\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*FILEFORMATS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A file format is a way in which information is stored or encoded in a computer file.

In Hive it refers to how records are stored inside the file.

As we are dealing with structured data, each record has to be its own structure.

How records are encoded in a file defines a file format.

These file formats mainly vary between data encoding, compression rate, usage of space and disk I/O.

Hive does not verify whether the data that you are loading matches the schema for the table or not.

However, it verifies if the file format matches the table definition or not.

Let us now discuss the types of file formats in detail.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TEXTFILE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TEXTFILE format is a famous input/output format used in Hadoop.

In Hive if we define a table as TEXTFILE it can load data of from CSV (Comma Separated Values), delimited by Tabs,

Spaces, and JSON data. This means fields in each record should be separated by comma or space or tab or

it may be JSON(JavaScript Object Notation) data.

By default, if we use TEXTFILE format then each line is considered as a record.

We can create a TEXTFILE format in Hive as follows:

#create table table\_name (schema of the table) row format delimited fields terminated by ',' | stored as TEXTFILE.

At the end, we need to specify the type of file format.

If we do not specify anything it will consider the file format as TEXTFILE format.

The TEXTFILE input and TEXTFILE output format are present in the Hadoop package as shown below:

org.apache.hadoop.mapred.TextInputFormat

org.apache.hadoop.mapred.TextOutputFormat

Let us see one example in Hive about how to create TEXTFILE table format, how to load data into TEXTFILE format and perform one basic select operation in Hive.

Creating TEXTFILE

create table olympic(athelete 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" and the schema of the table is as specified above.

The data inside the above input file is delimited by tab space.

As explained earlier the file format is specified as TEXTFILE at the end.

The schema of the table created above can be checked using 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 our input file data into our table which is of TEXTFILE format.

Now we will perform one basic SELECT operation on the data as shown below:

select athelete from olympic;

The data retrieved is as shown in Figure below:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SEQUENCEFILE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

We know that Hadoop's performance is drawn out when we work with a small number of files with big size rather than a large

number of files with small size. If the size of a file is smaller than the typical block size in Hadoop,

we consider it as a small file. Due to this, a number of metadata increases which will become an overhead to the NameNode.

To solve this problem sequence files are introduced in Hadoop.

Sequence files act as a container to store the small files.

Sequence files are flat files consisting of binary key-value pairs. When Hive converts queries to MapReduce jobs,it

decides on the appropriate key-value pairs to be used for a given record. Sequence files are in the binary format which

can be split and the main use of these files is to club two or more smaller files and make them as a one sequence file.

In Hive we can create a sequence file by specifying STORED AS SEQUENCEFILE in the end of a CREATE TABLE statement.

There are three types of sequence files:

• Uncompressed key/value records.

• Record compressed key/value records - only 'values' are compressed here

• Block compressed key/value records - both keys and values are collected in 'blocks' separately and compressed.

The size of the 'block' is configurable.

Hive has its own SEQUENCEFILE reader and SEQUENCEFILE writer libraries for reading and writing through sequence files.

In Hive we can create a sequence file format as follows:

create table table\_name (schema of the table) row format delimited fileds terminated by ',' | stored as SEQUENCEFILE;

Hive uses the SEQUENCEFILE input and output formats from the following packages:

org.apache.hadoop.mapred.SequenceFileInputFormat

org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat

Creating SEQUENCEFILE

create table olympic\_sequencefile(athelete 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 sequencefile;

Here we are creating a table with name olympic\_sequencefile and the schema of the table is as specified above and the data inside my input file is delimited by tab space. At the end the file format is specified as SEQUENCEFILE format. You can check the schema of your created table using:

describe olympic\_sequencefile;

\*\*\*\*\*\*

Now to load data into this table is somewhat different from loading into the table created using TEXTFILE format.

You need to insert the data from another table because this SEQUENCEFILE format is the binary format.

It compresses the data and then stores it into the table.

If you want to load directly as in TEXTFILE format that is not possible because we cannot insert the compressed files into tables.

So to load the data into SEQUENCEFILE we need to use the following approach:

INSERT OVERWRITE TABLE olympic\_sequencefile

SELECT \* FROM olympic;

We have already created table by name olympic which is of TEXTFILE format and then we are writing the contents of the olympic table into olympic\_sequencefile table.

Thus we have successfully loaded data into the SEQUENCEFILE.

Now let us perform the same basic SELECT operation which we have performed on the TEXTFILE format, on SEQUENCEFILE format also.

select athelete from olympic\_sequencefile;

-------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RCFILE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-------------------------------------------------------------

# 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.

# RCFILE is used when we want to perform operations on multiple rows at a time.

# RCFILEs are flat files consisting of binary key/value pairs, which shares many similarities with SEQUENCEFILE.

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.

Facebook uses RCFILE as its default file format for storing of data in their data warehouse as they perform different types of analytics using Hive.

In Hive we can create a RCFILE format as follows:

create table table\_name (schema of the table) row format delimited fields terminated by ',' | stored as RCFILE

Hive has its own RCFILE Input format and RCFILE output format in its default package:

org.apache.hadoop.hive.ql.io.RCFileInputFormat

org.apache.hadoop.hive.ql.io.RCFileOutputFormat

\*\*\*\*\*\*\*\*\*\*Creating RCFILE\*\*\*\*\*\*\*\*\*\*\*(centos support)

create table olympic\_rcfile(athelete 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 rcfile

Here we are creating a table with name olympic\_rcfile and the schema of the table is as specified above.

# The data inside the input file is delimited by tab space.

# At the end the file format is specified as RCFILE format.

# You can check the schema of your created table using:

describe olympic\_rcfile;

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 as shown below:

INSERT OVERWRITE TABLE olympic\_rcfile

SELECT \* FROM olympic;

We have already created a table by name olympic which is of TEXTFILE format and then we are writing the contents of the olympic table into olympic\_rcfile table.

As shown above we have successfully loaded data into the RCFILE.

Now let us perform the same basic SELECT operation which we have performed on the TEXTFILE format on RCFILE format as well as shown in Figure below:

select athelete from olympic\_rcfile;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ORCFILE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* (centos not support)

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%(eg: 100GB file will become 25GB).

As a result the speed of data processing also increases.

ORC 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.

