CSE3BDC/CSE5BDC
Lab 03: Apache Hive

Department of Computer Science and IT, La Trobe University

Objectives

- Learn the advantages of Hive over traditional MapReduce
- Gain experience writing and executing basic workloads in Hive
- See the connection between Hive and MapReduce

Like last week's lab, the work for this lab must be completed inside the Cloudera VM. Please refer to the instructions on LMS if you've forgotten how to start the VM.

Task 1: Word count

The task of counting how frequently words appear in a corpus of documents is commonly used as an introduction to big data processing. So, surely enough, our first exercise with Hive will be counting words!

1. Ensure that you have the lab files downloaded in the VM. Saving them to the desktop is fine, just remember to copy your work onto your student drive at the end of the lab.

2. Open the terminal in the same directory as the `.hql` lab files. These `.hql` files contain HiveQL code, which is a dialect of SQL used by Hive.

3. Run the first HiveQL script:

   ```
   $ hive -f t1-wordcount.hql
   ```

   Note that in Hive, we are using hard-coded directory paths rather than specifying command-line arguments, so be sure to not modify the path of the input data files without modifying the HiveQL file as well.

4. Take a look at the files inside the input directory "Input_data/1/". Notice there are many different files in it. When you give Hive an input directory, it takes all the files in it together as the input.

5. The program may take a short while to complete, so while you wait, take this opportunity look over the code of the `t1-wordcount.hql` file. The program creates the following three tables:

   - The `myinput` table. This table stores lines of text from a directory of text files. Each row of the table stores an entire line of text.
   - The `mywords` table. This table stores individual words extracted from the `myinput` table. The table is created by expanding (using the EXPLODE function) each row of the `myinput` table into multiple rows, where each row contains a single word. The create table command also strips the input of punctuation and control characters using a regular expression.
   - The `wordcount` table gets each word from the `mywords` table and counts the number of occurrences of each unique word using `count(1)`. It also removes any blank words using the `WHERE` clause by keeping only words that are not blank.

   Finally, this data is then written to an output file using the last two lines of the code.

6. Once the program has completed successfully, the tables will be stored in Hive. Go into the Hive interpreter and see what each table contains. Start a new terminal instance and type the following to get into the Hive interpreter (you can keep this terminal open to use the hive interpreter at anytime):

   ```
   $ hive
   ```

7. List all of the tables in Hive:

   ```
   SHOW TABLES;
   ```

   You should see the three tables created by the script listed.

8. Type in the following to see the first 10 rows of the `myinput` table:

   ```
   SELECT * FROM myinput LIMIT 10;
   ```

9. Repeat the above for the `mywords` and `wordcount` tables.

10. Quit out of the Hive interpreter by typing `exit;`

11. Try to run the Hive script again using the same command and see what happens.

    ```
    $ hive -f t1-wordcount.hql
    ```

12. This time the script fails with an `AlreadyExistsException`. This is because the system still contains the old tables you created. You need to drop the three tables at the beginning of the script before recreating them again. Insert the following three lines at the top of the script.

```
DROP TABLE myinput;
DROP TABLE mywords;
DROP TABLE wordcount;
```

13. Once the program has completed successfully, browse to the `task1-out` folder and view the generated output in a text editor. If everything went well, you should see a whole lot of words and numbers in no particular order. The columns are separated by the `\001` character (rendered as `SOH` in Sublime), which is the default Hive field delimiter.

14. You probably do not like having the output columns separated by `\001`. You can change the separator to anything you want. Do the following in order to make the output columns separated by the tab character `\t` instead. Insert the following commands just before the `SELECT * FROM wordcount;` line:

```
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
```

Task 2: Subqueries and Hive MapReduce analysis

Using subqueries

Let's try to redo the word count program using just two tables: `myinput` and `wordcount`. We will do this by writing the `mywords` table as a subquery within the `wordcount` table.

1. Copy the file `t1-wordcount.hql` to `t2-wordcount.hql` so that we don't have to start from scratch.

2. In the command to create the `wordcount` table (which begins with the line `CREATE TABLE wordcount AS`), replace the line

```
FROM mywords
```

with

```
FROM (<subquery>) splitwords
```

where `<subquery>` is the second and third line from the command to create the `mywords` table (`SELECT EXPLODE ...  FROM myinput`). This modification makes the `wordcount` table take its input from the result of a subquery instead of from the table `mywords`. The `splitwords` is just a name we give to the table created by the subquery, it can be any valid name.

3. Modify the `t2-wordcount.hql` script so that the output is saved to `task2-out` instead of `task1-out`.

4. Execute `t2-wordcount.hql` and check that you get the same output as for Task 1.

5. Now lets do a small experiment comparing the efficiency of `t1-wordcount.hql` and `t2-wordcount.hql`. Open the job browser in Hue now. While keeping the job browser open, first run `t1-wordcount.hql` and then `t2-wordcount.hql`. What do you notice?

