6. Enable Parallel Execution:

Hive converts a query into one or more stages. Stages could be a Mapreduce stage, sampling stage, a merge stage, a limit stage. By default, Hive executes these stages one at a time. A particular job may consist of some stages that are not dependent on each other and could be executed in parallel, possibly allowing the overall job to complete more quickly. Parallel execution can be enabled by setting below properties.

<Property>

<name>hive.exec.parallel</name>

<Value>true</value>

<Description>Whether to execute jobs in parallel</description>

</property>

<Property>

<name>hive.exec.parallel.thread.number</name>

<Value>8</value>

<Description> how many jobs at most can be executed in parallel</description>

</property>

7. Enable Vectorization

Vectorization feature is introduced into hive for the first time in hive-0.13.1 release only. By vectorized query execution, we can improve performance of operations like scans, aggregations, filters and joins, by performing them in batches of 1024 rows at once instead of single row each time.

We can enable vectorized query execution by setting below three properties in either hive shell or hive-site.xml file.

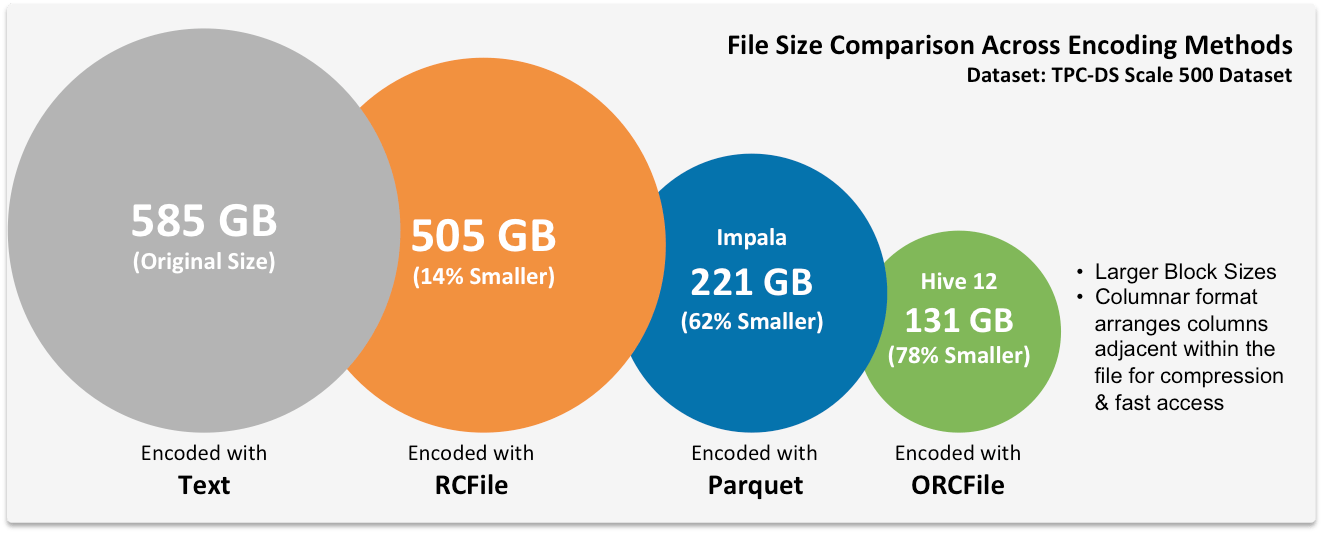
Hive> set hive.vectorized.execution.enabled = true;

Hive> set hive.vectorized.execution.reduce.enabled = true;

Hive> set hive.vectorized.execution.reduce.groupby.enabled = true;

12. Use ORC File Format

Using ORC (Optimized Record Columnar) file format we can improve the performance of Hive Queries very effectively. Below picture on file format best depicts the power of ORC file over other formats.



Create Tables with ORC File Format

We can create new hive table with ORC file format with just by adding STORED AS ORC clause to CREATE TABLE command in hive. Optionally we can provide compression techniques in TBLPROPERTIES clause.

Convert Existing Tables to ORC

Create a table with the same schema as the source table and STORED AS ORC, then we can submit below command to copy data from regular old table new ORC formatted table.

Hive> INSERT OVERWRITE TABLE orc\_emp SELECT \* FROM EMP;

Hive> INSERT OVERWRITE TABLE orc\_emp SELECT \* FROM EMP;

Key Default Notes

orc.compress ZLIB Compression to use in addition to columnar compression (one ofNONE, ZLIB, SNAPPY)

orc.compress.size 262,144 (= 256 KiB) Number of bytes in each compression chunk

orc.stripe.size 268,435,456 (= 256 MiB) Number of bytes in each stripe

orc.row.index.stride 10,000 Number of rows between index entries (must be >= 1,000)

orc.create.index true Whether to create inline indexes

Example ORC table creation:

Hive> CREATE TABLE EMP\_ORC (id int, name string, age int, address string)

> STORED AS ORC tblproperties (“orc.compress" = “SNAPPY");

Hive> INSERT OVERWRITE TABLE EMP\_ORC SELECT \* FROM EMP

Date Functions in Hive

Date data types do not exist in Hive. In fact the dates are treated as strings in Hive. The date functions are listed below.

UNIX\_TIMESTAMP ()

This function returns the number of seconds from the UNIX epoch (1970-01-01 00:00:00 UTC) using the default time zone.

UNIX\_TIMESTAMP (string date)

This function converts the date in format 'yyyy-MM-dd HH:mm:ss' into UNIX timestamp. This will return the number of seconds between the specified date and the UNIX epoch. If it fails, then it returns 0.

Example: UNIX\_TIMESTAMP ('2000-01-01 00:00:00') returns 946713600

UNIX\_TIMESTAMP (string date, string pattern)

This function converts the date to the specified date format and returns the number of seconds between the specified date and UNIX epoch. If it fails, then it returns 0.

Example: UNIX\_TIMESTAMP ('2000-01-01 10:20:30','yyyy-MM-dd') returns 946713600

FROM\_UNIXTIME (bigint number\_of\_seconds [, string format])

The FROM\_UNIXTIME function converts the specified number of seconds from UNIX epoch and returns the date in the format 'yyyy-MM-dd HH:mm:ss'.

Example: FROM\_UNIXTIME (UNIX\_TIMESTAMP ()) returns the current date including the time. This is equivalent to the SYSDATE in oracle.

TO\_DATE (string timestamp)

The TO\_DATE function returns the date part of the timestamp in the format 'yyyy-MM-dd'.

Example: TO\_DATE ('2000-01-01 10:20:30') returns '2000-01-01'

YEAR (string date)

The YEAR function returns the year part of the date.

Example: YEAR ('2000-01-01 10:20:30') returns 2000

MONTH (string date)

The MONTH function returns the month part of the date.

Example: MONTH ('2000-03-01 10:20:30') returns 3

DAY (string date), DAYOFMONTH (date)

The DAY or DAYOFMONTH function returns the day part of the date.

Example: DAY ('2000-03-01 10:20:30') returns 1

HOUR (string date)

The HOUR function returns the hour part of the date.

Example: HOUR ('2000-03-01 10:20:30') returns 10

MINUTE (string date)

The MINUTE function returns the minute part of the timestamp.

Example: MINUTE ('2000-03-01 10:20:30') returns 20

SECOND (string date)

The SECOND function returns the second part of the timestamp.

Example: SECOND ('2000-03-01 10:20:30') returns 30

WEEKOFYEAR (string date)

The WEEKOFYEAR function returns the week number of the date.

Example: WEEKOFYEAR ('2000-03-01 10:20:30') returns 9

DATEDIFF (string date1, string date2)

The DATEDIFF function returns the number of days between the two given dates.

