CSE3BDC/CSE5BDC
Lab 02: 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

Note: to get full marks for this lab complete tasks 1 to 5. Completing task 6 will give you one extra bonus mark so you will get 3 marks rather than the usual 2 marks for labs.

Task 1: Gitlab and Cloud X Labs

Note all the labs must be completed using Cloud X Lab and when you finish your work you need to commit and push your work to git lab in order to get the labs marked. In this task you will use Gitlab and Cloud X Labs:

1. The instructions for task 1 can be found in two different formats:
    1. List of instructions:
        - In Labs tab in LMS look for "Instructions for git lab and cloud x for lab 2 onwards"
    2. Demonstration video:
        - Can be found in ECHO 360 under collection "Lab related videos" (click on arrow to expand drop down folder) the video is called "Git lab and cloud X instructions needed for lab 2 onwards
2. Following the above instructions you will be able to do the following two main tasks
    1. Create a private repository in the Gitlab and upload Lab 2 files using Jupyter as shown in the video.
    2. Make your initial commit and push it to the origin master.

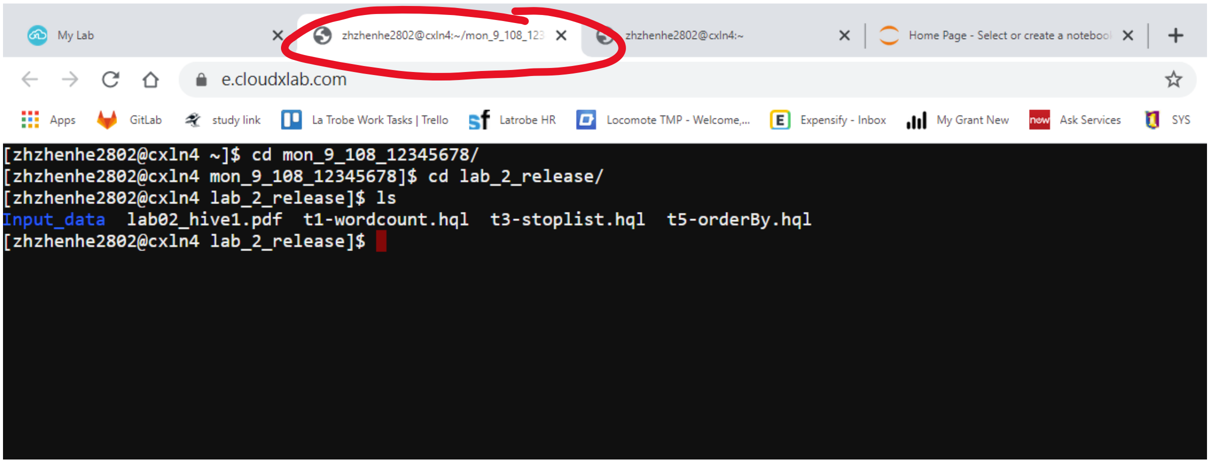
Task 2: Word count

For task 2 to task 6 you can watch demonstration videos in ECHO under the video labelled as "Demo of lab 2, tasks 2 to 6". Note the demo does not give you the solution to the exercises. You will still need to those yourself.