In Hive we can create a ORCFILE format as follows:

create table table\_name (schema of the table) row format delimited fields terminated by ',' | stored as ORC

Hive has its own ORCFILE Input format and ORCFILE output format in its default package:

org.apache.hadoop.hive.ql.io.orc

Creating ORCFILE

create table olympic\_orcfile(athelete 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 orcfile;

Here we are creating a table with name olympic\_orcfile and the schema of the table is as specified above. The data inside the input file is delimited by tab space.

At the end the file format is specified as ORCFILE format.

You can check the schema of your created table using:

describe olympic\_orcfile;

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.

INSERT OVERWRITE TABLE olympic\_orcfile

SELECT \* FROM olympic;

Here is a table created by name olympic which is of TEXTFILE format and then we need to write the contents of the olympic table into olympic\_orcfile table.

Thus we have successfully loaded data into the ORCFILE.

Now let us perform the same basic SELECT operation which we have performed on the TEXTFILE format, on ORCFILE format as well.

select athelete from olympic\_orcfile;

Thus you can use the above four file formats depending on your data.

For example,

------------------------------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*IMPORTANT POINTS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

------------------------------------------------------------------------------------------

a) If your data is delimited by some parameters( comma,tab,space,json) then you can use TEXTFILE format.

b) If your data is in small files whose size is less than the block size(64 MB IN VERSION 1 AND 128 MB IN HADOOP 2.X)

then you can use SEQUENCEFILE format.

c) If you want to perform analytics on your data and you want to store your data efficiently for that then you can

use RCFILE format.

d) If you want to store your data in an optimized way which lessens your storage and increases your performance then you

can use ORCFILE format.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Avro\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

---------------------------------------------------------------------

Widely used as a serialization platform

Row-based, offers a compact and fast binary format

Schema is encoded on the file so the data can be untagged

Files support block compression and are splittable

Supports schema evolution

----------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Parquet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

----------------------------------------------------------------------

Column-oriented binary file format

Uses the record shredding and assembly algorithm described in the Dremel paper

Each data file contains the values for a set of rows

Efficient in terms of disk I/O when specific columns need to be queried

NOTE :

IF YOUR USE CASE TYPICALLY SCANS OR RETRIVES ALL OF THE FEILDS IN A ROW IN EACH QUERY,AVRO IS USUALLY THE BEST CHOICE.

IF YOUR DATASET HAS MANY COLUMNS AND YOUR USE CASE TYPICALLY INVOLVES WORKING WITH A SUBSET OF THOSE COLUMNS RATHER

THAN ENTIRE RECORDS,PARQUET IS optimized for that kind of work.

hive> select \* from salesdata01 where from\_unixtime(unix\_timestamp(Order\_date, 'dd-MM-yyyy'),'yyyy-MM-dd') >= from\_unixtime(unix\_timestamp('2010-09-01', 'yyyy-MM-dd'),'yyyy-MM-dd') and from\_unixtime(unix\_timestamp(Order\_date, 'dd-MM-yyyy'),'yyyy-MM-dd') <= from\_unixtime(unix\_timestamp('2011-09-01', 'yyyy-MM-dd'),'yyyy-MM-dd') limit 10;

OK

1 3 13-10-2010 Low 6.0 261.54 0.04 Regular Air -213.25 38.94

80 483 10-07-2011 High 30.0 4965.7593 0.08 Regular Air 1198.97 195.99

97 613 17-06-2011 High 12.0 93.54 0.03 Regular Air -54.04 7.3

98 613 17-06-2011 High 22.0 905.08 0.09 Regular Air 127.7 42.76

103 643 24-03-2011 High 21.0 2781.82 0.07 Express Air -695.26 138.14

127 807 23-11-2010 Medium 45.0 196.85 0.01 Regular Air -166.85 4.28

128 807 23-11-2010 Medium 32.0 124.56 0.04 Regular Air -14.33 3.95

160 995 30-05-2011 Medium 46.0 1815.49 0.03 Regular Air 782.91 39.89

229 1539 09-03-2011 Low 33.0 511.83 0.1 Regular Air -172.88 15.99

230 1539 09-03-2011 Low 38.0 184.99 0.05 Regular Air -144.55 4.89

Time taken: 0.166 seconds, Fetched: 10 row(s)

hive> select \* from salesdata01 where from\_unixtime(unix\_timestamp(Order\_date, 'dd-MM-yyyy'),'yyyy-MM-dd') >= from\_unixtime(unix\_timestamp('2010-09-01', 'yyyy-MM-dd'),'yyyy-MM-dd') and from\_unixtime(unix\_timestamp(Order\_date, 'dd-MM-yyyy'),'yyyy-MM-dd') <= from\_unixtime(unix\_timestamp('2010-12-01', 'yyyy-MM-dd'),'yyyy-MM-dd') limit 10;

OK

1 3 13-10-2010 Low 6.0 261.54 0.04 Regular Air -213.25 38.94

127 807 23-11-2010 Medium 45.0 196.85 0.01 Regular Air -166.85 4.28

128 807 23-11-2010 Medium 32.0 124.56 0.04 Regular Air -14.33 3.95

256 1792 08-11-2010 Low 28.0 370.48 0.04 Regular Air -5.45 13.48

381 2631 23-09-2010 Low 27.0 1078.49 0.08 Regular Air 252.66 40.96

656 4612 19-09-2010 Medium 9.0 89.55 0.06 Regular Air -375.64 4.48

769 5506 07-11-2010 Critical 22.0 129.62 0.05 Regular Air 4.41 5.88

1457 10499 16-11-2010 Not Specified 29.0 6250.936 0.01 Delivery Truck 31.21 262.11

1654 11911 10-11-2010 Critical 25.0 397.84 0.0 Regular Air -14.75 15.22

2323 16741 30-09-2010 Medium 6.0 157.97 0.01 Regular Air -42.38 22.84

Time taken: 0.17 seconds, Fetched: 10 row(s)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Mapjoin\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Map joins are really efficient if a table on the other side of a join is small enough to fit in the memory.

This is similar to DistributedCache in MapReduce framework.

set hive.auto.convert.join=true;

------------------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Vectoriation\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

------------------------------------------------------------------------------

Vectorization allows Hive to process a batch of rows together instead of processing one row at a time.