Example: DATEDIFF ('2000-03-01', '2000-01-10') returns 51

DATE\_ADD (string date, int days)

The DATE\_ADD function adds the number of days to the specified date

Example: DATE\_ADD ('2000-03-01', 5) returns '2000-03-06'

DATE\_SUB (string date, int days)

The DATE\_SUB function subtracts the number of days to the specified date

Example: DATE\_SUB ('2000-03-01', 5) returns ‘2000-02-25’

Hive Built-in Functions

Functions in Hive are categorized as below.

Numeric and Mathematical Functions: These functions mainly used to perform mathematical calculations.

Date Functions: These functions are used to perform operations on date data types like adding the number of days to the date etc.

String Functions: These functions are used to perform operations on strings like finding the length of a string etc.

Conditional Functions: These functions are used to test conditions and returns a value based on whether the test condition is true or false.

Collection Functions: These functions are used to find the size of the complex types like array and map. The only collection function is SIZE. The SIZE function is used to find the number of elements in an array and map. The syntax of SIZE function is

SIZE (Array<A>) and SIZE (MAP<key, value>)

Type Conversion Function: This function is used to convert from one data type to another. The only type conversion function is CAST. The syntax of CAST is

CAST (expr as <type>)

The CAST function converts the expr into the specified type.

Table Generating Functions: These functions transform a single row into multiple rows. EXPLODE is the only table generated function. This function takes array as an input and outputs the elements of array into separate rows. The syntax of EXPLODE is

EXPLODE (ARRAY<A>)

When you use the table generating functions in the SELECT clause, you cannot specify any other columns in the SELECT clause.

What are the key features of HDFS?

The various Features of HDFS are:

Fault Tolerance – In Apache Hadoop HDFS, Fault-tolerance is working strength of a system in unfavorable conditions. Hadoop HDFS is highly fault-tolerant, in HDFS data is divided into blocks and multiple copies of blocks are created on different machines in the cluster. If any machine in the cluster goes down due to unfavorable conditions, then a client can easily access their data from other machines which contain the same copy of data blocks.

High Availability – HDFS is highly available file system; data gets replicated among the nodes in the HDFS cluster by creating a replica of the blocks on the other slaves present in the HDFS cluster. Hence, when a client wants to access his data, they can access their data from the slaves which contains its blocks and which is available on the nearest node in the cluster. At the time of failure of a node, a client can easily access their data from other nodes.

Data Reliability – HDFS is a distributed file system which provides reliable data storage. HDFS can store data in the range of 100s petabytes. It stores data reliably by creating a replica of each and every block present on the nodes and hence, provides fault tolerance facility.

Replication – Data replication is one of the most important and unique features of HDFS. In HDFS, replication data is done to solve the problem of data loss in unfavorable conditions like crashing of the node, hardware failure and so on.

Scalability – HDFS stores data on multiple nodes in the cluster, when requirement increases we can scale the cluster. There are two scalability mechanisms available: vertical and horizontal.

Distributed Storage – In HDFS all the features are achieved via distributed storage and replication. In HDFS data is stored in distributed manner across the nodes in the HDFS cluster.

What is the difference between NAS and HDFS?

* Hadoop distributed file system (HDFS) is the primary storage system of Hadoop. HDFS designs to store very large files running on a cluster of commodity hardware. While Network-attached storage (NAS) is a file-level computer data storage server. NAS provides data access to a heterogeneous group of clients.
* HDFS distribute data blocks across all the machines in a cluster. Whereas NAS, data stores on a dedicated hardware.
* Hadoop HDFS is designed to work with Mapreduce Framework. In Mapreduce Framework computation move to the data instead of Data to computation. NAS is not suitable for Mapreduce, as it stores data separately from the computations.
* Hadoop HDFS runs on the cluster commodity hardware which is cost effective. While a NAS is a high-end storage device which includes high cost.

Howb to find duplicate rows in hive?

<https://stackoverflow.com/questions/46748186/how-to-find-duplicate-rows-in-hive>

hive> select [every column], count(\*)

from mytable

group by [every column]

having count(\*) > 1;

This way you will get list of duplicated rows.

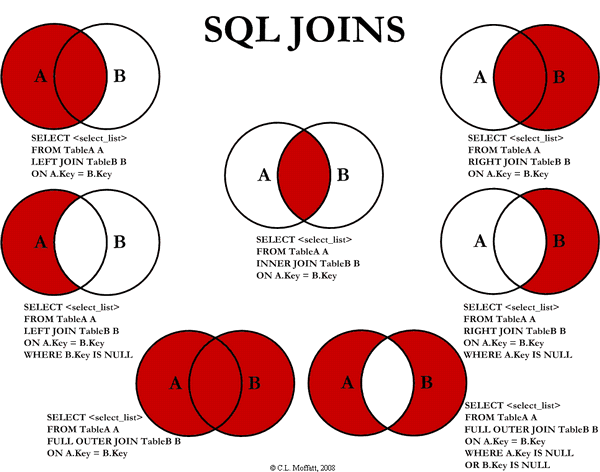
How to join multiple tables in Hive?

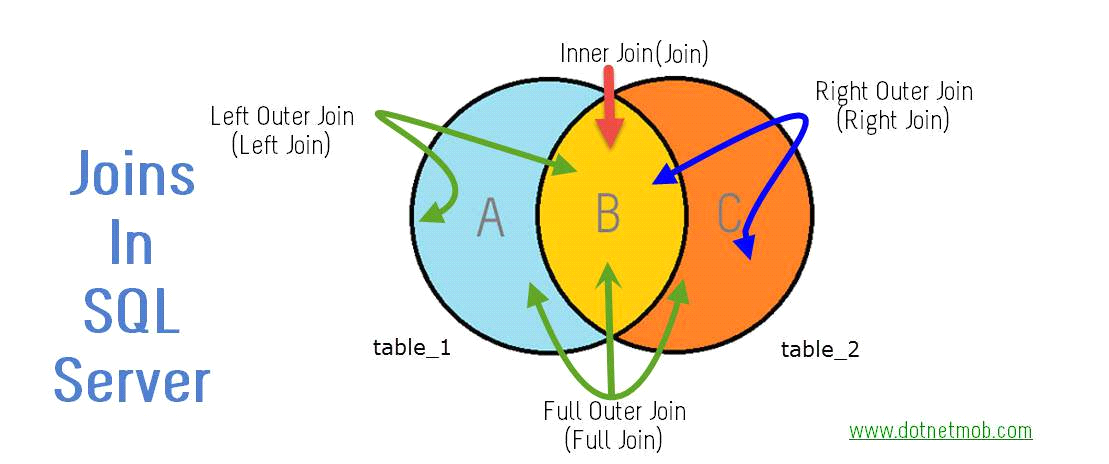
<http://hadoopquiz.blogspot.in/2016/05/hive-joining-multiple-tables-in-single.html>

<https://stackoverflow.com/questions/29028616/multiple-table-join-in-hive>

Yes, you can join multiple tables in a single query. This allows many opportunities for Hive to make optimizations that couldn't be done if you broke it into separate queries.

Joins in Hive





A cross join is simplest of all. It implements only one logical query processing

Phase-a Cartesian Product. This phase operates on the two tables provided as inputs to the join and produces a Cartesian product of the two. That is, each row from one input is matched with all rows from the other. So if you have m rows in one table and n rows in the other, you get m×n rows in the result.

Then are Inner joins : They apply two logical query processing phases: A Cartesian product between the two input tables as in a cross join, and then it filters rows based on a predicate that you specify in ON clause (also known as Join condition).

