Lab 10: Hive Tables

This lab explores how Hive table data is stored in HDFS.

Objective: Understand how Hive table data is stored in HDFS.

File locations: ~/data

Successful outcome: A new Hive table filled with the data from the `wh_visits` folder. You'll also learn how to use the `!sh` as a shell script inside Hive. You'll also learn how external tables work with Hive.

Before you begin: Complete the Preparing Data for Hive lab, or put the data from the solution of that lab into HDFS. Remember paths in the files need to be /user/hive/warehouse and /user/cloudera/

---

1. Review the Data

   1.1. Use the hadoop command to view the contents of the `/user/hive/warehouse/wh_visits` folder in HDFS that was created in an earlier lab. You should see two `part-m` files:

   ```
   # hdfs dfs -ls -R /user/hive/warehouse/wh_visits/
   ```

   1.2. Recall that the Pig projection to create these files had the following schema (do not input this - it is reprinted for reference only):

   ```
   project_potus = FOREACH potus GENERATE
   $0  AS  lname:chararray,
   $1  AS  fname:chararray,
   $6  AS  time_of_arrival:chararray,
   $11 AS  appt_scheduled_time:chararray,
   $21 AS  location:chararray,
   $25 AS  comment:chararray ;
   ```

   In this lab, you will define a Hive table that matches these records and contains the exported data from your Pig script.

2. Define a Hive Script

   2.1. In the Understanding data folder, there is a text file named `wh_visits.hive`. View its contents. Notice that it defines a Hive table named `wh_visits` with the following schema that matches the data in your `project_potus` folder:

```
# cd ~/data
# more wh_visits.hive

create table wh_visits (
   lname string,
   fname string,
   time_of_arrival string,
   appt_scheduled_time string,
   meeting_location string,
   info_comment string)
ROW  FORMAT  DELIMITED
FIELDS TERMINATED BY '\t' ;
```

2.2. Run the script with the following command:

```
# hive -f wh_visits.hive
```

2.3. If successful, you should see “OK” in the output along with the time it took to run the query. If not, check that you added sandbox to the /etc/hosts file successfully, see Lab 1 Step 9.

3. Verify the Table Creation

3.1. Start the Hive Shell, using the new `beeline` client:

```
# beeline -n hive -u jdbc:hive2://quickstart:10000
```

Note: beeline is the new client that replaces hive - for purposes of this course, we’ll use the hive for brevity

3.2. From the hive> prompt, enter the “show tables” command:

```
10000> show tables;
```

You should see `wh_visits` in the list of tables.

3.3. Use the describe command to view the details of `wh_visits`:

```
10000> describe wh_visits;
INFO  : OK
+----------------------+------------+----------+--+
|       col_name       | data_type  | comment  |
+----------------------+------------+----------+--+
| lname                | string     |          |
| fname                | string     |          |
| time_of_arrival      | string     |          |
| appt_scheduled_time  | string     |          |
| meeting_location     | string     |          |
| info_comment         | string     |          |
+----------------------+------------+----------+--+
6 rows selected (0.343 seconds)
```

3.4. Try running a query (even though the table is empty):

```
10000> select * from wh_visits limit 20;
```

You should see 20 rows returned. How is this brand new Hive table already populated with records?

Answer: In a previous lab, you already populated the `/user/hive/warehouse/wh_visits` folder with the output of a Pig job.

3.5. Why did the previous query not require a Tez or MapReduce job to execute?

Answer: The query selected all columns and did not contain a WHERE clause, the query just needs to read in the data from the file and display it.

4. Count the Number of Rows in a Table

   4.1. Enter the following Hive query, which outputs the number of rows in `wh_visits`:

```
 10000> select count(*) from wh_visits;
```

   4.2. How many rows are currently in wh_visits?

   Answer: 21,819

5. Selecting the Input file name

   5.1. Hive has two virtual columns that get created automatically for every table:

   `INPUT__FILE__NAME` and `BLOCK__OFFSET__INSIDE__FILE`

   Note: between each word in the column name there are two underscore characters, not just one. You must make sure you type both of them when using these columns in a hive command.

   You can use these column names in your queries just like any other column of the table. To demonstrate, run the following query:

```
 10000> select INPUT__FILE__NAME, lname, fname FROM wh_visits WHERE lname LIKE
'Y%';
```

5.2. The result of this query is visitors to the White House whose last name starts with "Y." Notice that the output also contains the particular file that the record was found in (beware of the port):

```
hdfs://quickstart:8020/user/hive/warehouse/wh_visits/part-m-00000 | YOUNG |
MICHELLE
hdfs://quickstart:8020/user/hive/warehouse/wh_visits/part-m-00001 | YOUNG |
LEDISI
...
```

6. Drop a Table

6.1. Let's see what happens when a managed table is dropped. Start by defining a simple table called `names` using the Hive Shell:

```
10000> create table names (id int, name string) ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t';
```

6.2. Use the Hadoop dfs command (with a `!sh`) to put `names.txt` into the table's warehouse folder:

```
10000> !sh hdfs dfs -put /home/cloudera/data/names.txt
/user/hive/warehouse/names/names.txt
```

Note: `!sh` only works in beeline - you can see other `!` commands with a `!help`. Note also there is no `;` at the end of a shell command!

6.3. View the contents of the table's warehouse folder:

```
10000> !sh hdfs dfs -ls /user/hive/warehouse/names
Found 1 items
cloudera hdfs    78 /user/hive/warehouse/names/names.txt
```

6.4. From the Hive Shell, run the following query:

```
10000> select * from names;
OK
0    Rich
1    Barry
2    George
3    Ulf
4    Danielle
5    Tom
6    manish
7    Brian
8    Mark
```

6.5. Now drop the `names` table:

```
10000> drop table names;
```

6.6. View the contents of the table's warehouse folder again. Notice the names folder is gone:

```
hive> !sh hdfs dfs -ls /user/hive/warehouse/names
ls: '/user/hive/warehouse/names': No such file or directory
```

NOTE: Be careful when you drop a managed table in Hive. Make sure you either have the data backed up somewhere else or that you no longer want the data.

7. Define an External Table

7.1. In this step you will see how external tables work in Hive. Start by putting `names.txt` into HDFS:

```
10000> !sh hdfs dfs -put /home/cloudera/data/names.txt names.txt
```

7.2. Create a folder in HDFS for the external table to store its data in:

```
10000> !sh hdfs dfs -mkdir hivedemo
```

7.3. Define the names table as external this time:

```
10000> create external table names (id int, name string) ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t' LOCATION '/user/cloudera/hivedemo';
```

7.4. Load data into the table:

```
10000> load data inpath '/user/cloudera/names.txt' into table names;
```

7.5. Verify that the load worked:

```
10000> select * from names;
```

7.6. Notice the `names.txt` file has been moved to `/user/cloudera/hivedemo`:

```
10000> !sh hdfs dfs -ls -R hivedemo
Found 1 items
-rw-r--r--   3 cloudera hdfs   78 hivedemo/names.txt
```

7.7. Similarly, verify that `names.txt` is no longer in your `/user/cloudera` folder in HDFS.

```
10000> !sh hdfs dfs -ls -R /user/cloudera
```

Why is it gone?

Answer: The LOAD command moved the file from `/user/cloudera` to `/user/cloudera/names`. The LOAD command does not copy files; it moves them.

7.8. Use the hdfs shell command to verify that the `/user/hive/warehouse` folder does not contain a subfolder for the names table.

7.9. Now drop the names table.

7.10. View the contents of `/user/cloudera/hivedemo`. Notice that `names.txt` is still there.

Result

As you just verified, the data for external tables is not deleted when the corresponding table is dropped. Aside from this behavior, managed tables and external tables in Hive are essentially the same. You now have a table in Hive named wh_visits that was loaded from the result of a Pig job. You also have an external table called names that stores its data in `/user/cloudera/hivedemo`. At this point, you should have a pretty good understanding of how Hive tables are created and populated.

Remember paths in the files need to be /user/hive/warehouse and /user/cloudera/

Lab 11: Partitions and Skew

This lab explores how Hive partitioning and skewed tables work.

Objective: To understand how Hive partitioning and skewed tables work.

File locations: `~/data`

---

1. View the data

1.1. Review the `hivedata_<state>.txt` files in `~/data`. This will be the data added to the table.

2. Define the Table in Hive

2.1. View the create table statement in `partitiondemo.sql`:

```
# more partitiondemo.sql
drop table if exists names;
create table names (id int, name string) partitioned by (state string)
     row format delimited fields terminated by '\t';
```

2.2. Run the query to define the names table:

```
# hive -f partitiondemo.sql

# hive
```

2.3. Show the partitions (there won't be any yet):

```
hive> show partitions names;
```

3. Load Data into the Table

3.1. When you load data into a partitioned table, you specify which partition the data goes into. For example:

```
hive> load data local inpath '/home/cloudera/data/hivedata_ca.txt' into table
names partition (state = 'CA');
```

3.2. Load the CO and SD files also:

```
hive> load data local inpath '/home/cloudera/data/hivedata_co.txt' into table
names partition (state = 'CO');
hive> load data local inpath '/home/cloudera/data/hivedata_sd.txt' into table
names partition (state = 'SD');
```

3.3. Verify that all of the data made it into the `names` table:

```
hive> select * from names;
OK
1    Ulf CA
2    Manish  CA
3    Brian   CA
4    George  CO
5    Mark    CO
6    Rich    SD
```

4. View the Directory Structure

4.1. View the contents of `/user/hive/warehouse/names`:

```
hive> dfs -ls -R /user/hive/warehouse/names/;
0    /user/hive/warehouse/names/state=CA
24  /user/hive/warehouse/names/state=CA/hivedata_ca.txt
0    /user/hive/warehouse/names/state=CO
16  /user/hive/warehouse/names/state=CO/hivedata_co.txt
0    /user/hive/warehouse/names/state=SD
6    /user/hive/warehouse/names/state=SD/hivedata_sd.txt
```

4.2. Notice that each partition has its own subfolder for storing its contents.

5. Perform a Query

5.1. When you specify a where clause that includes a partition, Hive is smart enough to only scan the files in that partition.

For example:

```
hive> select * from names where state = 'CA';
OK
1   Denise  CA
2   Manish  CA
3   Brian   CA
```

5.2. Notice that a MapReduce job was not executed. Why?

Answer: The result of the query is exactly the contents of the underlying files, so there is no need to run a MapReduce job. The files can simply be read and displayed.

5.3. You can select the partition field, even though it is not actually in the data file. Hive uses the directory name to retrieve the value:

```
hive> select name, state from names where state = 'CA';
```

5.4. You can still run queries across the entire dataset. For example, the following query spans multiple partitions:

```
hive> select name, state from names where state = 'CA' or state = 'SD';
```

When you are done, use quit to exit the Hive shell:

```
hive> quit;
```

6. Now create a skewed table

6.1. Verify the existence of the `salaries.txt` folder in `~/data/` and then put it into the `/user/cloudera/salarydata/` folder in HDFS. Remember paths in the files need to be /user/hive/warehouse and /user/cloudera/ :

```
# ls salaries.txt
# hdfs dfs -put salaries.txt salarydata/salaries.txt
```

6.2. View the contents of `skewdemo.hive`, which defines a skewed table named `skew_demo` using the `salaries.txt` data. Remember paths in the files need to be /user/hive/warehouse and /user/cloudera/:

```
# more skewdemo.hive
```

6.3. Which values are skewing this table?

Answer: The skewed values are the 95102 and 94040 zip codes.

