Experiment 11.2

Using Hive to create, alter, and drop databases, tables and views.

1. **Creating a table**

hive> CREATE TABLE posts (user STRING, post STRING, time BIGINT)

> ROW FORMAT DELIMITED

> FIELDS TERMINATED BY ','

> STORED AS TEXTFILE;

1st line: creates a table with 3 columns

2nd and 3rd line: how the underlying file should be parsed

4th line: how to store data

Statements must end with a semicolon and can span multiple rows

1. **Show tables**

hive> show tables;

Display all of the tables

OK

posts

Time taken: 0.221 seconds

Result is displayed between ̎OK̎ and ̎Time taken...

1. **Describe command**

hive> describe posts;

OK

user string

post string

time bigint

Time taken: 0.212 seconds

Displays schema for posts table

1. **Load Data Into a Table**

hive> LOAD DATA LOCAL INPATH 'data/user-posts.txt'

> OVERWRITE INTO TABLE posts;

Existing records the table posts are deleted; data in user-posts.txt is loaded into Hive’s posts table

$ hdfs dfs -cat /user/hive/warehouse/posts/user-posts.txt

user1,Funny Story,1343182026191

user2,Cool Deal,1343182133839

user4,Interesting Post,1343182154633

user5,Yet Another Blog,13431839394

Under the covers Hive stores it’s tables in /user/hive/warehouse(unless configured differently)

Load data from HDFS location

hive> LOAD DATA INPATH '/training/hive/user-posts.txt'

> OVERWRITE INTO TABLE posts;

File is copied from the provided location to /user/hive/warehouse/

(or configured location)

Load data from a local file system

hive> LOAD DATA LOCAL INPATH 'data/user-posts.txt'

> OVERWRITE INTO TABLE posts;

File is copied from the provided location to /user/hive/warehouse/

1. **Querying Data**

hive> select count (1) from posts;

This query counts the number of records in posts table, similar to count (\*).

Result is 4 records

OK

4

Time taken: 14.204 seconds

hive> select \* from posts where user="user2";

...

...

OK

user2 Cool Deal 1343182133839

Time taken: 12.184 seconds

Select records for "user2"

hive> select \* from posts where time<=1343182133839 limit 2;

...

...

OK

user1 Funny Story 1343182026191

user2 Cool Deal 1343182133839

Time taken: 12.003 seconds

Usually there are too many results to display, then one could utilize limit command to bound the display.

1. **Drop the Table – To remove the table.**

hive> DROP TABLE posts;

OK

Time taken: 2.182 seconds

1. **Joins**

**Inner joins**

The simplest kind of join is the inner join, where each match in the input tables results in a row in the output. Consider two small demonstration tables, sales (which lists the names of people and the IDs of the items they bought) and things (which lists the item IDs and their names):

hive> SELECT \* FROM sales;

Joe 2

Hank 4

Ali 0

Eve 3

Hank 2

hive> SELECT \* FROM things;

2 Tie

4 Coat

3 Hat

1 Scarf

We can perform an inner join on the two tables as follows:

hive> SELECT sales.\*, things.\*

> FROM sales JOIN things ON (sales.id = things.id);

Joe 2 2 Tie

Hank 4 4 Coat

Eve 3 3 Hat

Hank 2 2 Tie

The table in the FROM clause (sales) is joined with the table in the JOIN clause (things), using the predicate in the ON clause.

Hive (like MySQL and Oracle) allows you to list the join tables in the FROM clause and specify the join condition in the WHERE clause of a SELECT statement. For example, the following is another way of expressing the query we just saw:

SELECT sales.\*, things.\*

FROM sales, things

WHERE sales.id = things.id;

We can see how many MapReduce jobs Hive will use for any particular query by prefixing it with the EXPLAIN keyword:

EXPLAIN

SELECT sales.\*, things.\*

FROM sales JOIN things ON (sales.id = things.id);

The EXPLAIN output includes many details about the execution plan for the query, including the abstract syntax tree, the dependency graph for the stages that Hive will execute, and information about each stage

**Outer joins**

Outer joins allow you to find nonmatches in the tables being joined.

**LEFT OUTER JOIN**

The query will return a row for every row in the left table (sales), even if there is no corresponding row in the table it is being joined to (things):

hive> SELECT sales.\*, things.\*

> FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);

Joe 2 2 Tie

Hank 4 4 Coat

Ali 0 NULL NULL

Eve 3 3 Hat

Hank 2 2 Tie

**RIGHT OUTER JOINS**

In this case, all items from the things table are included, even those that weren’t purchased by anyone (a scarf):

hive> SELECT sales.\*, things.\*

> FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);

Joe 2 2 Tie

Hank 2 2 Tie

Hank 4 4 Coat

Eve 3 3 Hat

NULL NULL 1 Scarf

Finally, there is a full outer join, where the output has a row for each row from both tables in the join:

hive> SELECT sales.\*, things.\*

> FROM sales FULL OUTER JOIN things ON (sales.id = things.id);

Ali 0 NULL NULL

NULL NULL 1 Scarf

Hank 2 2 Tie

Joe 2 2 Tie

Eve 3 3 Hat

Hank 4 4 Coat

1. **Views**

A view is a sort of “virtual table” that is defined by a SELECT statement. Views can be used to present data to users in a way that differs from the way it is actually stored on disk. Often, the data from existing tables is simplified or aggregated in a particular way that makes it convenient for further processing.

In Hive, a view is not materialized to disk when it is created; rather, the view’s SELECT statement is executed when the statement that refers to the view is run.

Finding the mean maximum temperature for every year and weather station. First, let’s create a view for valid records—that is, records that have a particular quality value:

CREATE VIEW valid\_records

AS

SELECT \*

FROM records2

WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9);

When we create a view, the query is not run; it is simply stored in the metastore. Views are included in the output of the SHOW TABLES command, and you can see more details about a particular view, including the query used to define it, by issuing the DESCRIBE EXTENDED view\_name command

Let’s create a second view of maximum temperatures for each station and year. It is based on the valid\_records view:

CREATE VIEW max\_temperatures (station, year, max\_temperature)

AS

SELECT station, year, MAX(temperature)

FROM valid\_records

GROUP BY station, year;

With the views in place, we can now use them by running a query:

SELECT station, year, AVG(max\_temperature)

FROM max\_temperatures

GROUP BY station, year;