   (a) The processing for `t1-wordcount.hql` uses **three** separate MapReduce jobs:

      i. MapReduce job 1: Create the `mywords` table.

      ii. MapReduce job 2: Create the `wordcount` table.

      iii. MapReduce job 3: Output the `wordcount` table to the local directory.

   (b) The processing for `t2-wordcount.hql` uses **two** separate MapReduce jobs:

      i. MapReduce job 1: Create the `wordcount` table.

      ii. MapReduce job 2: Output the `wordcount` table to the local directory.

   (c) If you add up the total time taken by jobs for each of the two scripts, you should see that `t2-wordcount.hql` is faster. This is because by having one less MapReduce job `t2-wordcount.hql` performs roughly 1/3 less disk IO. This is because at the start of each MapReduce job all of the data needs to be loaded from disk, and then the results need to be written to disk again afterwards.

6. It's all good and well to be able to count a bunch of words, but the data right now is not presented in any useful order. Modify the program so that it presents the `wordcount` data ordered by the count in descending order (that is, the words with the most occurrences will appear first). As a secondary order, make the words also appear is ascending order.
   Hint: You may want to use the familiar SQL syntax shown below at the end of the query which creates the `wordcount` table (don't forget to delete the semi-colon at the end of the `GROUP BY`):

```
ORDER BY <col1> DESC, <col2> ASC;
```

7. Execute the script again and verify the output. You should now have an output dataset with the most frequent occurring words at the top and, in order to break ties (words with the same frequency), words are also listed in alphabetic order.

You have now modified and executed a basic word count example in Hive that also orders its output, with a program that is only about 20 lines long. But don't stop there—there's many other things we can do in Hive, just as easily!

Exercise 1. Modify the `t2-wordcount.hql` script again, this time so that it only outputs the top 10 most frequently occurring words. Hint: this is similar to how you added the `ORDER BY` clause earlier, but this time use `LIMIT` (see the LIMIT clause documentation).

Task 3: Stop list and joins

With just one line you can change the ordering of the output, and with another you can modify how many rows to select. If you were to write MapReduce code for this directly, it would take a lot more effort (trust us on this one!). But the SQL-like nature of Hive provides a lot more than just this—one of the most powerful tools in your arsenal is being able to use **joins**.

The join operation returns combinations of records from two tables. For example, if you have a table of students and a table of classes, you could use a join to obtain a list of student-class combinations.

Stop lists

A **stop list** is a list of words that we want to filter out of a data set. Typically stop lists include words which don't carry much meaning, like "a", "the", "in", etc. Therefore we want to look inside a data set and then discard every word in it that appears in the stop list.

1. We will start with the Hive script file `t3-stoplist.hql`. This file currently just creates the two tables `myinput` and `mywords` from task 1 and dumps the output to the directory `task3-out`. You will modify this file in order to filter out a set of stop list words from the `mywords` table.

2. Create another single-column table called `stopwords`. Then read the `stoplist.txt` file ("Input_data/3/stoplist.txt") into your newly created `stopwords` table. Refer to how the `myinput` table was created if you are having trouble. Don't forget to put `DROP TABLE` at the beginning of the script for the added table, since we will be rerunning the script many times.
   Note: you can assume each separate word in the stop list is on a different line, so there is no need to split lines. Take a look at the file to verify it.

3. Execute the script, and then go into the Hive interpreter to execute `SELECT * ... LIMIT 10` on the `stopwords` table, to make sure it has the correct data.

4. Next, create an interim (temporary) table called `stopjoin` that contains two columns. The first column is called `mword` and the second column is called `sword`. Take a look at Table 1 below for an example of what the `stopjoin` table should look like. The first column (`mword` column) of the stopjoin table just contains all the words inside the `mywords` table. For each `mword`, the second column, `sword`, contains the matching stop word. If an `mword` does not exist in the `stopword` table, then its corresponding `sword` is `NULL`.
   Your job now is to create the `stopjoin` table. You will need to `JOIN` the `mywords` table with the `stopwords` table to create the `stopjoin` table. Since we want to keep all the words in the `mywords` table, even the ones that do not match the `stopwords`, you need to use the `OUTER JOIN`. See below for example join syntax (note: you need to substitute the right names for col1, col2 and table1 and table2):

```
SELECT <table1.col1> AS mword, <table2.col1> AS sword
FROM <table1> LEFT OUTER JOIN <table2>
  ON (<table1.col1> = <table2.col1>);
```

mywords	stopwords
the	a
treasure	is
is	the
my	
treasure	

stopjoin	
mword	sword
the	the
treasure	NULL
is	is
my	NULL
treasure	NULL

Table 1: Example mywords and stopwords tables, and the expected stopjoin table.

5. Execute the script, and again check the table contains the correct information by using the Hive interpreter to do SELECT * ... LIMIT 10 on the stopjoin table.