Run the `skewdemo.hive` script:

```
# hive -f skewdemo.hive
```

6.4. View the contents of the underlying Hive warehouse folder:

```
# hdfs dfs -ls -R /user/hive/warehouse/skew_demo
```

6.5. Select a few records to make sure the table has data behind it:

```
# hive -f show_skewdemo.hive
```

Result

You now should have a better understanding of partitions and skew in Hive.

Lab 12: Analyzing Data with Hive

This lab explores techniques to analyze Big Data, using public visitor data from the White House.

Objective: Analyze the White House visitor data

File locations: /data

Successful outcome: You will have discovered several useful pieces of information about the White House visitor data.

Before you begin: Complete the Hive Tables Lab.

---

1. Find the first visit

   1.1. Using vi, gedit or a similar editor, create a new text file named `whitehouse.hive` and save it in your data folder.

   1.2. In this step, you will instruct the hive script to find the first visitor to the White House (based on our dataset). This will involve some clever handling of timestamps. This will be a long query, so enter it on multiple lines (note the lack of a ";" at the end of this first step).

   Start by selecting all columns where the time_of_arrival is not empty:

```
select * from wh_visits where time_of_arrival != ""
```

   1.3. To find the first visit, we need to sort the result. This requires converting the time_of_arrival string into a timestamp. We will use the unix_timestamp function to accomplish this. Add the following order by clause (again, no ";" at the end of the line):

```
order by unix_timestamp(time_of_arrival, 'MM/dd/yyyy hh:mm')
```

   1.4. Since we are only looking for one result, we certainly don't need to return every row. Let's limit the result to 10 rows, so we can view the first 10 visitors (this finishes the query, so will end with the ";" character):

```
limit 10;
```

   1.5. Save your changes to `whitehouse.hive`.

   1.6. From data dir, execute the script `whitehouse.hive` and wait for the results to be displayed:

```
# cd ~/data
# hive -f whitehouse.hive
```

1.7. The results should be 10 visitors, and the first visit should be in 2009, since that is when the dataset begins. The first visitors are Charles Kahn and Carol Keehan on 3/5/2009.

2. Find the Last Visit

2.1. This one is easy: just take the previous query and reverse the order by adding `desc`to the `order by` clause:

```
order by unix_timestamp(time_of_arrival, 'MM/dd/yyyy hh:mm') desc
```

2.2. Run the query again, and you should see that the most recent visit was Jackie Walker on 3/18/2011.

```
# hive -f whitehouse.hive
```

3. Find the most common comment

3.1. In this step, you will explore the info_comment field and try to determine the most common comment. You will use some of Hive's aggregate functions to accomplish this.

Start by using gedit (or editor of your choice) to create a new text file named `comments.hive` and save it in `~/data` folder.

3.2. You will now create a query that displays the 10 most frequently occurring comments. Start with the following select clause:

```
from wh_visits
select count(*) as comment_count, info_comment
```

This runs the aggregate count function on each group (which you will define later in the query) and names the result `comment_count`. For example, if "OPEN HOUSE" occurs five times then `comment_count` will be five for that group.

Notice: we are also selecting the info_comment column so we can see what the comment is.

3.3. Group the results of the query by the info_comment column:

```
group by info_comment
```

3.4. Order the results by comment_count, because we are only interested in comments that appear most frequently:

```
order by comment_count DESC
```

3.5. We only want the top results, so limit the result set to 10:

```
limit 10;
```

3.6. Save your changes to comments.hive and execute the script. Wait for the MapReduce job to execute.

```
# hive -f comments.hive
```

3.7. The output will be 10 comments and should look like:

```
9036
1253    HOLIDAY BALL ATTENDEES/
894 WHO EOP RECEP 2
700 WHO EOP 1 RECEPTION/
601 RESIDENCE STAFF HOLIDAY RECEPTION/
586 PRESS RECEPTION ONE (1)/
580 GENERAL RECEPTION 1
540 HANUKKAH RECEPTION./
540 GEN RECEP 5/
516 GENERAL RECEPTION 3
```

3.8. It appears that a blank comment is the most frequent comment, followed by the `HOLIDAY BALL`, then a variation of other receptions.

3.9. Modify the query so that it ignores empty comments. If it works, the comment “GEN RECEP 6/” will show up in your output.

Solution:

In `comments.hive`, insert the following line between your select and group statements:

```
where info_comment != ""
```

Save the changes, then back at the command line, re-run the query:

```
# hive -f comments.hive
```

4. Least Frequent Comment

4.1. Run the previous query again, but this time, find the 10 least occurring comments.

Remove DESC from your order statement so that it looks like this:

```
order by comment_count
```

Save the changes, then back at the command line, re-run the query:

```
# hive -f comments.hive
```

The output should look like:

```
1       CONGRESSIONAL BALL/
1       CONG BALL/
1       merged to u59031 CONGRESSIONAL BALL CONG BALL
1       COMMUNITY COLLEGE SUMMIT
1       48 HOUR WAVE EXCEPTION GRANTED DROP BY VISIT
1       WHO EOP/
1       "POTUS LUNCH WITH WASHINGTON
```

This seems accurate since 1 is the least number of times a comment can appear.

5. Analyze the Data Inconsistencies

   5.1. Analyzing the results of the most- and least-frequent comments, it appears that several variations of GENERAL RECEPTION occur. In this step, you will try to determine the number of visits to the POTUS involving a general reception by trying to clean up some of these inconsistencies in the data.

   NOTE: Inconsistencies like these are very common in big data, especially when human input is involved. In this dataset, we likely have different people entering similar comments but using their own abbreviations.

   5.2. Modify the query in comments.hive. Instead of searching for empty comments.

   Search for comments that contain variations of the string “GEN RECEP.”

```
where info_comment rlike '.*GEN.*\\s+RECEP.*'
```

   5.3. Change the limit clause from 10 to 30:

```
limit 30;
```

   5.4. Run the query again.

```
# hive -f comments.hive
```

5.5. Notice there are several GENERAL RECEPTION entries that only differ by a number at the end or use the GEN RECEP abbreviation:

```
580 GENERAL RECEPTION   1
540 GEN RECEP 5/
516 GENERAL RECEPTION   3
498 GEN RECEP 6/
438 GEN RECEP 4
31  GENERAL RECEPTION   2
23  GENERAL RECEPTION   3
20  GENERAL RECEPTION   6
20  GENERAL RECEPTION   5
13  GENERAL RECEPTION   1
```

5.6. Let's try one more query to try and narrow GENERAL RECEPTION visit. Modify the WHERE clause in comments.hive to include "%GEN%":

```
where info_comment like "%RECEP%" and info_comment like "%GEN%"
```

5.7. Leave the limit at 30, save your changes, and run the query again.

```
# hive -f comments.hive
```

5.8. The output this time reveals all the variations of GEN and RECEP. Next, let's add up the total number of them by running the following query:

```
from wh_visits
select count(*)
where info_comment like "%RECEP%"
and info_comment like "%GEN%";
```

Then save your changes and run the query again from the command line:

```
# hive -f comments.hive
```

5.9. Notice there are 2,697 visits to the POTUS with `GEN RECEP` in the comment field, which is about 12% of the 21,819 total visits to the POTUS in our dataset.

NOTE: More importantly, these results show that the conclusion from our first query, where we found that the most likely reason to visit the President was the HOLIDAY BALL with 1,253 attendees, is incorrect. This type of analysis is common in big data, and it shows how data analysts need to be creative and thorough when researching their data.

6. Verify the Result

6.1. We have 12% of visitors to the POTUS going for a general reception, but there were a lot of statements in the comments that contained WHO and EOP. Modify the query from the last step and display the top 30 comments that contain “WHO” and “EOP.”

You should be able to undo changes to `comments.hive` and restore it to the state before the last lab. Then make the following two additional edits:

Change the where clause to match WHO and EOP:

```
where info_comment like "%WHO%"
and info_comment like "%EOP%";
```

Add the DESC command back to the end of the order statement:

```
order by comment_count DESC
```

Finally, double-check select count(*) as comment_count, info_count. Make sure the “as…” portion is there.

Then save your changes and run the query again from the command line:

```
# hive -f comments.hive
```

The result should look like:

```
894 WHO EOP RECEP 2
700 WHO EOP 1 RECEPTION/
43  WHO EOP RECEP/
20  WHO EOP HOLIDAY RECEP/
13  WHO/EOP #2/
8   WHO EOP RECEPTION
7   WHO EOP RECEP
1   WHO EOP/
1   WHO EOP RECLEAR
```

6.2. Modify the script again, this time to run a query that counts the number of records with `WHO` and `EOP` in the comments, and run the query:

```
from wh_visits
select count(*)
where info_comment like "%WHO%" and info_comment like "%EOP%";
```

Run the query from the command line:

```
# hive -f comments.hive
```

You should get 1,687 visits, or 7.7% of the visitors to the POTUS. So GENERAL RECEPTION still appears to be the most frequent comment.

7. Find the Most Visits

   7.1. See if you can write a Hive script that finds the top 20 individuals who visited the POTUS most. Use the Hive command from Step 3 earlier in this lab as a guide. Tip: use a groupng by both fname and lname.

   7.2. The following script will accomplish the intention of the previous step:

```
from wh_visits
select count(*) as most_visit, fname, lname group by fname, lname
order by most_visit DESC limit 20;
```

   To verify that your script worked, here are the top 20 individuals who visited the POTUS along with the number of visits (your output may vary slightly due to randomization of names):

```
16   ALAN    PRATHER
15   CHRISTOPHER FRANKE
15   ROBERT      BOGUSLAW
...
```

8. Typical solution is found in the `comments.hive.soln` and `whitehouse.hive.soln` files.

Result

You have written several Hive queries to analyze the White House visitor data. The goal is for you to become comfortable with working with Hive, so hopefully you now feel like you can tackle a Hive problem and be able to answer questions about your big data stored in Hive.

Lab 13: MapReduce and Hive

This lab explores how Hive queries get executed as MapReduce jobs.

Objective: To understand better how Hive queries get executed as MapReduce jobs.

File locations: n/a

Successful outcome: No specific outcome. You will answer various questions about Hive queries and run a few examples.

Before you begin: Complete the Understanding Hive Table lab and start the Hive Shell.

---

1. The Describe Command

   1.1. Log into the Hive shell and run the describe command on the wh_visits table:

```
hive> describe wh_visits;
OK
lname   string
fname   string
time_of_arrival string
appt_scheduled_time
string meeting_location string
info_comment    string
```

   1.2. Did this query require a Tez or MapReduce job?

   Answer: No

   1.3. What is the name of the Hive resource that was accessed to retrieve this schema information?

   Answer: The Hive metastore contains the schema information of all tables.

2. A Simple Query

   2.1. Run the following query:

```
hive> select * from wh_visits where fname = "JOE";
```

   2.2. Open your browser and point it to the JobHistory UI: http://sandbox:19888/

   2.3. Notice that it did not execute any job.

3. A Sorted Query

   3.1. Run the following query:

```
hive> select * from wh_visits where fname = "JOE" sort by lname;
```

3.2. When the MapReduce job completes, find its job details page from the Job Browser.

3.3. How many map tasks were used to execute this query?

Answer: One map task

3.4. How many reduce tasks were used to execute this query?

Answer: One reduce task

3.5. The map task outputs <key,value> pairs and sends them to the reducer. What do you think this MapReduce job chose as the key for the mapper's output?

Answer: It makes sense for the mapper to use lname as the key, which would mean the visitors would already be sorted by last name when they got to the reducer.

4. Using the EXPLAIN Command

   4.1. The EXPLAIN command shows the execution plan of a query without actually executing the query. To demonstrate, add EXPLAIN to the beginning of the following query that you ran earlier in this lab:

```
hive> explain select * from wh_visits where fname = "JOE" sort by lname;
```

   4.2. Notice Stage-1 of the DAG has one mapper (look for Map Operator Tree) and one reducer (under Reduce Operator Tree). As you can see from this execution plan, the mapper is doing most of the work.