Vectorization improves the performance by fetching 1,024 rows in a single operation instead of fetching single

row each time. It improves the performance for operations like filter, join, aggregation, etc.

set hive.vectorized.execution.enabled=true;

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

---------------------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ORC filefromat\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

---------------------------------------------------------------------------------

Optimized Row Columnar format provides highly efficient ways of storing the hive data by reducing the data storage format

by 75% of the original. The ORCFile format is better than the Hive files format when it comes to reading, writing, and

processing the data. It uses techniques like predicate push-down,compression, and more to improve the performance of the

query.

ORC supports compressed (ZLIB and Snappy), as well as uncompressed storage.

Create table orctbl (id int, name string, address string) stored as ORC tblproperties (“orc.compress”= “SNAPPY”);

----------------------------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Cost-Based Query Optimization\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

----------------------------------------------------------------------------------------

set hive.cbo.enable=true;

set hive.compute.query.using.stats=true;

set hive.stats.fetch.column.stats=true;

set hive.stats.fetch.partition.stats=true;

------------------------------------------------------------------------------------------

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Enable compression in Hive\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

------------------------------------------------------------------------------------------

Compression techniques reduce the amount of data being transferred and so reduces the data transfer between mappers and

reducers.For better result, you need to perform compression at both mapper and reducer side separately.

Although gzip is considered as the best compression format but beware that it is not splittable and so should be applied

with caution.

Other formats are snappy, lzo, bzip, etc. You can set compression at mapper and reducer side using codes below-

set mapred.compress.map.output = true;

set mapred.output.compress= true;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*So let’s start with Hive performance tuning techniques!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

==================================================================================

1. Use Tez to Fasten the execution

========================================

Apache TEZ is an execution engine used for faster query execution.

It fastens the query execution time to around 1x-3x times.

To use TEZ execution engine, you need to enable it instead of default Map-Reduce execution engine.

TEZ can be enabled using the below query-

Set hive.execution.engine=tez;

-------------------------------------------------------------------------------------

2. Enable compression in Hive

======================================

Compression techniques reduce the amount of data being transferred and so reduces the data transfer between mappers and

reducers.For better result, you need to perform compression at both mapper and reducer side separately.

Although gzip is considered as the best compression format but beware that it is not splittable and so should be applied

with caution.

Other formats are snappy, lzo, bzip, etc. You can set compression at mapper and reducer side using codes below-

set mapred.compress.map.output = true;

set mapred.output.compress= true;

---------------------------------------------------------------------------------------------------

3. Use ORC file format

=============================

ORC (optimized record columnar) is great when it comes to hive performance tuning. We can improve the query performance

using ORC file format easily. You can check Hadoop file formats in detail here.There is no barrier like in which table you

can use ORC file and in response, you get faster computation and compressed file size.

It is very easy to create ORC table, and you just need to add STORED AS ORC command as shown below.

Syntax:

Create table orctbl (id int, name string, address string) stored as ORC tblproperties (“orc.compress”= “SNAPPY”);

Now simply you can also insert the data like-

Insert overwrite table orctbl select \* from tbldetails;

-----------------------------------------------------------------------------------------------------------

4. Optimize your joins

=====================

If you are using joins to fetch the results, it’s time to revise it. If you have large data in the tables,

then it is not advisable to just use normal joins we use in SQL. There are many other joins like Map Join; bucket joins etc.

which can be used to improve Hive query performance.

You can do the following with joins to optimize hive queries-

4.1. Use Map Join

==========================

Map join is highly beneficial when one table is small so that it can fit into the memory.

Hive has a property which can do auto-map join when enabled. Set the below parameter to true to enable auto map join.

Set hive.auto.convert.join to true to enable the auto map join.

You can either set this from the command line or from the hive-site.xml file.

<property>

<name>hive.auto.convert.join</name>

<value>true</value>

<description>Whether Hive enables the optimization about converting common join into mapjoin based on the input file size</description>

</property>

4.2.Use Skew Join

=============================

Skew join is also helpful when your table is skewed. Set the hive.optimize.skewjoin property to true to enable skew join.

set hive.optimize.skewjoin

<property>

<name>hive.optimize.skewjoin</name>

<value>true</value>

<description>

Whether to enable skew join optimization. The algorithm is as follows: At runtime, detect the keys with a large skew.

Instead of processing those keys, store them temporarily in an HDFS directory. In a follow-up 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.

</description> </property>

4.3Bucketed Map Join

==============================

If tables are bucketed by a particular column, you can use bucketed map join to improve the hive query performance.

You can set the below two property to enable the bucketed map join in Hive.

<property>

<name>hive.optimize.bucketmapjoin</name>

<value>true</value>

<description>Whether to try bucket mapjoin</description>

</property>

<property>

<name>hive.optimize.bucketmapjoin.sortedmerge</name>

<value>true</value>

<description>Whether to try sorted bucket merge map join</description>

</property>

--------------------------------------------------------------------------------------------------

5. Use partition

=====================

Partition is a useful concept in Hive. It is used to divide the large table based on certain column so that the whole data

can be divided into small chunks. It allows you to store the data under sub-directory inside a table.

Selecting the partition table is always a critical decision, and you need to take care of future data as well as the volume

of data as well. For example, if you have data of a particular location then partition based on state can be one of the

ideal choices.

Here is the syntax to create partition table-

CREATE TABLE countrydata\_partition

(Id int, countryname string, population int, description string)

PARTITIONED BY (country VARCHAR(64), state VARCHAR(64))

row format delimited

fields terminated by ‘\t’

stored AS textfile;

There are two types of partition in Hive-

Static partition

Dynamic partition

Static partition is the default one. To use dynamic partition in Hive, you need to set the following property-

set hive.exec.dynamic.partition=true;

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

----------------------------------------------------------------------------------------------------------------

6. Bucketing can also be used

==========================

If you have more number of columns on which you want the partitions, bucketing in the hive can be a better option.

We use CLUSTERED BY command to divide the tables in the bucket.

Here is the syntax to create bucketed table-

CREATE TABLE emp\_bucketed\_table(

ID int, name string, address string, salary string )

COMMENT ‘this is a bucketed table’

PARTITIONED BY (country VARCHAR(64))

CLUSTERED BY (state) INTO 10 BUCKETS

STORED AS TEXTFILE;

To enable bucketing in Hive, you need to set the following property-

SET hive.enforce.bucketing=true;

This should be set every time you are writing the data to the bucketed table.