Next comes the third type of joins Outer Joins: In an outer join, you mark a table as a preserved table by using the keywords LEFT OUTER JOIN, RIGHT OUTER JOIN, or FULL OUTER JOIN between the table names. The OUTER keyword is optional. The LEFT keyword means that the rows of the left table are preserved; the RIGHT keyword means that the rows in the right table are preserved; and the FULL keyword means that the rows in both the left and right tables are preserved. The third logical query processing phase of an outer join identifies the rows from the preserved table that did not find matches in the other table based on the ON predicate. This phase adds those rows to the result table produced by the first two phases of the join, and uses NULL marks as placeholders for the attributes from the nonpreserved side of the join in those outer rows.

Now if we look at the question: To return records from the left table which are not found in the right table use Left outer join and filter out the rows with NULL values for the attributes from the right side of the join.

Why we Integrate Hive with external sources?

Ans) Because Hive did not support for row level inserts updates and deletes. Hive doesn’t support Transactions hence Hive adds extensions to provide better performance in the context of Hadoop and to integrate with custom extensions and even external programs.

Hive Commands:

1) Hive> SET hive.cli.print.current.db=true;

2) Hive> SET hive.cli.print.header=true;

3) Hive> SELECT \* FROM system\_logs LIMIT 3;

CSV SERDE

ADD JAR /path/to/csv-serde.jar

CREATE TABLE stocks (ymd string…)

ROW FORMAT SERDE ‘com.bizo.hive.serde.csv.CSVserde’

STORED AS TEXTFILE;

Parquet File:

1) The latest hotness in file formats for Hadoop is columnar file storage.

2) Basically this means that instead of just storing rows of data adjacent to one another you also store column values adjacent to each other. So datasets are partitioned both horizontally and vertically.

3) This is particularly useful if your data processing framework just needs access to a subset of data that is stored on disk as it can access all values of a single column very quickly without reading whole records.

4) Just like ORC file, it’s great for compression with great query performance especially efficient when querying data from specific columns. Parquet format is computationally intensive on the write side, but it reduces a lot of I/O cost to make great read performance.

5) It enjoys more freedom than ORC file in schema evolution, that it can add new columns to the end of the structure.

6) If you’re chopping and cutting up datasets regularly then these formats can be very beneficial to the speed of your application, but frankly if you have an application that usually needs entire rows of data then the columnar formats may actually be a detriment to performance due to the increased network activity required.

7) One huge benefit of columnar oriented file formats is that data in the same column tends to be compressed together which can yield some massive storage optimizations (as data in the same column tends to be similar). It supports both File-Level Compression and Block-Level Compression. File-level compression means you compress entire files regardless of the file format, the same way you would compress a file in Linux. Some of these formats are splittable (e.g. bzip2, or LZO if indexed). Block-level compression is internal to the file format, so individual blocks of data within the file are compressed. This means that the file remains splittable even if you use a non-splittable compression codec like Snappy. However, this is only an option if the specific file format supports it.

8) Since Avro and Parquet have so much in common when choosing a file format to use with HDFS, we need to consider read performance and write performance. Because the nature of HDFS is to store data that is write once, read multiple times, we want to emphasize on the read performance. The fundamental difference in terms of how to use either format is this: Avro is a Row based format. If you want to retrieve the data as a whole, you can use Avro. Parquet is a Column based format. If your data consists of lot of columns but you are interested in a subset of columns, you can use Parquet.

ORC (Optimized Row Columnar) Input Format

ORC stands for Optimized Row Columnar which means it can store data in an optimized way than the other file formats. ORC reduces the size of the original data up to 75%. As a result the speed of data processing also increases and shows better performance than Text, Sequence and RC file formats. An ORC file contains rows data in groups called as Stripes along with a file footer. ORC format improves the performance when Hive is processing the data. We cannot load data into ORCFILE directly. First we need to load data into another table and then we need to overwrite it into our newly created ORCFILE. ORC File Format Full Form is Optimized Row Columnar File Format. ORC File format provides very efficient way to store relational data than RC file, by using ORC File format we can reduce the size of original data up to 75%.Comparing to Text, Sequence, RC file formats ORC is better.

* Column stored separately
* Knows Types - Uses Types specific en-coders
* Stores statistics (Min,Max,Sum,Count)
* Has Light weight Index
* Skip over blocks of rows that don’t matter
* Larger Blocks - 256 MB by default, Has an index for block boundaries

Using ORC files improves performance when Hive is reading, writing, and processing data comparing to Text, Sequence and RC. RC and ORC shows better performance than Text and Sequence File formats. Comparing to RC and ORC File formats always ORC is better as ORC takes less time to access the data comparing to RC File Format and ORC takes less space to store data. However, the ORC file increases CPU overhead by increasing the time it takes to decompress the relational data. ORC File format feature comes with the Hive 0.11 version and cannot be used with previous versions.

AVRO Format

Apache Avro is a language-neutral data serialization system. It was developed by Doug Cutting, the father of Hadoop. Since Hadoop writable classes lack language portability, Avro becomes quite helpful, as it deals with data formats that can be processed by multiple languages. Avro is a preferred tool to serialize data in Hadoop. Avro is an opinionated format which understands that data stored in HDFS is usually not a simple key/value combo like int/string. The format encodes the schema of its contents directly in the file which allows you to store complex objects natively. Honestly, Avro is not really a file format, it’s a file format plus a serialization and de-serialization framework with regular old sequence files you can store complex objects but you have to manage the process. Avro handles this complexity whilst providing other tools to help manage data over time and is a well thought out format which defines file data schemas in JSON (for interoperability), allows for schema evolutions (remove a column, add a column), and multiple serialization/deserialization use cases. It also supports block-level compression. For most Hadoop-based use cases Avro becomes really good choice. Avro depends heavily on its schema. It allows every data to be written with no prior knowledge of the schema. It serializes fast and the resulting serialized data is lesser in size. Schema is stored along with the Avro data in a file for any further processing. In RPC, the client and the server exchange schemas during the connection. This exchange helps in the communication between same named fields, missing fields, extra fields, etc. Avro schemas are defined with JSON that simplifies its implementation in languages with JSON libraries. Like Avro, there are other serialization mechanisms in Hadoop such as Sequence Files, Protocol Buffers, and Thrift.

Thrift & Protocol Buffers vs. Avro

Thrift and Protocol Buffers are the most competent libraries with Avro. Avro differs from these frameworks in the following ways –

* Avro supports both dynamic and static types as per the requirement. Protocol Buffers and Thrift use Interface Definition Languages (IDLs) to specify schemas and their types. These IDLs are used to generate code for serialization and deserialization.
* Avro is built in the Hadoop ecosystem. Thrift and Protocol Buffers are not built in Hadoop ecosystem. Unlike Thrift and Protocol Buffer, Avro's schema definition is in JSON and not in any proprietary IDL.

RC (Row-Columnar) File Input Format

RCFILE stands of Record Columnar File which is another type of binary file format which offers high compression rate on the top of the rows used when we want to perform operations on multiple rows at a time. RCFILEs are flat files consisting of binary key/value pairs, which share much similarity with SEQUENCE FILE. RCFILE stores columns of a table in form of record in a columnar manner. It first partitions rows horizontally into row splits and then it vertically partitions each row split in a columnar way. RCFILE first stores the metadata of a row split, as the key part of a record, and all the data of a row split as the value part. This means that RCFILE encourages column oriented storage rather than row oriented storage. This column oriented storage is very useful while performing analytics. It is easy to perform analytics when we “hive’ a column oriented storage type. We cannot load data into RCFILE directly. First we need to load data into another table and then we need to overwrite it into our newly created RCFILE.

* Columns stored separately
* Read and decompressed only needed one.
* Better compression
* Columns stored as binary Blobs
* Depend on Meta store to supply Data types
* Large Blocks - 4MB default
* Still search file for split boundary.