5. Use EXPLAIN EXTENDED

   5.1. Run the previous EXPLAIN again, except this time add the EXTENDED command:

```
hive> explain extended select * from wh_visits where fname = "JOE" sort by
lname;
```

   5.2. Compare the two outputs. Notice the EXTENDED command adds a lot of additional information about the underlying execution plan.

Result

You should now know how a mapreduce job executes in Hive.

Lab 14: Computing ngrams with Hive

This demonstration explores how to compute ngrams using Hive.

Objective: To understand how to compute ngrams using Hive.

---

1. Create a Hive Table for the Data

   1.1. This demonstration computes ngrams on the U.S. Constitution, which is in a text file in the data folder:

   ```
   # more constitution.txt
   ```

   Press q to exit more

   1.2. Start the Hive shell and define the following table:

   ```
   hive> create table constitution ( line string ) ROW FORMAT DELIMITED;
   ```

   Each line of text in the text file is going to be a record in our Hive table.

2. Load the Hive Table

   2.1. Load constitution.txt into the constitution table:

   ```
   hive> load data local inpath '/home/cloudera/data/constitution.txt' into
   table constitution;
   ```

   2.2. Verify that the data is loaded:

   ```
   hive> select * from constitution;
   ```

   You should see the contents of `constitution.txt` again.

3. Compute a Bigram

   3.1. Enter the following Hive command, which computes a bigram for the Constitution and shows the top 15 results:

   ```
   hive> select explode(ngrams(sentences(line),2,15)) as x from constitution;
   ```

The result should look like:

```
{"ngram":["of","the"],"estfrequency":194.0}
{"ngram":["shall","be"],"estfrequency":100.0}
{"ngram":["the","United"],"estfrequency":76.0}
{"ngram":["United","States"],"estfrequency":76.0}
{"ngram":["to","the"],"estfrequency":57.0}
{"ngram":["shall","have"],"estfrequency":44.0}
{"ngram":["the","President"],"estfrequency":30.0}
{"ngram":["shall","not"],"estfrequency":29.0}
{"ngram":["in","the"],"estfrequency":28.0}
{"ngram":["by","the"],"estfrequency":25.0}
{"ngram":["the","Congress"],"estfrequency":22.0}
{"ngram":["and","the"],"estfrequency":21.0}
{"ngram":["for","the"],"estfrequency":21.0}
{"ngram":["Vice","President"],"estfrequency":21.0}
{"ngram":["the","Senate"],"estfrequency":21.0}
{"ngram":["States","and"],"estfrequency":20.0}
{"ngram":["States","shall"],"estfrequency":19.0}
{"ngram":["any","State"],"estfrequency":18.0}
{"ngram":["Congress","shall"],"estfrequency":18.0}
{"ngram":["on","the"],"estfrequency":17.0}
```

4. Compute a Trigram

4.1. Run the previous query again, but this time compute a trigram:

```
hive> select explode(ngrams(sentences(line),3,20)) as result from
constitution;
```

4.2. The result should look like:

```
{"ngram":["the","United","States"],"estfrequency":68.0}
{"ngram":["of","the","United"],"estfrequency":51.0}
{"ngram":["shall","not","be"],"estfrequency":16.0}
{"ngram":["of","the","Senate"],"estfrequency":14.0}
{"ngram":["States","shall","be"],"estfrequency":13.0}
{"ngram":["House","of","Representatives"],"estfrequency":13.0}
{"ngram":["United","States","shall"],"estfrequency":13.0}
{"ngram":["shall","have","been"],"estfrequency":12.0}
{"ngram":["the","several","States"],"estfrequency":12.0}
{"ngram":["President","of","the"],"estfrequency":11.0}
{"ngram":["United","States","and"],"estfrequency":11.0}
{"ngram":["The","Congress","shall"],"estfrequency":10.0}
{"ngram":["the","House","of"],"estfrequency":10.0}
{"ngram":["United","States","or"],"estfrequency":10.0}
{"ngram":["Congress","shall","have"],"estfrequency":10.0}
{"ngram":["the","Vice","President"],"estfrequency":9.0}
{"ngram":["of","the","President"],"estfrequency":8.0}
{"ngram":["Consent","of","the"],"estfrequency":8.0}
{"ngram":["shall","be","the"],"estfrequency":7.0}
{"ngram":["by","the","Congress"],"estfrequency":7.0}
```

5. Compute a Contextual ngram

5.1. Let's find the 20 most frequent words that follow "the":

```
hive> select explode(context_ngrams(sentences(line), array("the",null),20))
as result from constitution;
```

5.2. The result looks like:

```
{"ngram":["United"],"estfrequency":76.0}
{"ngram":["President"],"estfrequency":30.0}
{"ngram":["Congress"],"estfrequency":22.0}
{"ngram":["Senate"],"estfrequency":21.0}
{"ngram":["several"],"estfrequency":15.0}
{"ngram":["Vice"],"estfrequency":12.0}
{"ngram":["State"],"estfrequency":11.0}
{"ngram":["same"],"estfrequency":10.0}
{"ngram":["Constitution"],"estfrequency":10.0}
{"ngram":["States"],"estfrequency":10.0}
{"ngram":["House"],"estfrequency":10.0}
{"ngram":["whole"],"estfrequency":10.0}
{"ngram":["office"],"estfrequency":9.0}
{"ngram":["right"],"estfrequency":8.0}
{"ngram":["Legislature"],"estfrequency":8.0}
{"ngram":["Consent"],"estfrequency":6.0}
{"ngram":["powers"],"estfrequency":6.0}
{"ngram":["supreme"],"estfrequency":6.0}
{"ngram":["people"],"estfrequency":6.0}
{"ngram":["first"],"estfrequency":6.0}
```

Result

You will be able to see how explode() and array() work as examples in Hive.

Lab 15: Joining Datasets in Hive

This lab explores performing joins of two datasets in Hive.

Objective: Perform a join of two datasets in Hive.

File locations: ~/data

Successful outcome: A table named stock_aggregates that contains a join of NYSE stock prices with the stock's dividend prices.

Before you begin: Your HDP cluster should be up and running within your VM.

---

1. Load the Data into Hive

1.1. View the contents of the file `setup.hive` in your data folder:

```
# more setup.hive
```

1.2. Notice that this script creates three tables in Hive. The nyse_data table is filled with the daily stock prices of stocks that start with the letter K and the dividends table that contains the quarterly dividends of those stocks. The `stock_aggregates` table is going to be used for a join of these two datasets and contain the stock price and dividend amount on the date the dividend was paid.

1.3. Run the `setup.hive` script from the folder:

```
# hive -f setup.hive
```

1.4. To verify that the script worked, enter the Hive Shell and run the following following queries:

```
# hive
hive> select * from nyse_data limit 20;
hive> select * from dividends limit 20;
```

You should see daily stock prices and dividends from stocks that start with the letter K.

1.5. The `stock_aggregates` table should be empty, but view its schema to verify that it was created successfully, then type quit to exit the Hive Shell:

```
hive> describe stock_aggregates;
OK
symbol          string
year            string
high            float
low             float
average_close   float
total_dividends float

hive> quit
```

2. Join the Datasets

   2.1. The join statement is going to be fairly long, so let's create it in a text file. Use gedit (or similar editor) to create a new text in the data folder named `join.hive`.

   2.2. We will break the join statement down into sections. First, the result of the join is going to be put into the `stock_aggregates` table, which requires an insert:

```
insert overwrite table stock_aggregates
```

   The overwrite causes any existing data in `stock_aggregates` to be deleted.

   2.3. The data being inserted is going to be the result of a select query that contains various insightful indicators about each stock. The result is going to contain the stock symbol, date traded, maximum high for the stock, minimum low, average close, and the sum of dividends, as shown here:

```
select a.symbol, year(a.trade_date), max(a.high), min(a.low), avg(a.close),
sum(b.dividend)
```

   2.4. The from clause is the `nyse_data` table:

```
from nyse_data a
```

   2.5. The join is going to be a left outer join of the `dividends` table:

```
left outer join dividends b
```

   2.6. The join is by stock symbol and trade date:

```
on (a.symbol = b.symbol and a.trade_date = b.trade_date)
```

   2.7. Let's group the result by symbol and trade date:

```
group by a.symbol, year(a.trade_date);
```

   2.8. Save your changes to `join.hive`.

3. Run the Query

   3.1. Run the query and wait for the MapReduce jobs to execute:

```
# hive -f join.hive
```

3.2. How many jobs does it take to perform this query?

Answer: One MapReduce job with one mapper and one reducer.

4. Verify the Results

   4.1. Enter the Hive Shell and run a select query to view the contents of `stock_aggregates`:

```
hive> select * from stock_aggregates;
```

   The output should look like:

```
KYO 2004    90.9    66.25   75.79952    0.544
KYO 2005    78.45   62.58   72.042656   0.91999996
KYO 2006    98.01   71.73   85.80327    0.851
KYO 2007    110.01      81.0    93.737686   NULL
KYO 2008    100.78      45.41 79.6098   NULL
KYO 2009    93.2    52.98   77.04389    NULL
KYO 2010    93.83   85.94   90.71 NULL
stock_symbol    NULL    NULL    NULL    NULL    NULL
```

   4.2. List the contents of the `stock_aggregates` directory in HDFS. The `000000_0` file was created as a result of the join query:

```
hive> !sh hdfs dfs -ls -R /apps/hive/warehouse/stock_aggregates/
-rw-r--r--   3 root hdfs 41109
/apps/hive/warehouse/stock_aggregates/000000_0
```

   4.3. View the contents of the `stock_aggregates` table using the cat command:

```
hive> !sh hdfs dfs -cat /apps/hive/warehouse/stock_aggregates/000000_0
```

5. You can see a good example of the script in `join.hive.soln`.

Result

The `stock_aggregates` table is a joining of the daily stock prices and the quarterly dividend amounts on the date the dividend was announced, and the data in the table is an aggregate of various statistics like max high, min low, etc.

Lab 16: Computing ngrams of Emails in Avro Format

This lab explores using Hive to compute ngrams.

Objective: Use Hive to compute ngrams.

File locations: ~/data

Successful outcome: A bigram of words found in a collection of Avro-formatted emails.

Before you begin: Your HDP cluster should be up and running within your VM.

---

1. View an Avro Schema

   1.1. Change directories to the data folder. Notice this folder contains a file named `sample.avro`. Try to view the file using cat or more:

```
# cat sample.avro
```

   1.2. Because the file is binary formatted, reading it as above is not possible. You will need to download the latest version of avro to your /home/cloudera/Downloads folder using the browser link below:

```
# http://www-eu.apache.org/dist/avro/avro-1.7.7/java/avro-tools-1.7.7.jar
```

   1.3. Now enter the following command to view the schema of the contents of sample.avro:

```
 # java -jar ~/Downloads/avro-tools-1.7.7.jar getschema  sample.avro
```

   1.4. How many fields do records in `sample.avro` have?

   Answer: Four fields; name, age, address and value

   1.5. Create a schema file for `sample.avro`:

```
 # java -jar ~/Downloads/avro-tools-1.7.7.jar getschema  sample.avro >
sample.avsc
```

   1.6. Put the schema file into HDFS:

```
 # hdfs dfs -put sample.avsc
```

NOTE: If you get a checksum error, remove the hidden .sample.avsc.crc file from the /home/cloudera/data folder on the CentOS file system. Then try the above command again.