----------------------------------------------------------------------------------------------

7. Parallel execution:

==========================

As we know, Hive converts the queries into different stages during execution. These stages are usually getting executed one

after the other and thus increases the time of execution. Below are some of the normal steps involved-

• MapReduce stage

• Sampling stage

• Limit stage

• Merge stage etc.

But the good thing is, you can set some of this independent stage to process parallel. This is a parallel execution in Hive. For this, you need to set the below properties to true-

Set hive.exec.parallel = true;

-------------------------------------------------------------------------------------------------

10. Avoid Global sorting

=============================

Global sorting in Hive is getting done by the help of the command ORDER BY in the hive. But the issue is, if you’re using

ORDER BY command, then the number of reducers will be set to one which can be illogical when you have large Hadoop dataset.

So when you don’t need global sorting, use SORT BY command which sorts the result per reducer.

Even you can also use DISTRIBUTE BY command if you want to control which particular rows will go with which reducer.

10.1 DISTRIBUTE BY

============================

It ensures each of N reducers gets non-overlapping ranges of column, but doesn’t sort the output of each reducer.

You end up with N or more unsorted files with non-overlapping ranges.

Example

We are Distributing By x on the following 5 rows to 2 reducer:

x1

x2

x4

x3

x1

1

2

3

4

5

x1

x2

x4

x3

x1

Reducer 1

x1

x2

x1

1

2

3

x1

x2

x1

Reducer 2

x4

x3

1

2

x4

x3

Note that all rows with the same key x1 is guaranteed to be distributed to the same reducer (reducer 1 in this case),

but they are not guaranteed to be clustered in adjacent positions.

10.2 CLUSTER BY:

=====================

Cluster By is a short-cut for both Distribute By and Sort By.

CLUSTER BY x ensures each of N reducers gets non-overlapping ranges, then sorts by those ranges at the reducers.

Ordering : Global ordering between multiple reducers.

Outcome : N or more sorted files with non-overlapping ranges.

For the same example as above , if we use Cluster By x, the two reducers will further sort rows on x:

Reducer 1 :

x1

x1

x2

1

2

3

x1

x1

x2

Reducer 2 :

x3

x4

1

2

x3

x4

Instead of specifying Cluster By, the user can specify Distribute By and Sort By, so the partition columns and sort columns

can be different.

10.3 Sort by:

======================

Hive uses SORT BY to sort the rows based on the given columns per reducer. If there are more than one reducer, then the

output per reducer will be sorted, but the order of total output is not guaranteed to be sorted.

set hive.mapred.reduce.tasks=2;

select \* from emp sort by sal;

10.4 ORDER BY

Order by guarantees the total ordering of the output. Even if there are multiple reducers, the overall order of the output

is maintained.

select \* from emp orderby sal;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Compressions in hive \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SET hive.exec.compress.output=true;

SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;

SET mapred.output.compression.type=BLOCK;

SET hive.exec.compress.output=true;

SET mapred.output.compression.codec=org.apache.hadoop.io.compress.SnappyCodec;

SET mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec;

SET mapred.output.compression.codec=org.apache.hadoop.io.compress.DefaultCodec;

SET mapred.output.compression.codec=org.apache.hadoop.io.compress.Bzip2Codec;

SET mapred.output.compression.codec=org.apache.hadoop.io.compress.lzo.LzopCodec;

10.5 File Formats:

HIVE STORAGE FORMATS

Among the different storage file formats that are used in hive, the default and simplest storage file format is the TEXTFILE.

10.5.1 TEXTFILE

The data in a TEXTFILE is stored as plain text, one line per record. The TEXTFILE is very useful for sharing data with

other tools and also when you want to manually edit the data in the file. However the TEXTFILE is less proficient when

compared to the other formats.

SYNTAX :

CREATE TABLE TEXTFILE\_TABLE (

COLUMN1 STRING,

COLUMN2 STRING,

COLUMN3 INT,

COLUMN4 INT

) STORED AS TEXTFILE;

10.5.2 SEQUENCE FILE

In sequence files the data is stored in a binary storage format consisting of binary key value pairs. A complete row is

stored as single binary value. Sequence files are more compact than text and fit well the map-reduce output format. Sequence

files do support block compression and can be compressed on value, or block level, to improve its IO profile further.

SEQUENCEFILE is a standard format that is supported by Hadoop itself and is good choice for Hive table storage especially

when you want to integrate Hive with other techonolgies in the Hadoop ecosystem.

The USING sequence file keywords lets you create a sequence File. Here is an example statement to create a table using

sequence File:

CREATE TABLE SEQUENCEFILE\_TABLE (

COLUMN1 STRING,

COLUMN2 STRING,

COLUMN3 INT,

COLUMN4 INT

) STORED AS SEQUENCEFILE

Due to the complexity of reading sequence files, they are often only used for “in flight” data such as intermediate data

storage used within a sequence of MapReduce jobs.

10.5.3 RCFILE OR RECORD COLUMNAR FILE:

=-======================================

The RCFILE is one more file format that can be used with Hive. The RCFILE stores columns of a table in a record columnar

format rather than row oriented fashion and provides considerable compression and query performance benefits with highly

efficient storage space utilization. Hive added the RCFile format in version 0.6.0.

RC file format is more useful when tables have large number of columns but only few columns are typically retrieved.

The RCFile combines multiple functions to provide the following features

Fast data storing

Improved query processing,

Optimized storage space utilization

Dynamic data access patterns.

SYNTAX:

CREATE TABLE RCFILE\_TABLE (

COLUMN1 STRING,

COLUMN2 STRING,

COLUMN3 INT,

COLUMN4 INT ) STORED AS RCFILE;

Compressed RCFile reduces the IO and storage significantly over text, sequence file, and row formats. Compression on a

column base is more efficient here since it can take advantage of similarity of the data in a column.

10.5.4 ORC FILE OR OPTIMIZED ROW COLUMNAR FILE:

=================================================

ORCFILE stands for Optimized Row Columnar File and it’s a new Hive File Format that was created to provide many advantages

over the RCFILE format while processing data. The ORC File format comes with the Hive 0.11 version and cannot be used with

previous versions.

Lightweight indexes are included with ORC file to improve the performance.

Also it uses specific encoders for different column data types to improve compression further, e.g. variable length

compression on integers ORC stores collections of rows in one file and within the collection the row data is stored in a

columnar format allowing parallel processing of row collections across a cluster.