When Hive can avoid MapReduce?

Hive can implements some kind of queries without using MapReduce, in so called local mode

Ex. 1) SELECT \*FROM employees;

2) SELECT \*FROM employees WHERE country = ‘US’ AND state = ‘CA’ LIMIT 100;

Furthermore, Hive will attempt to run other operations in local mode if the hive.exec.mode.local.auto property is set to true

Set hive.exec.mode.local.auto = true;

Otherwise, Hive uses MapReduce to run all other queries.

Case Statements

Hive> select chiller type, chillervalue,

> Case

> When chillervalue < 50000.0 then 'low'

> When chillervalue >= 50000.0 and optimized value < 70000.0 Then 'middle'

> When chillervalue >= 70000.0 and optimized value < 100000.0 Then 'high'

> Else 'very high'

> End as results from Targetable;

Window Based Functions in Hive:

Lag

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

Select ticker, date\_, close, lag (close, 1) over (partition by ticker) as yesterday\_price from stocks

Here using lag we can display the yesterday’s closing price of the ticker. Lag is to be used with over function, inside the Over Function you can use partition or order by classes.

Lead

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

Now using the lead function, we will find that whether the following day’s closing price is higher or lesser than Today’s and that can be done as follows.

Select ticker, date\_, close, case (lead (close, 1) over (partition by ticker)-close)>0 when true then "higher" when false then "lesser" end as Changes from stocks

FIRST\_VALUE

It returns the value of the first row from that window. With the below query, you can see the first row high price of the ticker for all the days.

Select ticker, first\_value (high) over (partition by ticker) as first\_high from stocks

LAST\_VALUE

It is the reverse of FIRST\_VALUE. It returns the value of the last row from that window. With the below query, you can see the last row high price value of the ticker for all the days.

Select ticker, last\_value (high) over (partition by ticker) as first\_high from stocks

The OVER clause

OVER with standard aggregates:

COUNT

SUM

MIN

MAX

AVG

OVER with a PARTITION BY statement with one or more Partitioning Columns.

OVER with PARTITION BY and ORDER BY with one or more partitioning and or ordering columns.

Analytics functions

* RANK
* ROW\_NUMBER
* DENSE\_RANK
* CUME\_DIST
* PERCENT\_RANK
* NTILE

Count

It returns the count of all the values for the expression written in the over clause. From the below query, we can find the number of rows present for each ticker.

Select ticker, count (ticker) over (partition by ticker) as cnt from stocks

Sum

It returns the sum of all the values for the expression written in the over clause. From the below query, we can find the sum of all the closing stock prices for that particular ticker.

Select ticker, sum (close) over (partition by ticker) as total from stocks

Finding running total

For suppose let us take if you want to get running total of the volume\_for\_the\_day for all the days for every ticker then you can do this with the below query.

Select ticker, date\_, volume\_for\_the\_day, sum (volume\_for\_the\_day) over (partition by ticker order by date\_) as running\_total from stocks

Rank

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

The below query will rank the closing prices of the stock for each ticker. The same you can see in the below screenshot.

Select ticker, close, rank () over (partition by ticker order by close) as closing from stocks

O/p

C 3.75 1

C 3.79 2

C 3.85 3

C 3.85 3

C 3.87 5

C 3.87 5

C 3.88 7

C 3.94 8

C 4.0 9

C 4.02 10

C 4.06 11

C 4.08 12

C 4.09 13

C 4.09 13

C 4.1 15

C 4.1 15

Row\_Number ()

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

From the below query, you will get the ticker, closing price and its row number for each ticker.

Select ticker, close, row\_number () over (order by close) as num from stocks

O/p

C 3.75 1

C 3.79 2

C 3.85 3

C 3.85 4

C 3.87 5

C 3.87 6

C 3.88 7

C 3.94 8

C 4.0 9

C 4.02 10

C 4.06 11

C 4.08 12

C 4.09 13

C 4.09 14

C 4.1 15

C 4.1 16

C 4.12 17

C 4.13 18

C 4.14 19

C 4.15 20

C 4.16 21

C 4.19 22

BSX 5.47 23

Dense\_Rank

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

The below query will rank the closing prices of the stock for each ticker. The same you can see in the below screenshot.

select ticker, close, dense\_rank () over(partition by ticker order by close) as closing from stocks

O/p

C 3.75 1

C 3.79 2

C 3.85 3

C 3.85 3

C 3.87 4

C 3.87 4

C 3.88 5

C 3.94 6

C 4.0 7

C 4.02 8

C 4.06 9

C 4.08 10

C 4.09 11

C 4.09 11

C 4.1 12

C 4.1 12

**INDEXING IN HIVE**

This blog focuses of the concepts involved in indexing in Hive. This post includes the following topics:

* When to use indexing.
* How indexing is helpful.
* How to create indexes for your tables.
* Perform some operations regarding the indexing in Hive.

What is an Index?

An Index acts as a reference to the records. Instead of searching all the records, we can refer to the index to search for a particular record. Indexes maintain the reference of the records. So that it is easy to search for a record with minimum overhead. Indexes also speed up the searching of data.

Why to use indexing in Hive?

Hive is a data warehousing tool present on the top of Hadoop, which provides the SQL kind of interface to perform queries on large data sets. Since Hive deals with Big Data, the size of files is naturally large and can span up to Terabytes and Petabytes. Now if we want to perform any operation or a query on this huge amount of data it will take large amount of time.

In a Hive table, there are many numbers of rows and columns. If we want to perform queries only on some columns without indexing, it will take large amount of time because queries will be executed on all the columns present in the table.

The major advantage of using indexing is; whenever we perform a query on a table that has an index, there is no need for the query to scan all the rows in the table. Further, it checks the index first and then goes to the particular column and performs the operation.

So if we maintain indexes, it will be easier for Hive query to look into the indexes first and then perform the needed operations within less amount of time.

Eventually, time is the only factor that everyone focuses on.

When to use Indexing?

Indexing can be use under the following circumstances:

* If the dataset is very large.
* If the query execution takes more amount of time than you expected.
* If a speedy query execution is required.
* When building a data model.

Indexes are maintained in a separate table in Hive so that it won’t affect the data inside the table, which contains the data. Another major advantage for indexing in Hive is that indexes can also be partitioned depending on the size of the data we have.

Types of Indexes in Hive

1) Compact Indexing

2) Bitmap Indexing

Bit map indexing was introduced in Hive 0.8 and is commonly used for columns with distinct values.

**Differences between Compact and Bitmap Indexing**

The main difference is the storing of the mapped values of the rows in the different blocks. When the data inside a Hive table is stored by default in the HDFS, they are distributed across the nodes in a cluster. There needs to be a proper identification of the data, like the data in block indexing. This data will be able to identity which row is present in which block, so that when a query is triggered it can go directly into that block. So, while performing a query, it will first check the index and then go directly into that block.

* Compact indexing stores the pair of indexed column’s value and its blockid.
* Bitmap indexing stores the combination of indexed column value and list of rows as a bitmap.

Let’s now understand what is bitmap?

A bitmap is a type of memory organization or image file format used to store digital images so with this meaning of bitmap, we can redefine bitmap indexing as given below.

“Bitmap index stores the combination of value and list of rows as a digital image.”