2. Create a Hive Table from an Avro schema

   2.1. View the contents of the CREATE TABLE query defined in the `create_sample_table.hive` file in your lab folder:

```
# more create_sample_table.hive
```

   2.2. Make sure the `avro.schema.file` property points to the schema file you created in the previous step… edit it to be /user/cloudera/data/sample.avsc as shown here:

```
WITH SERDEPROPERTIES (
'avro.schema.url'='hdfs:///user/cloudera/sample.avsc')
```

   2.3. Run the CREATE TABLE query:

```
# hive -f create_sample_table.hive
```

3. Verify the table

   3.1. Start the Hive shell, then run the show tables command and verify that you have a table named `sample_table`.

   3.2. Run the describe command on `sample_table`. Notice the schema for `sample_table`matches the Avro schema from `sample.avsc`.

```
hive> describe sample_table;
```

   3.3. Let's associate some data with `sample_table`. Copy `sample.avro` into the Hive warehouse folder by running the following command (all on a single line):

```
hive> !sh hdfs dfs -put /home/cloudera/data/sample.avro
/user/hive/warehouse/sample_table;
```

   3.5. View the contents of sample_table, then quit the Hive Shell:

```
hive> select * from sample_table;
OK
Foo 19  10, Bar Eggs Spam 800

hive> quit
```

NOTE: that there is only one record in `sample.avro`. You have now seen how to create a Hive table using an Avro schema file. This was a simple example; next you will complete these steps using a large data file that contains emails in an Avro format.

4. Create an Email-User table

4.1. There is an Avro file in your folder named `mbox7.avro`, which represents emails in an Avro format from a Hive mailing list for the month of July. Remember to use the `getschema` option of avro to view the schema of this file.

```
# java -jar ~/Downloads/avro-tools-1.7.7.jar getschema  mbox7.avro
```

4.2. How many fields do records in mbox7.avro have?

Answer: Four fields; sender, subject, date_sent and content

4.3. Run the `getschema` command again, but this time output the schema to a file named `mbox.avsc`:

```
# java -jar ~/Downloads/avro-tools-1.7.7.jar getschema  mbox7.avro >
mbox7.avsc
```

4.4. Put the Avro schema file into `/user/cloudera` in HDFS:

```
# hdfs dfs -put mbox7.avsc
```

4.5. Use more to view the contents of the `create_email_table.hive` script in your lab folder. Modify the `TBLPROPERTIES` to be ‘hdfs://sandbox.hortonworks.com:8020/user/root/mbox.avsc' to 'hdfs:///user/cloudera/mbox7.avsc' :

```
# more create_email_table.hive
```

4.6. Run the script to create the `hive_user_email` table:

```
# hive -f create_email_table.hive
```

4.7. Copy `mbox7.avro` into the `warehouse` directory:

```
# hdfs dfs -put mbox7.avro /user/hive/warehouse/hive_user_email
```

4.8. Start the Hive shell and verify the table has data in it:

```
hive> select * from hive_user_email limit 20;
```

5. Compute a Bigram

5.1. Use the Hive ngrams function to create a bigram of the words in `mbox7.avro`:

```
hive> select ngrams(sentences(content),2 ,10) from hive_user_email;
```

The output will be kind of a jumbled mess:

```
[{"ngram":["2013","at"],"estfrequency":802.0},{"ngram":["of","the"],"estfrequ
ency":391.0},{"ngram":["I","am"],"estfrequency":368.0},{"ngram":["I","have"],
"estfrequency":340.0},{"ngram":["J","E9r"],"estfrequency":306.0},{"ngram":["f
or","the"],"estfrequency":291.0},{"ngram":["you","are"],"estfrequency":289.0}
,{"ngram":["user","hive.apache.org"],"estfrequency":289.0},{"ngram":["to","th
e"],"estfrequency":276.0},{"ngram":["E9r","F4me"],"estfrequency":270.0}]
```

5.2. To clean this up, use the Hive explode function to display the output in a more readable format:

```
hive> select explode(ngrams(sentences(content),2 ,10)) from hive_user_email;
```

You should see a nice, readable list of 10 bigrams:

```
{"ngram":["2013","at"],"estfrequency":802.0}
{"ngram":["of","the"],"estfrequency":391.0}
{"ngram":["I","am"],"estfrequency":368.0}
{"ngram":["I","have"],"estfrequency":340.0}
{"ngram":["J","E9r"],"estfrequency":306.0}
{"ngram":["for","the"],"estfrequency":291.0}
{"ngram":["you","are"],"estfrequency":289.0}
{"ngram":["user","hive.apache.org"],"estfrequency":289.0}
{"ngram":["to","the"],"estfrequency":276.0}
{"ngram":["E9r","F4me"],"estfrequency":270.0}
```

5.3. Typically when working with word comparison we ignore case. Run the query again, but this time add the Hive `lower` function and compute 20 bigrams:

```
hive> select explode(ngrams(sentences(lower(content)),2 ,20)) from
`hive_user_email`;
```

The output should look like the following:

```
{"ngram":["2013","at"],"estfrequency":802.0}
{"ngram":["i","have"],"estfrequency":409.0}
{"ngram":["of","the"],"estfrequency":391.0}
```

```
{"ngram":["i","am"],"estfrequency":372.0}
{"ngram":["if","you"],"estfrequency":347.0}
{"ngram":["in","hive"],"estfrequency":337.0}
{"ngram":["for","the"],"estfrequency":309.0}
{"ngram":["j","e9r"],"estfrequency":306.0}
{"ngram":["you","are"],"estfrequency":289.0}
{"ngram":["user","hive.apache.org"],"estfrequency":289.0}
{"ngram":["to","the"],"estfrequency":276.0}
{"ngram":["outer","join"],"estfrequency":271.0}
{"ngram":["2013","06"],"estfrequency":270.0}
{"ngram":["e9r","f4me"],"estfrequency":270.0}
{"ngram":["left","outer"],"estfrequency":270.0}
{"ngram":["in","the"],"estfrequency":252.0}
{"ngram":["gmail.com","wrote"],"estfrequency":248.0}
{"ngram":["17","16"],"estfrequency":248.0}
{"ngram":["06","17"],"estfrequency":246.0}
{"ngram":["wrote","hi"],"estfrequency":234.0}
```

6. Compute a Context ngram

   6.1. From the Hive shell, run the following query, which uses the `context_ngrams` function to find the top 20 terms that follow the word "error":

```
hive> select explode(context_ngrams(sentences(lower(content)), array("error",
null) ,20)) from hive_user_email;
```

   The output should look like the following:

```
{"ngram":["in"],"estfrequency":102.0}
{"ngram":["return"],"estfrequency":97.0}
{"ngram":["org.apache.hadoop.hive.ql.exec.udfargumenttypeexception"],"estfreq
uency":49.0}
{"ngram":["failed"],"estfrequency":49.0}
{"ngram":["is"],"estfrequency":41.0}
{"ngram":["message"],"estfrequency":40.0}
{"ngram":["when"],"estfrequency":39.0}
{"ngram":["please"],"estfrequency":36.0}
{"ngram":["while"],"estfrequency":28.0}
{"ngram":["org.apache.thrift.transport.ttransportexception"],"estfrequency":2
8.0}
{"ngram":["datanucleus.plugin"],"estfrequency":26.0}
{"ngram":["during"],"estfrequency":18.0}
{"ngram":["query"],"estfrequency":16.0}
{"ngram":["hive"],"estfrequency":16.0}
{"ngram":["could"],"estfrequency":16.0}
{"ngram":["java.lang.runtimeexception"],"estfrequency":13.0}
{"ngram":["13"],"estfrequency":12.0}
{"ngram":["error"],"estfrequency":12.0}
{"ngram":["exec.execdriver"],"estfrequency":10.0}
{"ngram":["exec.task"],"estfrequency":10.0}
```

6.2. What is the most likely word to follow “error” in these emails?

Answer: “in”

6.3. Run a Hive query that finds the top 20 results for words in mbox7.avro that follow the phrase “error in.”

Solution:

```
select explode(context_ngrams(sentences(lower(content)), array("error", "in", null) ,20)) from hive_user_email;
```

Result

You have written several Hive queries that computed bigrams based on the data in the `mbox7.avro` file. You should also be familiar with working with Avro files, a popular file format in Hadoop.

Lab 17: Advanced Hive Queries

This lab explores some of the more advanced features of Hive work, including multi-table inserts, views, and windowing.

Objective: To understand how some of the more advanced features of Hive work, including multi-table inserts, views, and windowing.

File locations: ~/data

Successful outcome: You will have executed numerous Hive queries on customer order data.

Before you begin: Your CDH cluster should be up and running within your VM.

---

1. Create and Populate a Hive Table

1.1. From the command line, change directories to the data folder.

1.2. View the contents of the `orders.hive` file in that folder:

```
# more orders.hive
```

Notice it defines a Hive table named orders that has seven columns.

1.3. Execute the contents of orders.hive:

```
# hive -f orders.hive
```

1.4. From the Hive shell, verify that the script worked by running the following commands:

```
# hive
hive> describe orders;
hive> select count(*) from orders;
```

Your orders table should contain 99,999 records.

2. Analyze the Customer Data

2.1. Let's run a few queries to see what this data looks like.
Start by verifying that the `username` column actually looks like names:

```
hive> SELECT username FROM orders LIMIT 10;
```

You should see 10 first names.

2.2. The orders table contains orders placed by customers. Run the following query that shows the 10 lowest-price orders:

```
hive> SELECT username, ordertotal FROM orders ORDER BY ordertotal LIMIT 10;
```

The smallest orders are each $10, as you can see from the output:

```
Chelsea     10
Samantha    10
Danielle    10
Kimberly    10
Tiffany     10
Megan       10
Maria       10
Megan       10
Melissa     10
Christina   10
```

2.3. Run the same query, but this time use descending order:

```
hive> SELECT username, ordertotal FROM orders ORDER BY ordertotal DESC LIMIT
10;
```

The output this time is the 10 highest-priced orders:

```
Brandon     612
Mark        612
Sean        612
Jordan      612
Anthony     612
Paul        611
Jonathan    611
Eric        611
Nathan      611
Jordan      610
```

2.4. Let's find out if men or women spent more money:

```
hive> SELECT sum(ordertotal), gender FROM orders GROUP BY gender;
```

Based on the output, which gender has spent more money on purchases?

Answer: Men spent $9,919,847, and women spent $9,787,324.

2.5. The order_date column is a string with the format yyyy-mm-dd. Use the year function to extract the various parts of the date. For example, run the following query, which computes the sum of all orders for each year:

```
hive> SELECT sum(ordertotal), year(order_date) FROM orders GROUP BY
year(order_date);
```

The output should look like this. Verify, then quit the Hive shell:

```
4082780 2017
4404806 2014
4399886 2015
4248950 2016
2570769 2017

hive> quit;
```

3. Multi-File Insert

3.1. In this step, you will run two completely different queries, but in a single MapReduce job. The output of the queries will be in two separate directories in HDFS. Start by using gedit (or editor of your choice) to create a new text file in your lab folder named `multifile.hive`.

3.2. Within the text file, enter the following query. Notice there is no semicolon between the two INSERT statements:

```
FROM ORDERS o
INSERT OVERWRITE DIRECTORY '2017_orders'
```

```
SELECT o.* WHERE year(order_date) = 2017
INSERT OVERWRITE DIRECTORY 'software'
SELECT o.* WHERE itemlist LIKE '%Software%';
```

3.3. Save your changes to `multifile.hive`.

3.4. Run the query from the command line:

```
# hive -f multifile.hive
```

3.5. The above query executes in a single MapReduce job. Even more interesting, it only requires a map phase.

Why did this job not require a reduce phase?