ORC files compress better than RC files, enabling faster queries. To use it just add STORED AS orc to the end of your

create table statements like this:

CREATE TABLE mytable (

COLUMN1 STRING,

COLUMN2 STRING,

COLUMN3 INT,

COLUMN4 INT

) STORED AS orc;

========================WINDOWING FUNTIONS=================

Windowing allows you to create a window on a set of data further allowing aggregation surrounding that data. Windowing in Hive is introduced from Hive 0.11. In this blog, we will be giving a demo on the windowing functions available in Hive.

Windowing in Hive includes the following functions

* Lead
  + The number of rows to lead can optionally be specified. If the number of rows to lead is not specified, the lead is one row.
  + Returns null when the lead for the current row extends beyond the end of the window.
* Lag

The number of rows to lag can optionally be specified. If the number of rows to lag is not specified, the lag is one row.

Returns null when the lag for the current row extends before the beginning of the window.

* FIRST\_VALUE
* LAST\_VALUE

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.

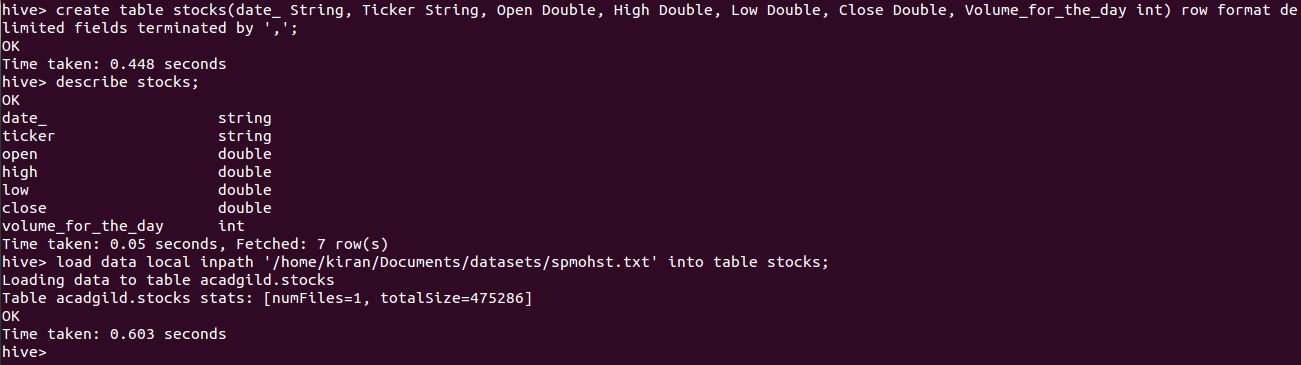
Analytics functions

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

To give you a brief idea of these windowing functions in Hive, we will be using stock market data. You can download the sample stocks data [from here](https://drive.google.com/open?id=0ByJLBTmJojjzbVhvSnQwNlhXUWs) and load into your stocks table.

Now we will create a table to load this stock market data as shown below.

create table stocks (date\_ String, Ticker String, Open Double, High Double, Low Double, Close Double, Volume\_for\_the\_day int) row format delimited fields terminated by ',';



Let us dive deeper into the window functions in Hive.

**Lag**

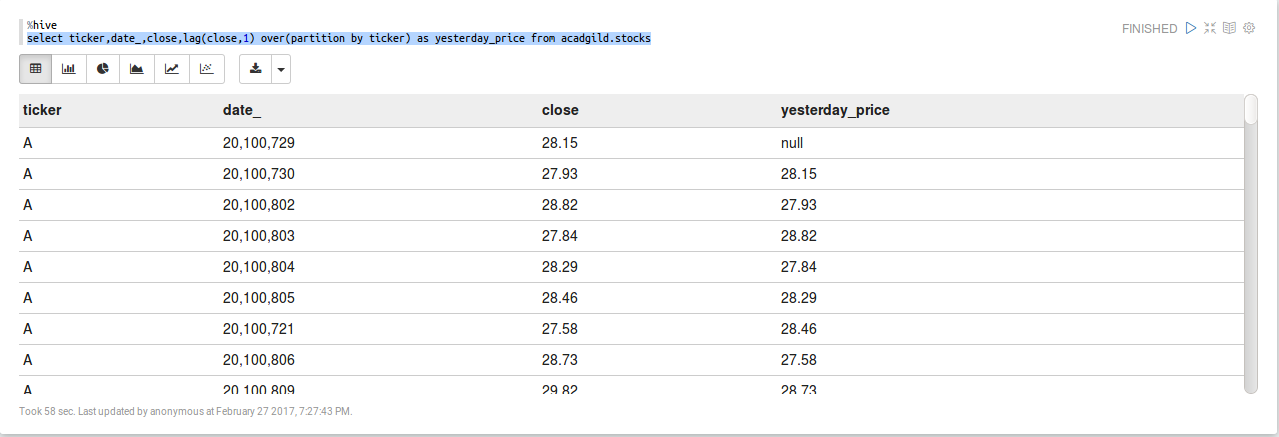
This function returns the values of the previous row. You can specify an integer offset which designates the row position else it will take the default integer offset as 1.

Here is the sample function for ***lag***

select ticker,date\_,close,lag(close,1) over(partition by ticker) as yesterday\_price from acadgild.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.

In the below screenshot, you can see the closing price of the stock for the day and the yesterday’s price.



**Lead**

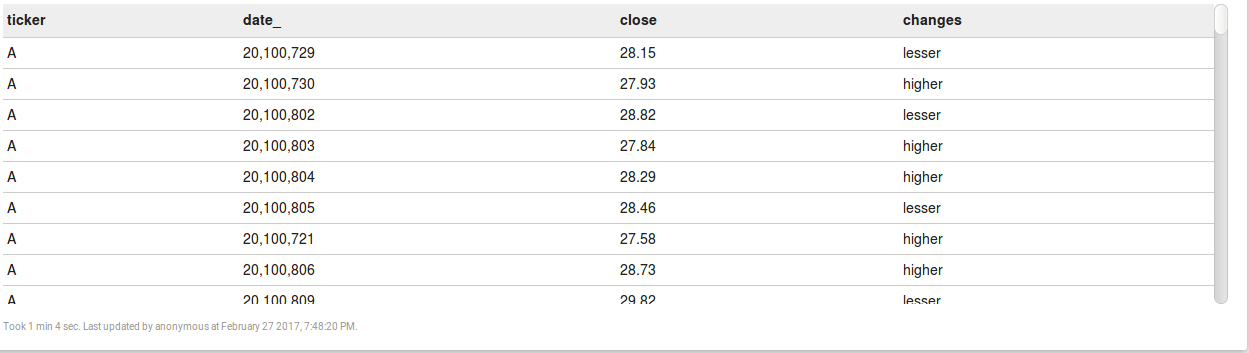
This function returns the values from the following rows. You can specify an integer offset which designates the row position else it will take the default integer offset as 1.