The following are the different operations that can be performed on Hive indexes:

* Creating index
* Showing index
* Alter index
* Dropping index

**Creating Index in Hive**

Syntax for creating a compact index in Hive is as follows:

CREATE INDEX index\_name

ON TABLE table\_name (column1, column2, …)

AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'

WITH DEFERRED REBUILD;

* Here, in the place of index\_name we can give any name of our choice, which will be the table’s INDEX NAME.
* In the ON TABLE line, we can give the table\_name for which we are creating the index and the names of the columns in brackets for which the indexes are to be created. We should specify the columns which are available only in the table.
* The org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler’ line specifies that a built in CompactIndexHandler will act on the created index, which means we are creating a compact index for the table.
* The WITH DEFERRED REBUILD statement should be present in the created index because we need to alter the index in later stages using this statement.

This syntax will create an index for our table, but to complete the creation, we need to complete the REBUILD statement. For this to happen, we need to add one more alter statement. A MapReduce job will be launched and the index creation is now completed.

ALTER INDEX index\_name on table\_name REBUILD;

This ALTER statement will complete our REBUILDED index creation for the table.

Examples – Creating Index

In this section we will first execute the hive query on non-indexed table and will note down the time taken by query to fetch the result.

In the second part, we will be performing the same query on indexed table and then will compare the time taken by query to fetch the result with the earlier case.

We will be demonstrating this difference of time with practical examples.

In first scenario we are performing operations on non-indexed table.

Let’s create a normal managed table to contain the Olympic dataset first.

create table Olympic (athlete STRING,age INT, country STRING, year STRING, closing STRING, sport STRING, gold INT, silver INT, bronze INT, total INT) row format delimited fields terminated by '\t' stored as textfile;

Here we are creating a table with name ‘Olympic’. The schema of the table is as specified and the data inside the input file is delimited by tab space. At the end of the line, we have specified ‘stored as textfile’, which means we are using a TEXTFILE format. You can check the schema of your created table using the command ‘describe Olympic;’ We can load data into the created table as follows:

load data local inpath ‘path of your file‘ into table Olympic;

We have successfully loaded the input file data into the table which is in the TEXTFILE format.

Let’s perform an Average operation on this ‘Olympic’ data. Let’s calculate the average age of the athletes using the following command:

SELECT AVG(age) from Olympic;

Here we can see the average age of the athletes to be 26.405433646812956 and the time for performing this operation is 21.08 seconds.

Now, let’s create the index for this table:

CREATE INDEX olympic\_index

ON TABLE Olympic (age)

AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'

WITH DEFERRED REBUILD

ALTER INDEX olympic\_index on Olympic REBUILD;

Here we have created an index for the ‘Olympic’ table on the age column. We can view the indexes created for the table by using the below command:

show formatted index on Olympic;

We can see the indexes available for the ‘Olympic’ table in the above image.

Now, let’s perform the same Average operation on the same table.

We have now got the average age as 26.405433646812956, which is same as the above, but now the time taken for performing this operation is 17.26 seconds, which is less than the above case.

Now we know that by using indexes we can reduce the time of performing the queries.

Can we have different indexes for the same table?

Yes! We can have any number of indexes for a particular table and any type of indexes as well.

Let’s create a Bitmap index for the same table:

CREATE INDEX olympic\_index\_bitmap

ON TABLE Olympic (age)

AS 'BITMAP'

WITH DEFERRED REBUILD;

ALTER INDEX olympic\_index\_bitmap on Olympic REBUILD;

* Here, As ‘BITMAP’ defines the type of index as BITMAP.

We have successfully created the Bitmap index for the table.

We can check the available indexes for the table using the below command:

show formatted index on Olympic;

Average Operation with Two Indexes

Now, let’s perform the same Average operation having the two indexes.

This time, we have got the same result in 17.614 seconds which is same as in the case of compact index.

Note: With different types (compact, bitmap) of indexes on the same columns, for the same table, the index which is created first is taken as the index for that table on the specified columns.

Now let’s delete one index using the following command:

DROP INDEX IF EXISTS olympic\_index ON Olympic;

We can check the available indexes on the table to verify whether the index is deleted or not.

We have successfully deleted one index i.e., olympic\_index, which is a compact index.

We now have only one index available for our table, which is a bitmap index.

Average Operation with Bitmap Index

Let’s perform the same Average age operation on the same table with bitmap index.

We have got the average age as 26.105433646812956, which is same as the above cases but the operation was done in just 16.47 seconds, which is less than the above two cases.

Through the above examples, we have proved the following:

* Indexes decrease the time for executing the query.
* We can have any number of indexes on the same table.
* We can use the type of index depending on the data we have.
* In some cases, Bitmap indexes work faster than the Compact indexes and vice versa.

When not to use indexing?

It is essential to know when and where indexing shouldn’t be used. They should not be used in the following scenarios:

* Indexes are advised to build on the columns on which you frequently perform operations.
* Building more number of indexes also degrade the performance of your query.
* Type of index to be created should be identified prior to its creation (if your data requires bitmap you should not create compact).This leads to increase in time for executing your query.

**VIEWS IN HIVE**

String Functions in Hive

1) ASCII

ASCII Function converts the first character of the string into its numeric ASCII value.

Ex: hive> select ASCII (‘hadoop’) from Tri100 where sal=22000;

104

hive> select ASCII(‘Hadoop’) from Tri100 where sal=22000;

72

hive> select ASCII(‘A’) from Tri100 where sal=22000;

65

2) CONCAT

The CONCAT function concatenates all the strings/columns.

hive> select CONCAT(name,'+',location) from Tri100;

Rahul +Hyderabad

Mohit+Banglore

Rohan+Banglore

Ajay+Bangladesh

Srujay+Srilanka

hive> select CONCAT(name, ' ', 'Got Hike of', ' ', Hike, ' ' ,'from', ' ' ,sal) from Tri100;

Rahul Got Hike of 40000 from 30000

Mohit Got Hike of 25000 from 22000

Rohan Got Hike of 40000 from 33000

Ajay Got Hike of 45000 from 40000

Srujay Got Hike of 30000 from 25000

3) CONCAT\_WS

Syntax: “CONCAT\_WS (string delimiter, string str1, str2……)”

The CONCAT\_WS function concatenates all the strings only strings and Column with datatype string.

hive> select CONCAT\_WS('+',name, location) from Tri100;

Rahul+Hyderabad

Mohit+Banglore

Rohan+Banglore

Ajay+Bangladesh

Srujay+Srilanka

hive> select CONCAT\_WS(' ',name,'from',location) from Tri100;

Rahul from Hyderabad

Mohit from Banglore

4) FIND\_IN\_SET:

Syntax: “FIND\_IN\_SET (string search\_string, string source\_string\_list)”

The FIND\_IN\_SET function searches for the search\_string in the source\_string\_list and returns the position of the first occurrence in the source\_string\_list. Here the source\_string\_list should be comma delimited one.

hive> select FIND\_IN\_SET('ha', 'ho,hi,ha,bye') from Tri100 where sal=22000;

3

hive> select FIND\_IN\_SET('rahul', name) from Tri100;

1

hive> select FIND\_IN\_SET('ho', 'homahahi') from Tri100;

0

5) LPAD:

Syntax: “LPAD (string str, int len, string pad)”

The LPAD function returns the string with a length of len characters left-padded with pad.

hive> select LPAD(name,6,'#') from Tri100;

#rahul

#Mohit

#Rohan

##Ajay

Srujay

6) RPAD:

Syntax: “RPAD (string str, int len, string pad)”

The RPAD function returns the string with a length of len characters Right-padded with pad.

hive> select RPAD(name,6,'#') from Tri100;

rahul#

Mohit#

Rohan#

Ajay##

Srujay

7) TRIM:

TRIM Function removes all the trailing spaces from the string.

hive> select TRIM(' hive ') from Tri100 where sal=22000;

hive

hive> select LTRIM(' hive') from Tri100 where sal=22000;

hive

hive> select RTRIM('hive ') from Tri100 where sal=22000;

hive

8) SPLITT:

Syntax: SPLITT (‘string1:string2’,’pat’)

Split function splits the string depending on the pattern path and returns an array of strings.

hive> select split('hadoop:hive',':') from Tri100 where sal=22000;

["hadoop","hive"]

9) SubString:

The SUBSTR or SUBSTRING function returns a part of the source string from the start position with the specified length of characters.