Answer: Because the query only does a SELECT *, no reduce phase was needed.

3.6. Verify that the two queries executed successfully by viewing the folders in HDFS:

```
# hdfs dfs -ls
```

You should see two new folders: `2017_orders` and `software`.

3.7. View the output files in these two folders. Verify that the `2017_orders` directory contains orders from only the year 2017, and verify that the `software` directory contains only orders that included ‘Software.’

4. Define a View

4.1. Start the Hive shell. Define a view named `2016_orders` that contains the orderid, order_date, username, and itemlist columns of the `orders` table where the order_date was in the year 2016.

Solution: The `2016_orders` view:

```
# hive
hive> CREATE VIEW 2016_orders AS
SELECT orderid, order_date, username, itemlist FROM orders
WHERE year(order_date) = '2016';
```

4.2. Run the show tables command:

```
hive> show tables;
```

You should see `2016_orders` in the list of tables.

4.3. To verify your view is defined correctly, run the following query:

```
hive> SELECT COUNT(*) FROM 2016_orders;
```

The `2016_orders` view should contain around 21,544 records.

5. Find the Maximum Order of each customer

5.1. Suppose you want to find the maximum order of each customer. This can be done easily enough with the following Hive query.

Run this query now:

```
hive> SELECT max(ordertotal), userid FROM orders GROUP BY userid;
```

5.2. How many different customers are in the orders table?

Answer: There are 100 unique customers in the orders table.

5.3. Suppose you want to add the itemlist column to the previous query. Try adding it to the SELECT clause by the following method and see what happens:

```
hive> SELECT max(ordertotal), userid, itemlist FROM orders GROUP BY userid;
```

Notice this query is not valid because itemlist is not in the GROUP BY key.

5.4. We can join the result set of the max-total query with the orders table to add the itemlist to our result. Start by defining a view named max_ordertotal for the maximum order of each customer:

```
hive> CREATE VIEW max_ordertotal AS
SELECT max(ordertotal) AS maxtotal, userid FROM orders
GROUP BY userid;
```

5.5. Now join the orders table with your max_ordertotal view:

```
hive> SELECT ordertotal, orders.userid, itemlist FROM orders
JOIN max_ordertotal
ON max_ordertotal.userid = orders.userid
AND
max_ordertotal.maxtotal = orders.ordertotal
ORDER BY orders.userid;
```

5.6. The end of your output should look like:

```
 600 98
Grill,Freezer,Bedding,Headphones,DVD,Table,Grill,Software,Dishwasher,DVD,Micro
wave,Adapter
 600 99  Washer,Cookware,Vacuum,Freezer,2-Way Radio,Bicycle,Washer &
Dryer,Coffee Maker,Refrigerator,DVD,Boots,DVD
 600 100 Bicycle,Washer,DVD,Wrench Set,Sweater,2-Way
Radio,Pants,Freezer,Blankets,Grill,Adapter,pillows
```

6. Fixing the GROUP BY key error

6.1. Let's compute the sum of all of the orders of all of the customers. Start by entering the following query:

```
 hive> SELECT sum(ordertotal), userid FROM orders GROUP BY userid;
```

Notice that the output is the sum of all orders, but displaying just the userid is not very exciting.

6.2. Try to add the username column to the SELECT clause in the following manner and see what happens:

```
 hive> SELECT sum(ordertotal), userid, username FROM orders GROUP BY userid;
```

This generates the infamous "Expression not in GROUP BY key" error, because the username column is not being aggregated but the ordertotal is.

6.3. An easy fix is to aggregate the username values using the collect_set function, but output only one of them:

```
 hive> SELECT sum(ordertotal), userid, collect_set(username)[0] FROM orders
GROUP BY userid;
```

You should get the same output as before, but this time the username is included.

7. Using the OVER Clause

7.1. Now let's compute the sum of all orders for each customer, but this time use the OVER clause to not group the output and to also display the itemlist column:

```
 hive> SELECT userid, itemlist, sum(ordertotal) OVER (PARTITION BY userid)
FROM orders;
```

NOTE: the output contains every order, along with the items they purchased and the sum of all of the orders ever placed from that particular customer.

8. Using the Window Functions

8.1. It is not difficult to compute the sum of all orders for each day using the GROUP BY clause:

```
hive> select order_date, sum(ordertotal) FROM orders GROUP BY order_date;
```

Run the query above and the tail of the output should look like:

```
2017-07-28  18362
2017-07-29  3233
2017-07-30  4468
2017-07-31  4714
```

8.2. Suppose you want to compute the sum for each day that includes each order. This can be done using a window that sums all previous orders along with the current row:

```
hive> SELECT order_date, sum(ordertotal) OVER
(PARTITION BY order_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM orders;
```

To verify that it worked, your tail of your output should look like:

```
2017-07-31  3163
2017-07-31  3415
2017-07-31  3607
2017-07-31  4146
2017-07-31  4470
2017-07-31  4610
2017-07-31  4714
```

9. Using the Hive Analytics Functions

9.1. Run the following query, which displays the rank of the ordertotal by day:

```
hive> SELECT order_date, ordertotal, rank() OVER
(PARTITION BY order_date ORDER BY ordertotal) FROM orders;
```

9.2. To verify it worked, the output of July 31, 2017, should look like:

```
2017-07-31  48  1
2017-07-31  104 2
```

```
2017-07-31  119 3
2017-07-31  130 4
2017-07-31  133 5
2017-07-31  135 6
2017-07-31  140 7
2017-07-31  147 8
2017-07-31  156 9
2017-07-31  192 10
2017-07-31  192 10
2017-07-31  196 12
2017-07-31  240 13
2017-07-31  252 14
2017-07-31  296 15
2017-07-31  324 16
2017-07-31  343 17
2017-07-31  500 18
2017-07-31  528 19
2017-07-31  539 20
```

9.3. As a challenge, see if you can run a query similar to the previous one except compute the rank over months instead of each day.

Solution: The rank query by month:

```
SELECT substr(order_date,0,7), ordertotal, rank() OVER
(PARTITION BY substr(order_date,0,7) ORDER BY ordertotal) FROM orders;
```

10. Histograms

10.1. Run the following Hive query, which uses the histogram_numeric function to compute 20 (x,y) pairs of the frequency distribution of the total order amount from customers who purchased a microwave (using the orders table):

```
hive> SELECT explode(histogram_numeric(ordertotal,20))
AS x FROM orders
WHERE itemlist LIKE "%Microwave%";
```

The output should look like the following:

```
{"x":14.333333333333332,"y":3.0}
{"x":33.87755102040816,"y":441.0}
{"x":62.52577319587637,"y":679.0}
{"x":89.37823834196874,"y":965.0}
{"x":115.1242236024843,"y":1127.0}
{"x":142.6468885672939,"y":1382.0}
{"x":174.07664233576656,"y":1370.0}
{"x":208.06909090909105,"y":1375.0}
{"x":242.55486381322928,"y":1285.0}
{"x":275.8625954198475,"y":1048.0}
```

```
{"x":304.71100917431284,"y":872.0}
{"x":333.1514423076924,"y":832.0}
{"x":363.7630208333335,"y":768.0}
{"x":397.51587301587364,"y":756.0}
{"x":430.9072847682117,"y":604.0}
{"x":461.68715083798895,"y":537.0}
{"x":494.1598360655734,"y":488.0}
{"x":528.5816326530613,"y":294.0}
{"x":555.5166666666672,"y":180.0}
{"x":588.7979797979801,"y":198.0}
```

10.2. Write a similar Hive query that computes 10 frequency-distribution pairs for the ordertotal from the orders table where ordertotal is greater than $200.

```
SELECT explode(histogram_numeric(ordertotal,10)) AS x FROM orders
WHERE ordertotal > 200;

{"x":218.8195174551819,"y":7419.0}
{"x":254.10237580993478,"y":6945.0}
{"x":293.4231618807192,"y":6338.0}
{"x":334.57302573203015,"y":5635.0}
{"x":379.79714934930786,"y":4841.0}
{"x":428.1165628891644,"y":4015.0}
{"x":473.1484734420741,"y":2391.0}
{"x":511.2576946288467,"y":1657.0}
{"x":549.0106899902812,"y":1029.0}
{"x":589.0761194029857,"y":670.0}
```

Result

You should now be comfortable running Hive queries and using some of the more advanced features of Hive, like views and the window functions.

Lab 18: Hive Optimizations

This lab explores various methods to optimize Hive queries, including vectorization. Your data will come from Sensor files in an IoT application.

Objective: To learn how to configure Hive queries, create a table that uses the ORC file format, enable vectorization for a query, and use cost-based optimization on a query.

File locations: ~/data/SensorFiles

1. Create and Populate the Tables in Hive

   1.1. Change directory to the `data/SensorFiles` folder. Review the contents of `building.csv` and `HVAC.csv`. The `building.csv` file represents a static list of buildings owned by a company, and `HVAC.csv` is a collection of temperatures read from sensors in the buildings.

```
# more building.csv
# more HVAC.csv
-- press q to exit more
```

   1.2. Run the `create_hive_tables.sql` script from the `data/SensorFiles` folder to create two tables in Hive and populate them with the data in the CSV files:

```
# hive -f create_hive_tables.sql
```

   1.3. Verify it worked by entering the Hive shell and running two queries:

```
# hive
hive> select * from building;
hive> select * from hvac limit 20;
```

2. Run a Query using MapReduce

   2.1. Exit the Hive shell and then use more to view the contents of the ‘run_demo_mr.sql’ file:

```
hive> quit;
# more run_demo_mr.sql
set hive.execution.engine=mr;
select h.*, b.country, b.hvacproduct, b.buildingage, b.buildingmgr from
building b
join hvac h
on b.buildingid = h.buildingid;
```

   Notice the MapReduce engine is specifically set, even though it is the default execution engine.

   2.2. Quit the Hive shell and run the query:

```
# hive -f run_demo_mr.sql
```

   You should see 8,000 rows output. Note how long it takes for the query to execute.

3. Create an ORC table

   3.1. To demonstrate vectorization, the data in the Hive table must be in the ORC format. Enter the Hive shell and create a new table named `hvac_orc` from the existing hvac table:

```
# hive
hive> create table hvac_orc stored as orc as select * from hvac;
```

   3.2. The `hvac_orc` table contains the same data as the hvac table, except it is in the ORC format:

```
hive> describe formatted hvac_orc;
```

   3.3. Verify the data is in hvac_orc:

```
hive> select * from hvac_orc limit 20;
```

4. Run a Query with Vectorization

   4.1. First, run a query without vectorization on the text data:

```
hive> set hive.vectorized.execution.enabled=false;
hive> select date, count(buildingid) from hvac group by date;
```

   4.2. Second, run the same query but on the ORC data:

```
hive> select date, count(buildingid) from hvac_orc group by date;
```

   4.3. Enable vectorization:

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

   4.3. Third, run the query on the ORC table again:

```
hive> select date, count(buildingid) from hvac_orc group by date;
```

   4.4. To verify that vectorization is used, use the explain command:

```
hive> explain select date, count(buildingid) from hvac_orc group by date;
```

   Look for “Execution mode: vectorized” in the output.

5. Use Cost-Based Optimization

   5.1. Run the following command on the hvac table:

```
hive> select count(*) from hvac;
```

5.2. Run the explain command on the following query:

```
hive> explain select buildingid, max(targettemp-actualtemp) from hvac group
by buildingid;
```

Note that Basic stats are COMPLETE but that Column stats are NONE.