Here is the sample function for ***lead***

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 acadgild.stocks

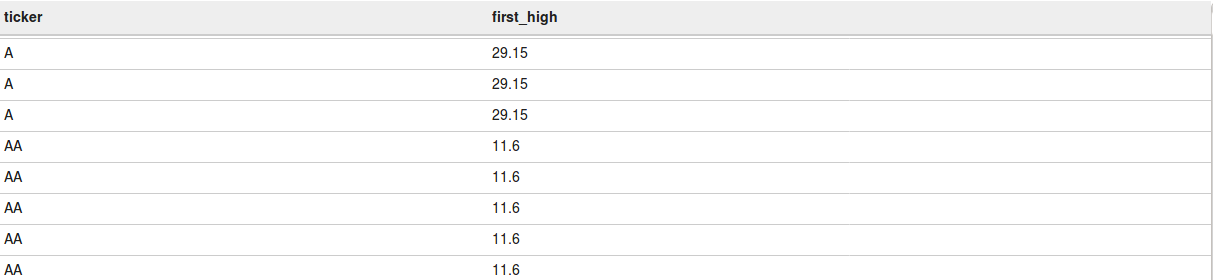
In the below screenshot, you can see the result.



**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.

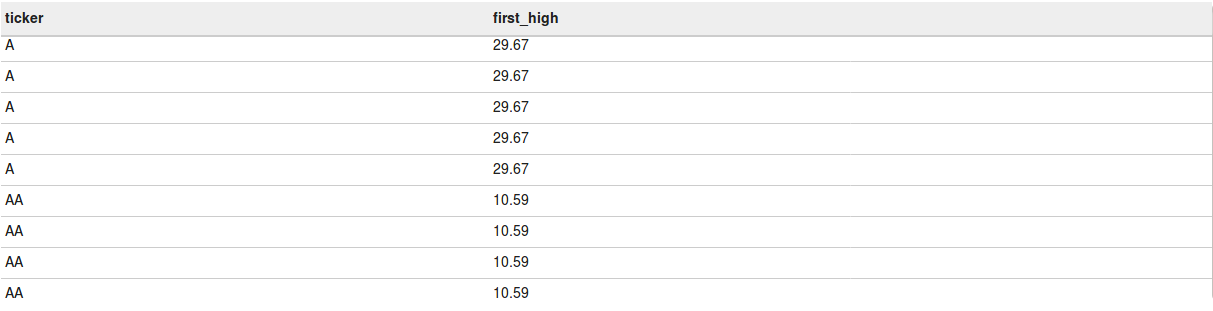
select ticker,first\_value(high) over(partition by ticker) as first\_high from acadgild.stocks



**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 acadgild.stocks



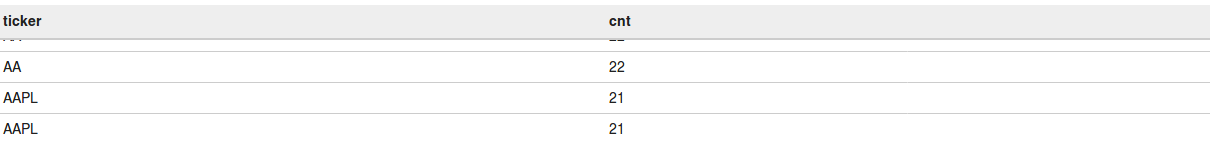
Let us now see the usage of the aggregate function using Over.

**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 acadgild.stocks

For each partition, the count of ticker will be calculated, you can see the same in the below screen shot.

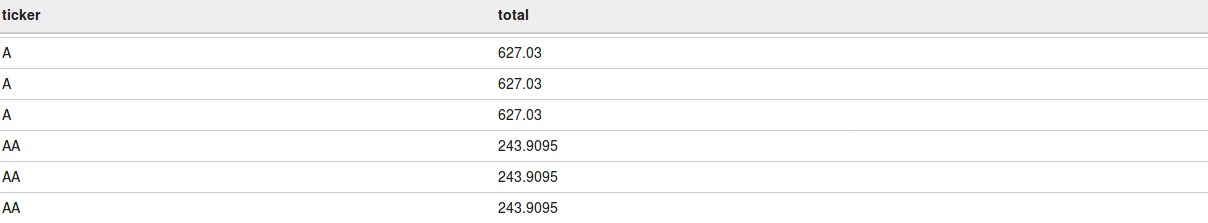


**Sum**

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 acadgild.stocks

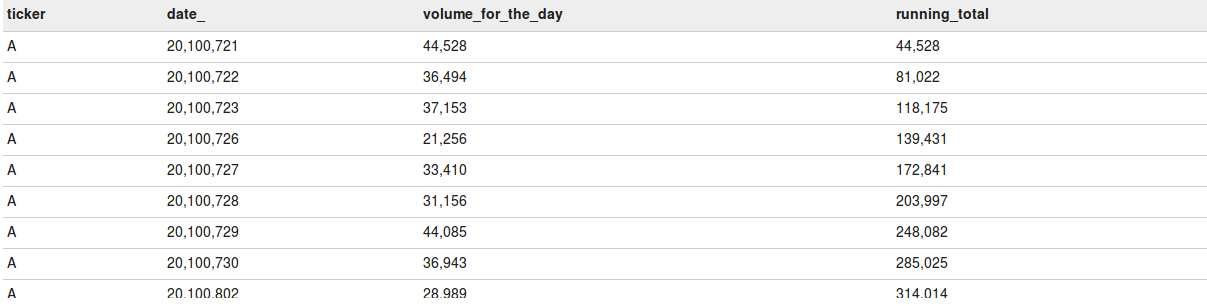
For each ticker, the sum of all the closing prices will be calculated, you can see the same in the below screen shot.