If the length is not given, then it returns from the start position to the end of the string.

hive> select SUBSTR(name,4) from Tri100;

ul

it

hive> select SUBSTR(name,2,3) from Tri100;

ahu

ohi

oha

jay

ruj

hive> select SUBSTR('hadoophivehive',2,5) from Tri100 where sal=22000;

adoop

10) Format:

Syntax: “FORMAT\_NUMBER (number X, int D)”

Formats the number X to a format like #,###,###.##, rounded to D decimal places and returns result as a string. If D=0 then the value will only have fraction part there will not be any decimal part.

hive> select name, format\_number (Hike,2) from Tri100;

Rahul 40,000.00

Mohit 25,000.00

11) INSTRING:

Syntax: “INSTRING (string str, string substring)”

Returns the position of the first occurrence of substring in str. Returns null if either of the arguments are null and returns 0 if substring could not be found in str. Beware that this is not zero based. The first character in str has index 1.

hive> select instr ('rahul', 'ul') from Tri100 where sal=22000;

4

12) Locate:

Syntax: “Locate (string substring, string str[,int pos])”

Returns the position of the first occurrence of Substring in String after position pos

hive> select locate('ul','rahul',2) from Tri100 where sal=22000;

4

hive> select locate('ul','rahul',5) from Tri100 where sal=22000;

0

Regexp\_Extract:

Syntax: “regexp\_extract (string subject, string pattern, int index)”

Returns the string extracted using the pattern.

hive> select regexp\_extract('foothebar', 'foo(.\*?)(bar)',2) from Tri100 where sal=22000;

bar

hive> select regexp\_extract('foothebar', 'foo(.\*?)(bar)',1) from Tri100 where sal=22000;

the

hive> select regexp\_extract('foothebar', 'foo(.\*?)(bar)',0) from Tri100 where sal=22000;

foothebar

Regexp\_Repalce:

Syntax: “regexp\_replace (string INITIAL\_STRING, string PATTERN, string REPLACEMENT)”

Returns the string resulting from replacing all substrings in INITIAL\_STRING that match the java regular expression syntax defined in PATTERN with instances of REPLACEMENT.

hive> select regexp\_replace ('foothebar', 'oo|ba', ' ') from Tri100 where sal=22000;

fther

**How to handle JSON Data**

I/P:-

{"name":"Ravi","age":25}

{"name":"Rani","city":"Hyd"}

{"name":"Mani","age":24,"city":"Del"}

**Table creation**:-

hive> create table raw(line string);

**Loading data**:-

hive> load data local inpath 'json1' into table raw;

hive> select \*from raw;

OK

{"name":"Ravi","age":25}

{"name":"Rani","city":"Hyd"}

{"name":"Mani","age":24,"city":"Del"}

* Now I wanted it in Structured manner
* and I wanted extract fields from table 'raw'

There are two ways to extract fields

1) get\_json\_object () --- UDF

2) json\_tuple () --- UDTF

1) get\_json\_object () :- To get a particular field

hive> select get\_json\_object (line, '$.name'),

> get\_json\_object (line, '$.age'),

> get\_json\_object (line, '$.city') from raw;

OK

Ravi 25 NULL

Rani NULL Hyd

Mani 24 Del

Note:- In Json if any field is missed we will get NULL

2) How to get multiple fields at a time or to get continuous sequence of fields we use json\_tuple ()

Hive> select x.\* from raw

> lateral view json\_tuple (line,'name','age','city') x as n, a, c;

OK

Ravi 25 NULL

Rani NULL Hyd

Mani 24 Del

Where x.\* is for multiple fields and also it is the alias name for lateral view

Why we use lateral view here

Because json\_tuple () is a UDTF a UDTF function doesn’t allow other column names that is why we use lateral view. Lateral view doesn’t accept other functions it accepts only UDTF

**Final table**

**Table creation**

hive> create table raw1(name string, age int, city string)

> row format delimited

> fields terminated by ','

> stored as textfile;

**Data loading**

hive> insert into table raw1

> select get\_json\_object(line,'$.name'),

> get\_json\_object(line,'$.age'),

> get\_json\_object(line, '$.city') from raw;

Query ID = jayreddy\_20180430033310\_111d5a7f-935c-4e7e-9f72-d3635b2a0c8b

Total jobs = 3

Launching Job 1 out of 3

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

hive> select \* from raw1;

OK

Ravi 25 NULL

Rani NULL Hyd

Mani 24 Del

**Nested JSON**

I/P:-

{"name":"Ravi","age":25,"wife":{"name":"Rani","age":24,"city":"Hyd"},"city":"Del"}

{"name":"Ram","age":30,"wife":{"name":"Sita","qual":"B.Tech","city":"Hyd"},"city":"Hyd"}

Table Creation:-

Hive> create table jraw (line string);

Data Loading:-

Hive> load data local inpath 'json2' into table jraw;

Loading data to table today\_prac.jraw

Table today\_prac.jraw stats: [numFiles=1, totalSize=172]

hive> select \*from jraw;

OK

{"name":"Ravi","age":25,"wife":{"name":"Rani","age":24,"city":"Hyd"},"city":"Del"}

{"name":"Ram","age":30,"wife":{"name":"Sita","qual":"B.Tech","city":"Hyd"},"city":"Hyd"}

Intermediate Table:-

hive> create table jraw1(name string, age int, wife string, city string);

hive> insert into table jraw1

> select x.\* from jraw

> lateral view json\_tuple(line, 'name', 'age', 'wife', 'city') x as n, a, w, c;

Query ID = jayreddy\_20180430050418\_01a0cd3a-4dd7-43e4-a149-4df23783db69

Total jobs = 3

Launching Job 1 out of 3

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

Starting Job = job\_1525088648645\_0001, Tracking URL = <http://ubuntu:8088/proxy/application_1525088648645_0001/>

Kill Command = /home/jayreddy/INSTALL/hadoop-2.6.0/bin/hadoop job -kill job\_1525088648645\_0001

Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 0

2018-04-30 05:05:02,203 Stage-1 map = 0%, reduce = 0%

2018-04-30 05:05:28,511 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 3.85 sec

MapReduce Total cumulative CPU time: 3 seconds 850 msec

hive> select \*from jraw1;

OK

Ravi 25 {"name":"Rani","age":24,"city":"Hyd"} Del

Ram 30 {"name":"Sita","qual":"B.Tech","city":"Hyd"} Hyd

Final Table:-

hive> create table jraw2(name string, wname string, age int, wage int, wqual string, city string, wcity string)

> row format delimited

> fields terminated by ','

> stored as textfile;

hive> insert into table jraw2

>select name,get\_json\_object(wife,'$.name'),age,get\_json\_object(wife,'$.age'),get\_json\_object(wife,'$.qual'),city,get\_json\_object(wife,'$.city') from jraw1;

Query ID = jayreddy\_20180430052244\_63a79d4b-1e0e-4990-a953-7697d4e67ca1

Total jobs = 3

Launching Job 1 out of 3

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

hive> select \*from jraw2;

OK

Ravi Rani 25 24 NULL Del Hyd

Ram Sita 30 NULL B.Tech Hyd Hyd

Time taken: 0.114 seconds, Fetched: 2 row(s)