5.3. To use CBO, you need the table stats computed. Run the following command to compute the table statistics for hvac:

```
hive> analyze table hvac compute statistics;
```

5.3. Run the following command to compute column statistics for some of the columns in hvac:

```
hive> analyze table hvac compute statistics for columns targettemp,
actualtemp, buildingid;
```

5.4. The following commands can be found in `run_demo_cbo.sql` file located in the `~/data/SensorFiles` folder. You can open them in gedit or another editor and then copy and paste these into the Hive shell one at a time:

```
set hive.compute.query.using.stats=true;
set hive.cbo.enable=true;
set hive.stats.fetch.column.stats=true;
```

5.5. Now run the following query again:

```
hive> select count(*) from hvac;
```

Notice that this does not even run a MapReduce job and that the output is displayed immediately. It uses table statistics.

5.7. Run the following explain command:

```
hive> explain select buildingid, max(targettemp-actualtemp) from hvac group
by buildingid;
```

5.8. Look carefully at the output of the explain command. The value of Column stats should be COMPLETE.

NOTE: Once you have the stats computed for a table, you can turn CBO on and off using
the `hive.cbo.enable` and `hive.compute.query.using.stats`properties.

Result

You now can run a Hive query with the optimized queries.

Lab 19: Streaming Data with Hive and Python

This lab explores custom reducer scripts, and using them to optimize a Hive query.

Objective: Use a custom reducer script to optimize a Hive query.

File locations: ~/data

Successful outcome: The join query from the previous lab that executed in two MapReduce jobs will now execute in one MapReduce job.

Before you begin: This lab is dependent on completion of the Advanced Hive Queries lab.

---

1. Create the `max_ordertotal` view

   1.1. In the previous lab, you defined a view named `max_ordertotal`. Enter a Hive shell and use the describe command to verify:

   ```
   hive> set hive.execution.engine=mr;
   hive> describe max_ordertotal;
   OK
   maxtotal    int    None
   userid      int    None
   ```

   NOTE: If you do not have this view, define it now as:

   ```
   hive> CREATE VIEW max_ordertotal AS
   SELECT max(ordertotal) AS maxtotal, userid FROM orders GROUP BY userid;
   ```

2. Think in MapReduce

   2.1. Consider the following join statement that you executed in a previous lab (no typing required):

   ```
   SELECT ordertotal, orders.userid, itemlist
   FROM orders
   ```

```
JOIN max_ordertotal ON max_ordertotal.userid = orders.userid
AND  max_ordertotal.maxtotal = orders.ordertotal
ORDER BY orders.userid;
```

This join statement requires two MapReduce jobs to execute.

2.2. What if we could send all of the orders by a particular customer to the same reducer? How could we accomplish this?

Answer: Use the DISTRIBUTE BY clause and distribute the records by the userid column.

2.3. Suppose we have distributed the records so that we know the same reducer handles all orders from a customer. Then we could sort the orders by ordertotal ascending, and the first order would be their maximum order. Run the following query to understand the logic here:

```
hive> SELECT * FROM orders
DISTRIBUTE BY userid
SORT BY userid, ordertotal;
```

2.4. Look closely at the output. Each customer's largest order should appear last in his or her respective list of orders. For example, Caitlin F's largest order was $600 on April 25, 2012. You could also retry the query using descending order DESC as below:

```
72094   2012-04-25  100 Caitlin F   600 ...
87194   2013-01-05  100 Caitlin F   588 ...
53034   2011-06-11  100 Caitlin F   588 ...
56003   2011-07-30  100 Caitlin F   588 ...

hive> SELECT * FROM orders
DISTRIBUTE BY userid
SORT BY userid, ordertotal DESC;
```

This data may not be visible in your terminal window. You should, however, be able to see the smallest orders at the end of the list and verify that they go in descending order according to value, like this:

```
95790   2013-05-24  100 Caitlin F   13  Freezer
77781   2012-07-29  100 Caitlin F   13  Air Compressor
54316   2011-07-02  100 Caitlin F   10  Software
```

The reducer gets all orders from a customer, and the first order the reducer receives is the largest one (which is what we are trying to find). In the next step, you will use a custom reducer using Python that pulls this top value off. When you have finished reviewing the records, quit the hive shell.

```
hive> quit;
```

3. Use a Custom Reducer

3.1. Using the gedit text editor (or your favorite one), open the file `max_order.py` in your /home/cloudera/data folder.

3.2. Notice that this Python script prints the first line that it processes. Then it hangs on to the userid and skips all subsequent lines until the userid changes.

3.3. Change to the lab's directory, then copy `max_order.py` into the `/tmp` folder and make it executable:

```
# cp max_order.py /tmp
# chmod +x /tmp/max_order.py
```

3.4. Start the Hive shell, and add `max_order.py` as a resource using the add file command:

```
hive> add file /tmp/max_order.py;
Added resources:[/tmp/max_order.py]
```

NOTE: The add file command makes the file available to all mappers and reducers of this Hive query.

3.5. Specify three reducers so we can verify the logic of our query:

```
hive> set mapreduce.job.reduces=3;
```

3.6. Now run the following join query, which uses the Python script as its reducer. You may want to type this in a text file so you can rerun it easier if you have a typo and make sure you use the proper path to `max_order.py`.

```
hive> from (
select userid,ordertotal,itemlist from orders
distribute by userid
sort by userid,ordertotal DESC)
orders
insert overwrite directory 'maxorders'
reduce userid,ordertotal,itemlist using 'max_order.py';
```

The query should execute a single MapReduce job, and you should also notice three reducers.

4. View the Results

4.1. List the contents of the `maxorders` folder in HDFS. You should see three files, one from each reducer:

```
hive> dfs -ls maxorders;
Found 3 items
-rw-r--r--   3   root    hdfs    3606    maxorders/000000_0
-rw-r--r--   3   root    hdfs    3719    maxorders/000001_0
-rw-r--r--   3   root    hdfs    3680    maxorders/000002_0
```

4.2. View the contents of one of the files:

```
hive> dfs -cat maxorders/000000_0;
...
90588 Boots,Grill,Spark Plugs,Vacuum,Coffee Maker,DVD,2-Way
Radio,Dolls,Games,DVD,pillows,Pants
93600 Dishwasher,Table,Grill,DVD,DVD,DVD,Keychain,Dryer, Washer &
Dryer,Grill,Coffee Maker,pillows
96600 Table,Jeans,Washer,Wrench Set,Grill,Color Laser Printer,Dryer,Air
Compressor,DVD,Dolls,2-Way Radio,Sweater
99600 Washer,Cookware,Vacuum,Freezer,2-Way Radio,Bicycle,Washer &
Dryer,Coffee Maker,Refrigerator, DVD,Boots,DVD
```

The output shows the userid, ordertotal, and itemlist of the largest order placed by each customer.

Result

You used a custom reducer (a Python script) to modify a Hive query that originally took two MapReduce jobs to execute so that it can now be executed in a single MapReduce job. You also learned how to assign a custom reducer (or mapper) to a Hive query.

Lab 20: Working with Parquet Compressed Data Files

This lab explores creating Parquet files in Hive, and using them in a Hive query.

Objective: Using Parquet compressed files in a Hive query.

File locations: ~/data

Successful outcome: Querying compressed data in Hive.

Before you begin: This lab requires creating a rating table in Hive

1. Create an External Table named `ratings` and insert data into it. Verify the data was successfully loaded into the ratings table and you now have 655982 records.

```
hive> CREATE EXTERNAL TABLE ratings (
userid int,
movieid int ,
rating int ,
tstamp string
)ROW FORMAT  DELIMITED
 FIELDS TERMINATED BY ','
LOCATION '/user/cloudera/data/ratings' ;

hive> LOAD DATA local INPATH '/home/cloudera/data/ratings.csv' INTO TABLE ratings;
hive> Select * from ratings limit 20;
hive> Select count(*) from ratings;
```

1.1. Create a new Parquet tale named ratings_parquet:

```
hive> CREATE TABLE ratings_parquet LIKE ratings STORED AS PARQUET ;
```

1.2. Load the data into ratings_parquet from the ratings table:

```
hive> INSERT OVERWRITE TABLE ratings_parquet SELECT * FROM ratings;
```

1.3. Verify the schema and file type of the table ratings_parquet:

```
hive> DESC FORMATTED ratings_parquet;
```

```
Query: describe FORMATTED ratings_parquet
+-----------------------------+------------------------------------------------------------------------------------------+-------------------+
| name                        | type                                                                                     | comment           |
+-----------------------------+------------------------------------------------------------------------------------------+-------------------+
| # col_name                  | data_type                                                                                | comment           |
|                             | NULL                                                                                     | NULL              |
| userid                      | int                                                                                      | NULL              |
| movieid                     | int                                                                                      | NULL              |
| rating                      | int                                                                                      | NULL              |
| tstamp                      | string                                                                                   | NULL              |
|                             | NULL                                                                                     | NULL              |
| # Detailed Table Information | NULL                                                                                    | NULL              |
| Database:                   | default                                                                                  | NULL              |
| Owner:                      | datacouch                                                                                | NULL              |
| CreateTime:                 | Tue May 29 10:20:40 UTC 2018                                                             | NULL              |
| LastAccessTime:             | UNKNOWN                                                                                  | NULL              |
| Protect Mode:               | None                                                                                     | NULL              |
| Retention:                  | 0                                                                                        | NULL              |
| Location:                   | hdfs://datacouch.c.alpine-comfort-195107.internal:8020/user/hive/warehouse/ratings_parquet | NULL            |
| Table Type:                 | MANAGED_TABLE                                                                            | NULL              |
| Table Parameters:           | NULL                                                                                     | NULL              |
|                             | numRows                                                                                  | -1                |
|                             | transient_lastDdlTime                                                                    | 1527588478        |
|                             | NULL                                                                                     | NULL              |
| # Storage Information       | NULL                                                                                     | NULL              |
| SerDe Library:              | org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe                              | NULL              |
| InputFormat:                | org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat                            | NULL              |
| OutputFormat:               | org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat                           | NULL              |
| Compressed:                 | No                                                                                       | NULL              |
| Num Buckets:                | 0                                                                                        | NULL              |
| Bucket Columns:             | []                                                                                       | NULL              |
| Sort Columns:               | []                                                                                       | NULL              |
| Storage Desc Params:        | NULL                                                                                     | NULL              |
|                             | field.delim                                                                              | ,                 |
|                             | serialization.format                                                                     | ,                 |
+-----------------------------+------------------------------------------------------------------------------------------+-------------------+
Fetched 31 row(s) in 0.01s
```

1.4. Open a terminal or the Hadoop browser and view the parquet file in HDFS:

```
hive> hdfs dfs -ls /user/hive/warehouse/ratings_parquet;
Found 1 items
-rwxrwxrwx   1 cloudera supergroup    9063131 2018-09-30 08:12
/user/hive/warehouse/ratings_parquet/000000_0

desc ratings_parquet;
OK
userid                      int
movieid                     int
rating                      int
tstamp                      string
Time taken: 0.088 seconds, Fetched: 4 row(s)
```

1.5. Create an avro file from the original ratings_parquet csv file:

```
hive> CREATE TABLE ratings_avro STORED AS AVRO
AS SELECT userid,movieid,rating,tstamp FROM ratings_parquet;
```

1.6. Create another table like ratings_parquet to use with Snappy compression:

```
hive> create table parquet_snappy like ratings_parquet;
Query: create table parquet_snappy like ratings_parquet
Fetched 0 row(s) in 0.05s
```

1.7. Modify the hive compression default to use Snappy compression:

```
hive> set COMPRESSION_CODEC=snappy;
```

1.8. Insert data into parquet_snappy using Snappy compression:

```
hive> insert into parquet_snappy select * from ratings_parquet;
Query: insert into parquet_snappy select * from ratings_parquet
Query submitted at: 2018-05-29 10:37:20 (Coordinator: http://datacouch:25000)
Query progress can be monitored at: http://datacouch:25000/query_plan?query_id=404d4241431793fe:12d9661e00000000
Modified 655982 row(s) in 6.33s
```

1.9. Verify the data was loaded then select some data from the parquet_snappy table:

```
hive> select count(*) from parquet_snappy;
Total MapReduce CPU Time Spent: 5 seconds 920 msec
OK
655982
Time taken: 28.335 seconds, Fetched: 1 row(s)
```

```
hive> select * from parquet_snappy limit 10;
OK
1 1193    5       978300760
1 661     3       978302109
1 914     3       978301968
1 3408    4       978300275
1 2355    5       978824291
1 1197    3       978302268
1 1287    5       978302039
1 2804    5       978300719
1 594     4       978302268
1 919     4       978301368
Time taken: 0.075 seconds, Fetched: 10 row(s)
```

Result

You used Avro formatted and Parquet compressed data tables with a Hive query.