**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 acadgild.stocks

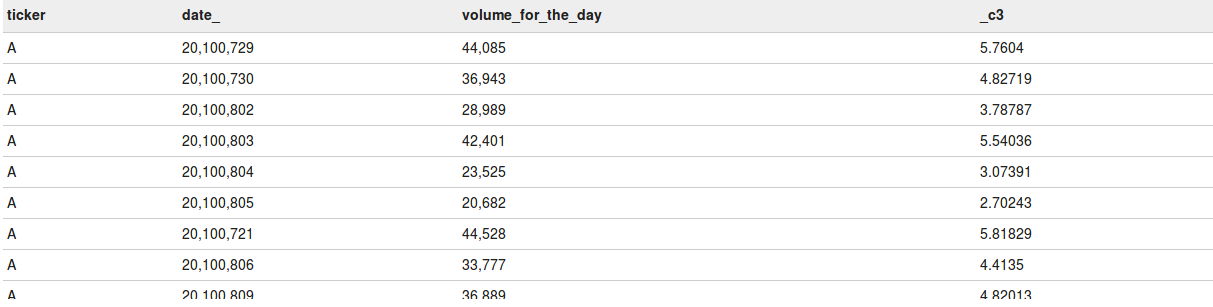


In the above screenshot, you can see the volume\_for\_the\_day for each day and the running total is the sum of volume\_for\_the\_day’s that are elapsed.

**Finding the percentage of each row value**

Now let’s take a scenario where you need to find the percentage of the volume\_for\_the\_day on the total volumes for that particular ticker and that can be done as follows.

select ticker,date\_,volume\_for\_the\_day,(volume\_for\_the\_day\*100/(sum(volume\_for\_the\_day) over(partition by ticker))) from acadgild.stocks

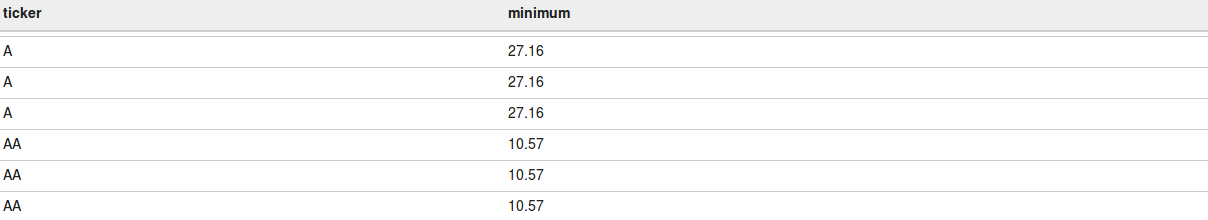


In the above screenshot, you can see that the percentage contribution of the volumes for the day is found based on the total volume for that ticker.

**Min**

It returns the minimum value of the column for the rows in that over clause. From the below query, we can find the minimum closing stock price for each particular ticker.

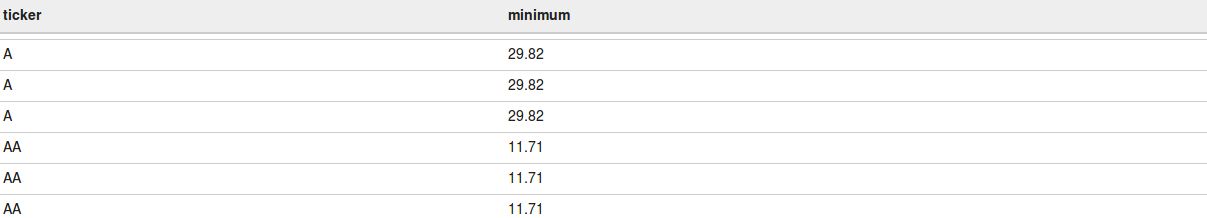
select ticker, min(close) over(partition by ticker) as minimum from acadgild.stocks



**Max**

It returns the maximum value of the column for the rows in that over clause. From the below query, we can find the maximum closing stock price for each particular ticker.

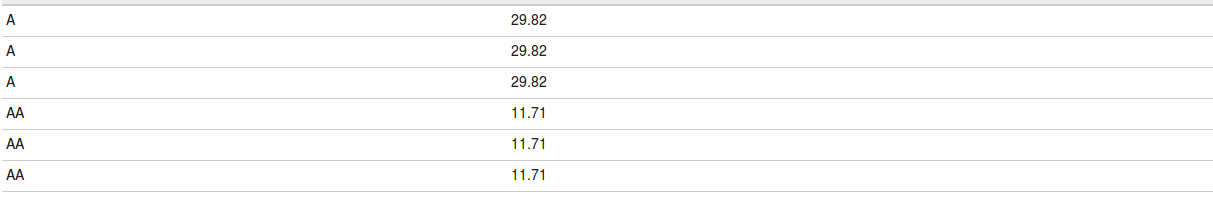
select ticker, max(close) over(partition by ticker) as maximum from acadgild.stocks



**AVG**

It returns the average value of the column for the rows that over clause returns. From the below query, we can find the average closing stock price for each particular ticker.

select ticker, avg(close) over(partition by ticker) as maximum from acadgild.stocks



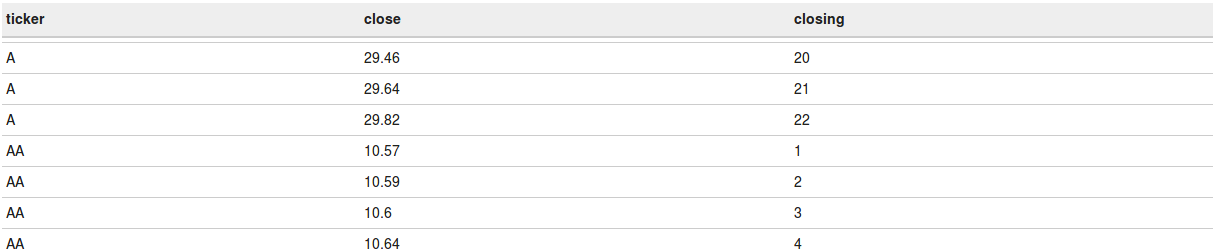
Now let us work on some Analytic functions.