**How to handle xml Data**

I/P:-

<rec><name>Ravi</name><age>25</age></rec>

<rec><name>Rani</name><sex>F</sex></rec>

Hive> create table xraw (line string);

Hive> select \*from xraw;

OK

<rec><name>Ravi</name><age>25</age></rec>

<rec><name>Rani</name><sex>F</sex></rec>

**How to Extract the fields from xml data**:-

* for string fields we use xml parser===> xpath\_string
* for numeric fields we use xml parser===> xpath\_int, xpath\_double, xpath\_long...
* for collection fields extraction we can use xml parser====> xpath(line, 'rec/fieldname/text()')

Hive> select xpath\_string (line, 'rec/name') from xraw;

OK

Ravi

Rani

Hive> select xpath\_int (line, 'rec/age') from xraw;

OK

25

0

* In xml data parsing if you missed string field you will get "blank space"

and if you missed numeric field you will get "zero"

Hive> select xpath\_string (line, 'rec/sex') from xraw;

Ok

F

Final table:-

Hive> create table xraw1 (name string, age int, sex string)

> row format delimited fields terminated by '|'

> stored as textfile;

Data Loading:-

Hive> insert into table xraw1

> select xpath\_string (line, 'rec/name'),

> xpath\_int (line, 'rec/age'),

> xpath\_string (line, 'rec/sex') from xraw;

Query ID = jayreddy\_20180430195316\_8a7acfa7-2113-4332-8130-3e47137210d5

Total jobs = 3

Launching Job 1 out of 3

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

Hive> select \*from xraw1;

OK

Ravi 25

Rani 0 F

I/P:-

<tr><cid>101</cid><pr>1000</pr><pr>2000</pr><pr>3000</pr></tr>

<tr><cid>102</cid><pr>4000</pr><pr>5000</pr><pr>6000</pr></tr>

<tr><cid>103</cid><pr>7000</pr><pr>8000</pr><pr>9000</pr></tr>

Hive> create table xxraw (line string);

Hive> load data local inpath 'xml2' into table xxraw;

Hive> select \*from xxraw;

OK

<tr><cid>101</cid><pr>1000</pr><pr>2000</pr><pr>3000</pr></tr>

<tr><cid>102</cid><pr>4000</pr><pr>5000</pr><pr>6000</pr></tr>

<tr><cid>103</cid><pr>7000</pr><pr>8000</pr><pr>9000</pr></tr>

Hive> select xpath\_int (line,'tr/cid'),

> xpath (line,'tr/pr/text()') from xxraw;

OK

101 ["1000","2000","3000"]

102 ["4000","5000","6000"]

103 ["7000","8000","9000"]

Hive> create table xxraw1 (cid int, pr Array<string>);

Hive> insert into table xxraw1

> select xpath\_int (line,'tr/cid'),

> xpath (line,'tr/pr/text()') from xxraw;

Hive> select \*from xxraw1;

OK

xxraw1.cid xxraw1.pr

101 ["1000","2000","3000"]

102 ["4000","5000","6000"]

103 ["7000","8000","9000"]

Task:- for each customer total bill(price) we need to count

* If i apply explode on pr we get price like below

Hive> select explode (pr) as p from xxraw1;

OK

1000

2000

3000

4000

5000

6000

7000

8000

9000

Note:- But we miss the clarity that each price is of which customer to overcome this situation along with price we need customerID (cid)but there is a rule that along with UDTF function other column expressions are cannot be used so to overcome this problem we are using "lateral view"

Hive> select cid, mypr from xxraw1

> lateral view explode(pr) p as mypr;

OK

101 1000

101 2000

101 3000

102 4000

102 5000

102 6000

103 7000

103 8000

103 9000

Hive> create table xxraw2 (cid int, pr int)

> row format delimited

> fields terminated by ',';

Hive> insert into table xxraw2

> select cid, mypr from xxraw1

> lateral view explode(pr) p as mypr;

Query ID = jayreddy\_20180430214302\_c03509db-a724-494e-9909-5d0a1e5f310f

Total jobs = 3

Launching Job 1 out of 3

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

Hive> select \*from xxraw2;

OK

101 1000

101 2000

101 3000

102 4000

102 5000

102 6000

103 7000

103 8000

103 9000

Hive> create table xxraw3 (cid int, total int);

Hive> insert into table xxraw3

> select cid, sum (pr) as total from xxraw2 group by cid;

Query ID = jayreddy\_20180430215225\_fbebcfd4-94e4-4eb9-b461-ca8000e4e649

Total jobs = 1

Launching Job 1 out of 1

Number of reduce tasks not specified. Estimated from input data size: 1

Hive> select \*from xxraw3;

OK

101 6000

102 15000

103 24000

If your xml tag having parametres how to deal that?

I/P:-

<tr><cid>101</cid><pr id="p1">2000</pr><pr id="p2">4000</pr></tr>

<tr><cid>102</cid><pr id="p3">5000</pr><pr id="p4">6000</pr></tr>

Hive> create table xxraw (line string);

Hive> load data local inpath 'xml3' into table xxxraw3;

Hive> select \*from xxxraw;

OK

<tr><cid>101</cid><pr id="p1">2000</pr><pr id="p2">4000</pr></tr>

<tr><cid>102</cid><pr id="p3">5000</pr><pr id="p4">6000</pr></tr>

Hive> select xpath\_int (line,'tr/cid'),

> xpath (line,'tr/pr/text()') from xxxraw;

OK

101 ["2000","4000"]

102 ["5000","6000"]

**How to take product\_id**

Hive> select xpath (line,'tr/pr/@id') from xxxraw;

OK

["p1","p2"]

["p3","p4"]

Hive> select xpath\_int (line,'tr/cid'),

> xpath (line, 'tr/pr/text()'),

> xpath (line, 'tr/pr/@id') from xxxraw;

OK

101 ["2000","4000"] ["p1","p2"]

102 ["5000","6000"] ["p3","p4"]

**Hive Subqueries**

Usage of subqueries in Apache Hive queries In this article, we will learn how and where to use subqueries in Apache Hive. Sub-queries in Hive queries are allowed in from clause ad where clause.

The following steps explain you in detail about subqueries in Hive

1) **FROM Clause**

Hive> select name, newsalary from (select name, salary\*1.1 as newsalary from employee) emp\_new;

Subquery columns can be accessed in main query. In the above query, We have accessed newsalary column in main query from subquery.

Union and Union all are also allowed in the subquery of from clause.

Hive> select name, newsalary from (

Select name, salary\*1.1 as newsalary from employee

union

Select name, salary from employee

) emp\_new;

2) **In where clause**

Subquery in the where clause is used to define list of elements.

The following query defines list of deptno elements from department table. And the main query retrieves names from employee table whose department number available in department table.

Hive> select emp\_name from employee where deptno IN (select deptno from department);

**We can use both in or not in in where clause**.

IN or NOT IN subqueries should have only one column in subquery.

If subquery has more than one column, Hive throws an error Subquery can contain only 1 item in Select List. Subqueries need to be only right side, Left side sub-queries not allowed.

The following query is not valid as subquery defined left side.

Select name from employee where (select deptno from department) d in (10, 20);

We can use exists or not exists in where clause to compare main query data with subquery data.

The query below compares deptno in employee table to deptno in department table and retrieves matched rows.

Hive> select name from employee e where exists (select name from department d where e.deptno=d.deptno);

**Hive Joins**

Hive>Select employee.name, employee.salary, employee.deptno, employee.doj, department.deptno, department.deptname from employee **JOIN** department **ON** employee.deptno = department.deptno;

Hive>Select employee.name, employee.salary, employee.deptno, employee.doj, department.deptno, department.deptname from employee **INNER JOIN** department **ON** employee.deptno=department.deptno;

Hive> select e.name, e.salary, e.deptno, e.doj, d.deptno, d.deptname from employee e **INNER JOIN** department d **ON** e.deptno=d.deptno;

**Outer Joins**

Hive> select e.name, e.salary, e.deptno, e.doj, d.deptno, d.deptname from employee e **LEFT OUTER JOIN** department **ON** e.deptno=d.deptno;

The output above displays NULL for deptno 25 row as it does not have any data department table.