Lab 21: Exploring AVRO JSON Data Files

This lab explores advanced Avro formatted tables.

Objective: Using Avro files in a Hive.

File locations: ~/data

Successful outcome: Reviewing stored Avro data in Hive.

Before you begin: This lab reviews Avro files used with Hive

---

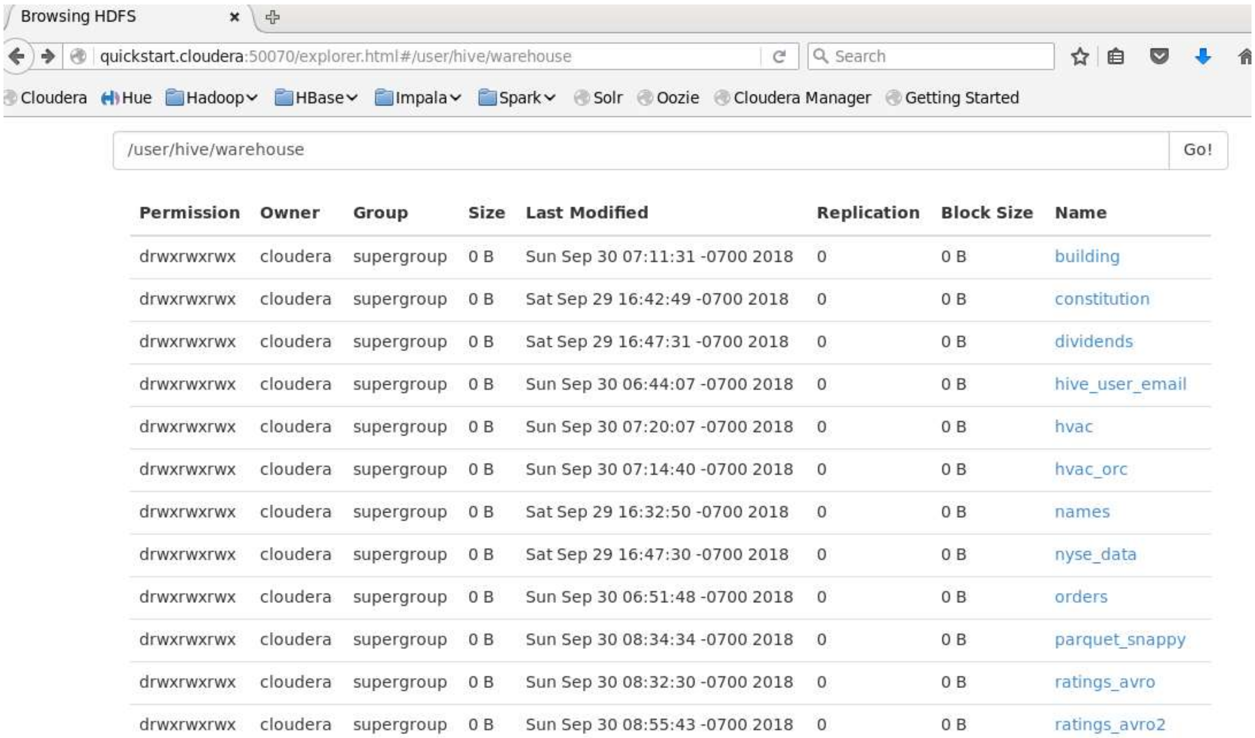
1. Converting a csv file into a ratings table with Avro file formatting in Hive

```
hive> CREATE TABLE ratings_avro2 STORED AS AVRO AS SELECT
userid,movieid,rating,tstamp FROM ratings ;
```

1.1. Verify the table was created successfully with the format:

```
hive> dfs -ls /user/hive/warehouse/ratings_avro2;
Found 1 items
-rwxrwxrwx   1 cloudera supergroup   12467907 2018-09-30 08:55
/user/hive/warehouse/ratings_avro2/000000_0
```

Remember you can view the table information by drilling down under the /user/hive/warehouse folder to view ratings_avro2 directory.



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 07:11:31 -0700 2018	0	0 B	building
drwxrwxrwx	cloudera	supergroup	0 B	Sat Sep 29 16:42:49 -0700 2018	0	0 B	constitution
drwxrwxrwx	cloudera	supergroup	0 B	Sat Sep 29 16:47:31 -0700 2018	0	0 B	dividends
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 06:44:07 -0700 2018	0	0 B	hive_user_email
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 07:20:07 -0700 2018	0	0 B	hvac
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 07:14:40 -0700 2018	0	0 B	hvac_orc
drwxrwxrwx	cloudera	supergroup	0 B	Sat Sep 29 16:32:50 -0700 2018	0	0 B	names
drwxrwxrwx	cloudera	supergroup	0 B	Sat Sep 29 16:47:30 -0700 2018	0	0 B	nyse_data
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 06:51:48 -0700 2018	0	0 B	orders
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 08:34:34 -0700 2018	0	0 B	parquet_snappy
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 08:32:30 -0700 2018	0	0 B	ratings_avro
drwxrwxrwx	cloudera	supergroup	0 B	Sun Sep 30 08:55:43 -0700 2018	0	0 B	ratings_avro2

2. You can view the avro files in JSON format with the following command from a terminal window:

```
# avro-tools tojson 000000_0  | head -5

{"userid":{"int":1},"movieid":{"int":1193},"rating":{"int":5},"tstamp":{"string":"978300760"}}
{"userid":{"int":1},"movieid":{"int":661},"rating":{"int":3},"tstamp":{"string":"978302109"}}
{"userid":{"int":1},"movieid":{"int":914},"rating":{"int":3},"tstamp":{"string":"978301968"}}
{"userid":{"int":1},"movieid":{"int":3408},"rating":{"int":4},"tstamp":{"string":"978300275"}}
{"userid":{"int":1},"movieid":{"int":2355},"rating":{"int":5},"tstamp":{"string":"978824291"}}
```

2.1. You can view the avro file schema in JSON format with the following command from a terminal window:

```
# avro-tools getschema 000000_0

{
  "type" : "record",
  "name" : "ratings_avro2",
  "namespace" : "default",
  "fields" : [ {
```

```
  "name" : "userid",
  "type" : [ "null", "int" ],
  "default" : null
 }, {
  "name" : "movieid",
  "type" : [ "null", "int" ],
  "default" : null
 }, {
  "name" : "rating",
  "type" : [ "null", "int" ],
  "default" : null
 }, {
  "name" : "tstamp",
  "type" : [ "null", "string" ],
  "default" : null
 } ]
}
```

2.2. You can now view the data in the avro file schema with the following command from a terminal window:

```
hive> select count(*) from ratings_avro2;
Total MapReduce CPU Time Spent: 5 seconds 920 msec
OK
655982
Time taken: 28.335 seconds, Fetched: 1 row(s)

 hive> select * from ratings_avro2 limit 10;
OK
1 1193     5       978300760
1 661      3       978302109
1 914      3       978301968
1 3408     4       978300275
1 2355     5       978824291
1 1197     3       978302268
1 1287     5       978302039
1 2804     5       978300719
1 594      4       978302268
1 919      4       978301368
Time taken: 0.075 seconds, Fetched: 10 row(s)
```

Result

You used Avro JSON formatted data tables with a Hive query.

Lab 22: Exploring Hive Regular Expressions

This lab explores regular expressions used with Hive.

Objective: Using RegEx with Hive.

File locations: ~/data

Successful outcome: Understanding of RegEx in Hive.

SerDe is an abstraction in Hive API for reading and writing data in a particular format. It stands for Serializer and Deserializer.

RegexSerDe (a type of Serde) will read records based on supplied regular expression. Only available in Hive.

Before you begin: This lab reviews RegEx used with Hive

---

1. Create a new table in Hive called ratings_serde

```
hive> CREATE EXTERNAL Table ratings_serde(userid string,movieid string,rating int,tstamp string)row format serde 'org.apache.hadoop.hive.serde2.RegexSerDe' with SERDEPROPERTIES("input.regex"="(\\d*),(\\d*),(\\d*),(\\w*)");
```

1.1. Load a file ratings.csv into the table ratings_serde with a delimiter of ‘,’, If the file ratings.csv file is not already loaded into the HDFS path /user/cloudera use the command: hdfs dfs -put ratings.csv

```
hive> LOAD DATA INPATH '/user/cloudera/ratings.csv' INTO TABLE ratings_serde;
```

1.2. Ensure the data was loaded properly from the file:

```
hive> select * from ratings_serde limit 5;
OK
1	1193	5	978300760
1	661	3	978302109
1	914	3	978301968
1	3408	4	978300275
1	2355	5	978824291
Time taken: 0.068 seconds, Fetched: 5 row(s)
```

2. Create a new table in Hive called ratings_serde

2.1. Creating a table from fixed length file in Hive:

```
hive> Create external table rating_fixed
(userid int,
movieid int,
rating int,
tstamp string)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
with SERDEPROPERTIES("input.regex" = "(.{4})(.{3})(.{1})(.{10})" );
```

2.2. Creating a file fixed.txt using an editor like gedit as follows:

```
123456742017-05-10
789654332017-05-11
```

2.3. Import fixed.txt file into HDFS:

```
# hdfs dfs -put fixed.txt
```

2.4. Load the data into the table rating_fixed:

```
hive> LOAD DATA INPATH '/user/cloudera/fixed.txt' INTO TABLE rating_fixed;
```

2.5. Verify the data is in the table rating_fixed:

```
hive> Select * from rating_fixed;
OK
1234        567        4        2017-05-10
7896        543        3        2017-05-11
Time taken: 0.068 seconds, Fetched: 2 row(s)
```

Result

You used Regular Expression parsing on data tables with a Hive query.

Lab 23: Exploring Indexes in Hive

This lab explores using indexes to improve performance with Hive queries.

Objective: Using Indexes with Hive.

File locations: ~/data

Successful outcome: Understanding Indexing in Hive.

Before you begin: This lab reviews Index types used with Hive

---

1. We will start by exploring Compact Indexing in Hive

```
hive> SELECT count(movieid) from ratings where rating > 2;

Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1533450775174_0011, Tracking URL = http://training-talend.localdomain:8089/proxy/application_1
/
Kill Command = /usr/hdp/2.5.0.0-1245/hadoop/bin/hadoop job  -kill job_1533450775174_0011
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2018-08-05 10:41:54,930 Stage-1 map = 0%,  reduce = 0%
2018-08-05 10:42:02,246 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 4.33 sec
2018-08-05 10:42:09,605 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 6.98 sec
MapReduce Total cumulative CPU time: 6 seconds 980 msec
Ended Job = job_1533450775174_0011
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 6.98 sec   HDFS Read: 14115182 HDFS Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 6 seconds 980 msec
OK
546395
Time taken: 22.488 seconds, Fetched: 1 row(s)
hive>
```

Here we can see the average age of the athletes to be 546395 and the time for performing this operation is 22.488 seconds.