6. Currently the stopjoin table contains rows for blank words (empty strings). We can count how many of these rows there are by running the following query in the Hive interpreter:

```
SELECT COUNT(1) FROM stopjoin
WHERE mword LIKE "";
```

You should see that there are over 50,000 rows for useless blank words! We will now prevent your script from adding these rows to the stopjoin table. To do this, add a WHERE clause to the end of the CREATE stopjoin ... query that only keeps the words that do not match the empty string, "":

```
WHERE mywords.word NOT LIKE "";
```

Run your updated script, then use the Hive interpreter to count the rows with blank words again. This time the count should be 0.

7. To get a better idea of what is in the stopjoin table, do the following in the Hive interpreter:

   (a) Select the first 10 lines where mword is "the". The result should be 10 rows where both columns have the word "the", since "the" is a stop word.

   (b) Select the first 10 lines where mword is "help". The result should be 10 rows where the first column has "help" and the second row has NULL, since "help" is not a stop word.

8. Next, create a new table called stoplistOut, which contains only the rows in the stopjoin table where the second column (sword) is NULL. These are the words that we want to keep, since they are not stop words. The syntax for selecting null values is as follows: WHERE <*col*> IS NULL. The stoplistOut table **should only contain a single column**, which includes each kept mword. See Table 2 for the contents of the stoplistOut table for our running example. Take a look at the contents of the table in the Hive interpreter to make sure it contains the correct information. Again try looking for words "the" and then "help" and see if the result is what you expect.

stoplistOut	
`mword`	`sword`
treasure	NULL
my	NULL
treasure	NULL

Table 2: The `stoplistOut` table only considers rows from `stopjoin` where `sword` is `NULL`.

Exercise 2. Extend the `stoplistOut` query to produce word counts for each unique word. Table 3 shows the contents of the new `stoplistOut` table for our running example.

1. The `stoplistOut` table should have two columns, `mword` and `count`. You can obtain the count by using `COUNT(1)` and `GROUP BY`. Refer to the Hive documentation on GROUP BY if you need to.
2. Sort the data in descending order according to `count` and ascending order in terms of `mword`. This is very similar to task 2.
3. Limit the output to 10 rows.
4. Edit the "Dump output to file" part of the script to save the table `stoplistOut` instead of `mywords`.

Run the program and compare the output with that of task 2. You should see many of the top words from task 2 are absent from the output of task 3, as these were included in the stop list file.

stoplistOut	
`mword`	`count`
treasure	2
my	1

Table 3: The updated `stoplistOut`, which contains word counts instead of duplicates.

Task 4: Include list

An include list is the inverse of a stop list. That is, an include list contains all of the words that we want to *keep* rather than all of the words that we want to *remove*. For example:

Word list	Include list	Final list (with count)
big	fish	fish (2)
fish	eat	eat (1)
eat	giraffe	
other		
fish		

Exercise 3. Copy your completed `t3-stoplist.hql` from Exercise 2 to a new file called `t4-includelist.hql`. Modify the script so that it now loads data from the file located at "Input_data/4/includelist.txt" and saves output to "task4-out". Now it is your turn to show what you are capable of. Modify the program so that it now produces the desired output. Remember the output needs to include the count of the included words and sorted according to the same criteria as task 2 and 3. Hint: if you change the **left outer join** to an **inner join**, you will not need to check for null values.

Task 5: Sorting

`SORT BY`, like `ORDER BY`, is a clause used in queries to tell Hive that it should perform some sorting on the data collection. However, the difference between the two is that `ORDER BY` guarantees total global order in the output by enforcing only one reducer, while `SORT BY` only guarantees ordering of the rows within each reducer, as it uses multiple reducers. While this may not order the data perfectly, it is generally more efficient. You will now experiment with this.

1. In order to see a difference between `SORT BY` and `ORDER BY` we need to have multiple reducers. By default Hive uses just one reducer. Look at `t5-orderBy.hql` to find the line where we set the number of reducers to 2.

   ```
   set mapred.reduce.tasks = 2;
   ```

2. Currently, the `t5-orderBy.hql` script uses `ORDER BY` to perform sorting. Run the script and take a look at the output file. You should notice that all of the data is globally sorted by ascending count order.

3. Now copy `t5-orderBy.hql` into a new file called `t5-sortBy.hql`. Modify `t5-sortBy.hql` so that it uses `SORT BY` instead of `ORDER BY`. Also modify the output directory name to `task5sortBy-out`.

4. Execute `t5-sortBy.hql` and look at the output. You should see that the output is effectively two sorted lists concatenated one after the other. This is because `SORT BY` only sorts internal to the reducer, and in this script we set two reducers. `SORT BY` allows us to achieve more parallelism during reduction (and is therefore faster), but does not produce a globally sorted order. Whereas `ORDER BY` sorts the data using a single reducer (regardless of how `mapred.reduce.tasks` is set), hence it can produce a globally sorted order.

5. Modify the number of reducers to 5 for `t5-sortBy.hql` and run it again to see what happens.