If we want to see even unmatched rows from department (right side) table, Then we can use right outer join like below.

Hive> select e.name, e.salary, e.deptno, e.doj, d.deptno, d.deptname from employee e **RIGHT OUTER JOIN** department d **ON** e.deptno=d.deptno;

The output above displays NULL for deptno 45 row as it does not have any data in employee table.

If we want to see unmatched rows from both tables, we need to use full outer join like below.

Hive> select e.name, e.salary, e.deptno, e.doj, d.deptno, d.deptname from employee e **FULL OUTER JOIN** department d **ON** e.deptno=d.deptno;

Full outer join output contains NULL rows from both left outer and right outer joins.

**Analysis on Subqueries**

Hive supports subqueries only in the FROM clause. The columns in the subquery select list are available in the outer query just like columns of a table.

Example: Select col1, col2 from (select col3+col4 as col1, col5+col6 as col2 from table 1)table2;

**There are a few limitations**:

* These subqueries are only supported on the right-hand side of an expression.
* IN/NOT IN subqueries may only select a single column.
* EXISTS/NOT EXISTS must have one or more correlated predicates.
* References to the parent query are only supported in the WHERE clause of the subquery.

**Case Statements**

Sometimes it's necessary to translate stored values (or values to be stored) from one representation to another. Suppose there is a column status with legal values from 0 to 9 but the end-users should receive strings which explain the meaning of the numeric values in short, eg.: 'ordered', 'delivered', 'back delivery', 'out of stock', ... . The recommended way to do this is a separate table where the numeric values maps to the explanatory strings. Notwithstanding this, application developers may favor a solution within an application server.

The CASE expression, which is shown on this page, is a technique for solving the described situation as part of a SELECT, INSERT or UPDATE command as well as solving additional problems .As part of the language it's a powerful term which can be applied at plenty places within SQL commands. On this page we focus on its use together with the SELECT command. The strategy and syntax for CASE within INSERT and UPDATE are equivalent and are presented over there. In comparison with the recommended technique of a separate table for the translation the CASE expression is much more flexible (which is not an advantage in all cases).

-- Technical term: "simple case"

-- Select id, contact\_type in a translated version and contact\_value

SELECT id,

CASE contact\_type

WHEN 'fixed line' THEN 'Phone'

WHEN 'mobile' THEN 'Phone'

ELSE 'Not a telephone number'

END,

contact\_value

FROM contact;

* The CASE expression is introduced with its key word CASE and runs up to the END key word. In this first example it specifies a column name and a series of WHEN/THEN clauses with an optional ELSE clause. The WHEN/THEN clauses are compared and evaluated against the values of the named column, one after the other. If none of them hits, the ELSE clause applies. If there is no ELSE clause and non of the WHEN/THEN clauses hit, the NULL special marker will be applied.

The comparison between the values of the column and the fixed values within the WHEN/THEN clause is done solely by "=" (equals). This is a good starting point, but real applications need more than that. Therefore there is a variant of the CASE.

-- Technical term: "searched case"

-- Select person’s name, weight and a denomination of the weight

SELECT firstname, lastname, weight,

CASE

WHEN (weight IS NULL OR weight = 0) THEN 'weight is unknown'

WHEN weight < 40 THEN 'lightweight'

WHEN weight BETWEEN 40 AND 85 THEN 'medium'

ELSE 'heavyweight'

END As Result

FROM person;

* The crucial point is the direct succession of the two key words CASE and WHEN. There is no column name between them. In this variant there must be a complete expression, which evaluates to one of the 3-value-logic terms true, false or unknown, between each WHEN and THEN. Now it is possible to use all the comparisons and Boolean operators as they are known by the WHERE clause. It is even possible to compare different columns or function calls with each other.

**Hive Views**

A view allows a query to be saved and treated like a table. It is a logical construct, as it does not store data like a table. In other words, materialized views are not currently supported by Hive. When a query references a view, the information in its definition is combined with the rest of the query by Hive’s query planner. Logically, you can imagine that Hive executes the view and then uses the results in the rest of the query.

**Views to Reduce Query Complexity**

When a query becomes long or complicated, a view may be used to hide the complexity by dividing the query into smaller, more manageable pieces; similar to writing a function in a programming language or the concept of layered design in software. Encapsulating the complexity makes it easier for end users to construct complex queries from reusable parts. For example, consider the following query with a nested subquery:

FROM (

SELECT \* FROM people JOIN cart

ON (cart.people\_id=people.id) WHERE firstname='john'

) a SELECT a.lastname WHERE a.id=3;

It is common for Hive queries to have many levels of nesting. In the following example, the nested portion of the query is turned into a view:

CREATE VIEW shorter\_join AS

SELECT \* FROM people JOIN cart

ON (cart.people\_id=people.id) WHERE firstname='john';

Now the view is used like any other table. In this query we added a WHERE clause to the SELECT statement. This exactly emulates the original query:

SELECT lastname FROM shorter\_join WHERE id=3;

Hive View For writing queries, we sometimes find it difficult to frame query when it is nested and complex. This scenario may often occur in the case of joins and we will be covering the same in this blog to simplify our way of querying in HIVE with help of VIEWS. When a query becomes long or complicated, a view may be used to hide the complexity by dividing the query into smaller, more manageable pieces; similar to writing a function in a programming language or the concept of layered design in software. Encapsulating the complexity makes it easier for end users to construct complex queries from reusable parts. For example, consider the following query with a nested subquery: FROM (SELECT \* FROM people JOIN cart ON (cart.id1=people.id) WHERE last\_name='Henderson')a SELECT a.first\_name WHERE a.id='40-8340379'; It is common for Hive queries to have many levels of nesting. In the following example of Hive View, the nested portion of the query is turned into a simple one.

First for Hive View, we will create a table inside Hive and load the respective data. Commands for doing so are: create table people(id string, last\_name string, email string, gender string, ip\_address string)row format delimited fields terminated by ','; load data local inpath '/home/prateek/Documents/HIVE/people.csv' into table people; create table cart(id1 string, first\_name string, job\_title string)row format delimited fields terminated by ','; load data local inpath '/home/prateek/Documents/HIVE/cart.csv' into table cart; Instead of using Hive View, while executing formal nested query, we see the size of a query is more and complex also multiple jobs need to complete in a single query. In this case, most difficult part is to guess the combination of columns and rows. But if the query is correct you will have the result. Now for the different approach, we create a VIEW named J\_View as this Hive View will be showing data after joining table. NOTE:- Hive View does not store data permanently. It is a logical construct, as it does not store data like a table. CREATE VIEW J\_View AS SELECT \* FROM people JOIN cart ON (cart.id1=people.id); Also, refer below screenshot for to see the complete data stored in this VIEW we created. Now the Hive View is used like any other table. In this query, we added a WHERE clause to the SELECT statement. This exactly emulates the original query: SELECT first\_name FROM J\_View WHERE id='40-8340379';

Similarly, we can query this Hive View as many times we want for the different result without making our query complex.

CREATE VIEW newusers

AS

SELECT DISTINCT T1.uuid

FROM user\_visit T1

WHERE T1.firstSeen="${hiveconf:date}";

Then give that variable a value when invoking hive:

hive --hiveconf date = 20140522 -f 'create\_newusers\_view.hql'