1.1. Now let's add an index to the table:

```
hive> CREATE INDEX rating_index ON TABLE ratings (movieid) AS
'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'
WITH DEFERRED REBUILD;
```

1.2. And now add the index to the table:

```
hive> ALTER INDEX rating_index on ratings REBUILD;
```

1.3. We can view the indexes created ifor the table by using the command:

```
hive> show formatted index on ratings;
```

```
Loading data to table default.default__ratings_rating_index__
Table default.default__ratings_rating_index__ stats: [numFiles=1, numRows=3653, totalSize=5691583, rawDataSize=5687930]
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 10.14 sec   HDFS Read: 14116273 HDFS Write: 5691685 SUCCESS
Total MapReduce CPU Time Spent: 10 seconds 140 msec
OK
Time taken: 24.191 seconds
hive> show formatted index on ratings;
OK
idx_name                tab_name                col_names               idx_tab_name            idx_type                comment

rating_index            ratings                 movieid                 default__ratings_rating_index__ compact
Time taken: 0.134 seconds, Fetched: 5 row(s)
hive>
```

2. We can now rerun the query to see what impact Indexing has in Hive

```
hive> SELECT count(movieid) from ratings where rating > 2;
```

```
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1533450775174_0014, Tracking URL = http://training-talend.localdomain:8089/proxy/applic
/
Kill Command = /usr/hdp/2.5.0.0-1245/hadoop/bin/hadoop job  -kill job_1533450775174_0014
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2018-08-05 10:48:43,072 Stage-1 map = 0%,  reduce = 0%
2018-08-05 10:48:49,313 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 4.24 sec
2018-08-05 10:48:55,516 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 6.88 sec
MapReduce Total cumulative CPU time: 6 seconds 880 msec
Ended Job = job_1533450775174_0014
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 6.88 sec   HDFS Read: 14115176 HDFS Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 6 seconds 880 msec
OK
546395
Time taken: 21.548 seconds, Fetched: 1 row(s)
hive>
```

We have now got the count as 546395, which is same as the above, but now the time taken for performing this operation is 21.548 seconds, which is less than the above case.

3. There are other types of indexes available in Hive. We will explore others now. Let's try a Bitmap index.

```
hive> CREATE INDEX rating_index_bitmap ON TABLE ratings (movieid) AS 'BITMAP' WITH DEFERRED REBUILD;
```

3.1. And now add the index to the table:

hive> ALTER INDEX rating_index on ratings REBUILD;

```
Number of reduce tasks not specified. Estimated from input data size: 2
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1533450775174_0016, Tracking URL = http://training-talend.localdomain:8089/proxy
/application_1533450775174_0016/
Kill Command = /usr/hdp/2.5.0.0-1245/hadoop/bin/hadoop job  -kill job_1533450775174_0016
Hadoop job information for Stage-2: number of mappers: 1; number of reducers: 2
2018-08-05 11:11:50,453 Stage-2 map = 0%,  reduce = 0%
2018-08-05 11:12:00,841 Stage-2 map = 100%,  reduce = 0%, Cumulative CPU 8.61 sec
2018-08-05 11:12:12,459 Stage-2 map = 100%,  reduce = 50%, Cumulative CPU 15.45 sec
2018-08-05 11:12:13,488 Stage-2 map = 100%,  reduce = 100%, Cumulative CPU 22.09 sec
MapReduce Total cumulative CPU time: 22 seconds 90 msec
Ended Job = job_1533450775174_0016
Loading data to table default.default__ratings_rating_index_bitmap__
Table default.default__ratings_rating_index_bitmap__ stats: [numFiles=2, numRows=655982, totalSize=
74732562, rawDataSize=74076580]
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 17.73 sec   HDFS Read: 14116070 HDFS Write: 8007
2093 SUCCESS
Stage-Stage-2: Map: 1  Reduce: 2   Cumulative CPU: 22.09 sec   HDFS Read: 80082376 HDFS Write: 7473
2786 SUCCESS
Total MapReduce CPU Time Spent: 39 seconds 820 msec
OK
Time taken: 57.409 seconds
hive>
```

Here, As 'BITMAP' defines the type of index as BITMAP. We have successfully created the Bitmap index for the table.

3.2. We can check the available indexes on the table. We should see both indexes we placed on the table:

hive> show formatted index on ratings;

```
hive> show formatted index on ratings;
OK
idx_name                tab_name                col_names               idx_tab_name            idx_type                comment


rating_index            ratings                 movieid                 default__ratings_rating_index__     compact
rating_index_bitmap     ratings                 movieid                 default__ratings_rating_index_bitmap__      bitmap

Time taken: 0.136 seconds, Fetched: 6 row(s)
hive>
```

4. Count operation with two indexes.

4.1 Now, let's perform the same Count operation having the two indexes.

hive> SELECT count(movieid) from ratings where rating > 2;

```
Query ID = talend_20180805112550_9e51fb67-42cf-4c0d-9c4d-afa2691ff70f
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1533450775174_0017, Tracking URL = http://training-talend.localdomain:8089/proxy/applicati
Kill Command = /usr/hdp/2.5.0.0-1245/hadoop/bin/hadoop job  -kill job_1533450775174_0017
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2018-08-05 11:25:57,516 Stage-1 map = 0%,  reduce = 0%
2018-08-05 11:26:03,709 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 4.48 sec
2018-08-05 11:26:09,895 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 7.96 sec
MapReduce Total cumulative CPU time: 7 seconds 960 msec
Ended Job = job_1533450775174_0017
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 7.96 sec   HDFS Read: 14115176 HDFS Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 7 seconds 960 msec
OK
546395
Time taken: 21.96 seconds, Fetched: 1 row(s)
hive>
```

This time, we have got the same result in 21.96 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.

4.2. Let’s delete one of the indexes using the following command.

```
hive> DROP INDEX IF EXISTS rating_index ON ratings;

hive> show formatted index on ratings;
```

```
hive> show formatted index on ratings;
OK
idx_name                tab_name                col_names               idx_tab_name            idx_type                comment


rating_index_bitmap     ratings                 movieid                 default__ratings_rating_index_bitmap__  bitmap
Time taken: 0.171 seconds, Fetched: 4 row(s)
```

5. Count operation with Bitmap indexes.

5.1. Let’s perform the same Count operation with the Bitmap index.

```
hive> SELECT count(movieid) from ratings where rating > 2;
```

```
Query ID = talend_20180805113319_44e6ddf3-de64-4333-bb66-1fcaf28ec853
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1533450775174_0018, Tracking URL = http://training-talend.localdomain:8089/proxy/appl
Kill Command = /usr/hdp/2.5.0.0-1245/hadoop/bin/hadoop job  -kill job_1533450775174_0018
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2018-08-05 11:33:25,701 Stage-1 map = 0%,  reduce = 0%
2018-08-05 11:33:31,923 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 4.3 sec
2018-08-05 11:33:38,108 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 6.95 sec
MapReduce Total cumulative CPU time: 6 seconds 950 msec
Ended Job = job_1533450775174_0018
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 6.95 sec   HDFS Read: 14115182 HDFS Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 6 seconds 950 msec
OK
546395
Time taken: 20.781 seconds, Fetched: 1 row(s)
hive>
```

We have got the average age as 546395, which is same as the above cases but the operation was done in just 20.781 seconds, which is less than the above two cases.

Result

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.

Lab 24: Deduplication using Hive

This lab explores using deduplication to improve performance with Hive queries.

Objective: Using data deduplication with Hive.

File locations: ~/data

Successful outcome: Understanding deduplication in Hive.

Before you begin: This lab reviews deduplication with Hive

1. We will start by loading sample.csv into Hadoop. If you don't have the directory already, create it with the following command

```
# hdfs dfs -mkdir data/
```

2. Next we will load sample.csv into Hadoop

```
# hdfs dfs -put /home/cloudera/data/sample.csv data/
```

3. Next we will create a table call table_station_t in Hive as follows:

```
hive>create external table station_t(year_col int,temp string, station_id int, date_col string) row format delimited fields terminated by ',';
```

4. Next we will create a table call table_station_t in Hive as follows:

```
hive>create external table station_t(year_col int,temp string, station_id int, date_col string) row format delimited fields terminated by ',';
hive> create external table station_t(year_col int,temp string, station_id int,
date_col string) row format delimited fields terminated by ',';
OK
Time taken: 0.063 seconds
```

5. Next load the csv file data into the table:

```
hive> load data inpath '/user/cloudera/data/sample.csv' into table station_t;
Loading data to table default.station_t
Table default.station_t stats: [numFiles=3, numRows=0, totalSize=2086614, rawDataSize=0]
OK
Time taken: 0.803 seconds
```

6. Select the data from the Hive table to verify it loaded correctly:

```
hive> select * from station_t limit 2;
hive> select * from station_t limit 2;
OK
1945    3       720586  03-06-1945
1945    -1.8    783960  04-06-1945
Time taken: 0.137 seconds, Fetched: 2 row(s)
```

7. Now let's partition the data and eliminate duplicate values with MapReduce:

```
hive> create table unique_station_t as select year_col, temp, station_id,
date_col from (select *, ROW_NUMBER() OVER (Partition by station_id order by
date_col desc) as ROWNUM from station_t) x where ROWNUM = 1;

hive> select count(*) from station_t;

MapReduce Total cumulative CPU time: 4 seconds 650 msec
Ended Job = job_1502688361572_0007
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 4.65 sec   HDFS Read: 702338
HDFS Write: 6 SUCCESS
Total MapReduce CPU Time Spent: 4 seconds 650 msec
OK
24043
```

8. Now let's partition the data and eliminate duplicate values with MapReduce:

```
hive> create table unique_station_t as select year_col, temp, station_id,
date_col from (select *, ROW_NUMBER() OVER (Partition by station_id order by
date_col desc) as ROWNUM from station_t) x where ROWNUM = 1;

hive> select count(*) from unique_station_t;

MapReduce Total cumulative CPU time: 4 seconds 610 msec
Ended Job = job_1502688361572_0006
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 4.61 sec   HDFS Read: 639048
HDFS Write: 6 SUCCESS
Total MapReduce CPU Time Spent: 4 seconds 610 msec
OK
22636
Time taken: 43.867 seconds, Fetched: 1 row(s)
```

Result

We have seen that data deduplication can help with the performance of Hive Queries

Lab 25: Converting Date formats in Hive

This lab explores using date formats in Hive.

Objective: Using date formatting with Hive.

File locations: ~/data

Successful outcome: Understanding data formats in Hive.

Before you begin: This lab reviews data with Hive

---

1. Let's start with a review of current date in traditional format in Hive:

```
hive> select date_col from unique_station_t limit 5;
```

```
hive> select date_col from unique_station_t limit 5;
OK
1987-06-15
1970-10-19
1960-09-01
2010-07-13
2009-10-18
Time taken: 0.069 seconds, Fetched: 5 row(s)
```

2. To change the date formatting of the current date in Hive use the following command:

```
hive> select from_unixtime(unix_timestamp(date_col,"yyyy-MM-
dd"),"dd/MM/yyyy") from station_t limit 5;
```

```
hive> select from_unixtime(unix_timestamp(date_col,"yyyy-MM-dd"),"dd/MM/yyyy") f
rom station_t limit 5;
OK
03/06/1945
04/06/1945
05/06/1945
06/06/1945
07/06/1945
Time taken: 0.058 seconds, Fetched: 5 row(s)
hive>
```

Result

We have seen that the formatting of dates is possible in Hive Queries