**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 acadgild.stocks



**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(partition by ticker order by close) as num from acadgild.stocks

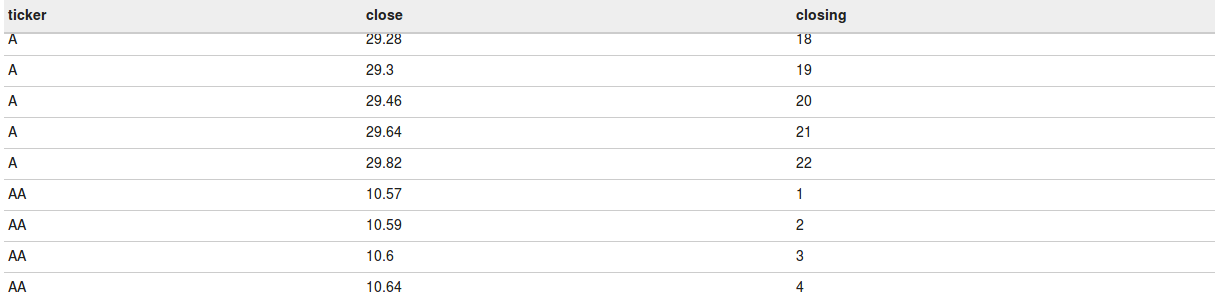


**Dense\_rank**

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 acadgild.stocks

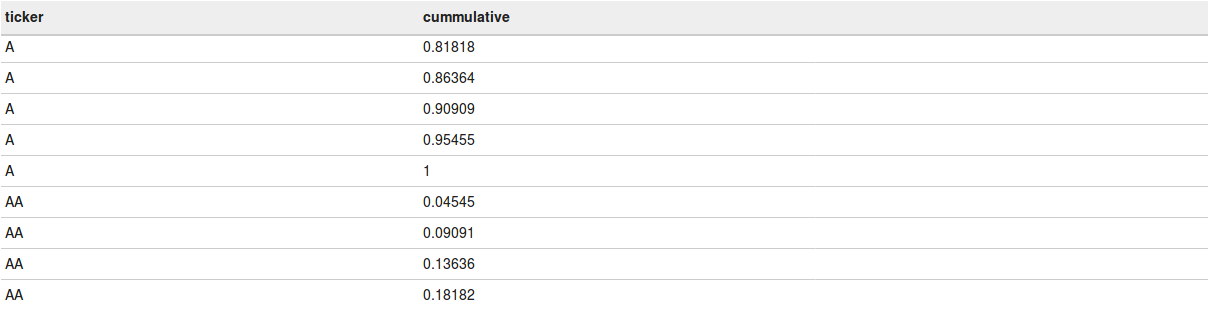


**Cume\_dist**

It returns the cumulative distribution of a value. It results from 0 to 1. For suppose if the total number of records are 10 then for the 1st row the cume\_dist will be 1/10 and for the second 2/10 and so on till 10/10.

This cume\_dist will be calculated in accordance with the result set returned by the over clause. The below query will result in the cumulative of each record for every ticker.

select ticker,cume\_dist() over(partition by ticker order by close) as cummulative from acadgild.stocks

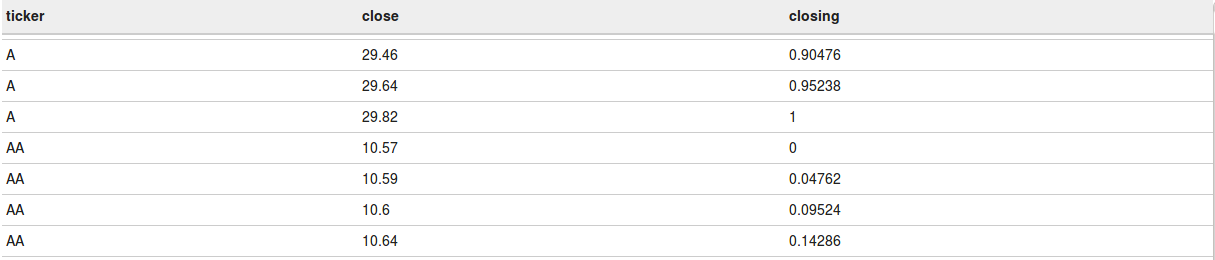


**Percent\_rank**

It returns the percentage rank of each row within the result set of over clause. Percent\_rank is calculated in accordance with the rank of the row and the calculation is as follows (rank-1)/(total\_rows\_in\_group – 1). If the result set has only one row then the percent\_rank will be 0.

The below query will calculate the percent\_rank for every row in each partition and you can see the same in the below screen shot.

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



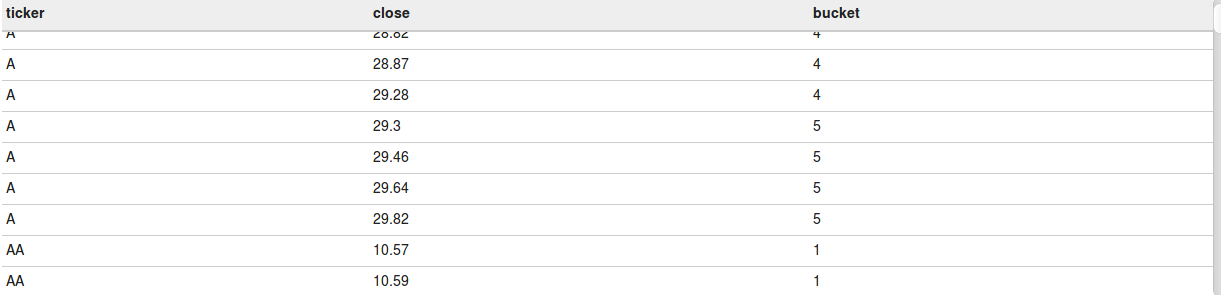
**Ntile**

It returns the bucket number of the particular value. For suppose if you say Ntile(5) then it will create 5 buckets based on the result set of the over clause after that it will place the first 20% of the records in the 1st bucket and so on till 5th bucket.

The below query will create 5 buckets for every ticker and the first 20% records for every ticker will be in the 1st bucket and so on.

select ticker,ntile(5) over(partition by ticker order by close ) as bucket from acadgild.stocks

In the below screenshot, you can see that 5 buckets will be created for every ticker and the least 20% closing prices will be in the first bucket and the next 20% will be in the second bucket and so on till 5th bucket for all the tickers.



This is how we can perform windowing operations in Hive.