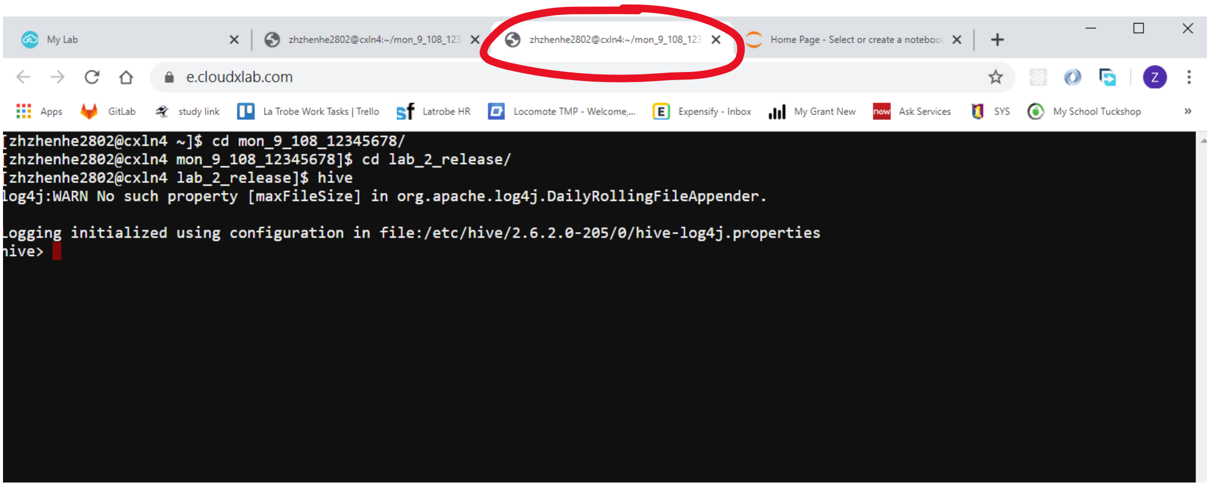
The task of counting how frequently words appear in a corpus of documents is commonly used as an introduction to big data processing. So, surely enough, our first exercise with Hive will be counting words!

1. Open two web consoles one for hive interpreter and the other is for linux command line. Also open the Jupyter notebook for editing the files. See diagrams below on what you should have opened.

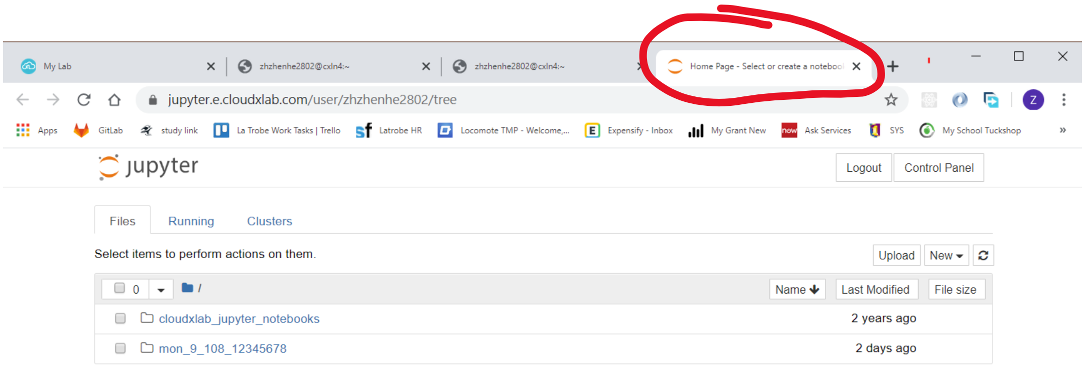
   Here is the linux web console:



   Here is the hive web console:



   Here is the hive web Jupyter notebook for editing your files:



2. Navigate to the Lab files directory where you can find `.hql` files. These `.hql` files contain HiveQL code, which is a dialect of SQL used by Hive. So screen shot above to see how to navigate to the right directory.

3. In the first console type the following to get into the Hive interpreter (you can keep this terminal open to use the hive interpreter at anytime):

   ```
   $ hive
   ```

4. List all of the tables in Hive:

   ```
   SHOW TABLES;
   ```

   You should see a lot of tables created by other users. In order to distinguish between your tables from others, you should insert your student id in front of the table name. We will show you how in the next step

5. The file `t2-wordcount.hql` is already setup to rename all the tables with your student id as prefix. You just need to first set a variable called `studentId` to your own student ID and all the tables will automatically be created with your student ID as prefix.

   1. In Jupyter modify the file `t2-wordcount.hql` by replacing the `12345678` with your student ID.

      ```
      set hivevar:studentId=12345678;
      ```

      As shown on the gitlab and cloud x instructions. In jupyter change Language to SQL so you get nice colour coding when editing the file.

   2. Whenever you create or drop tables make sure to use the following notation:

      ```
      ${studentId}_<table name>
      ```

6. Go to the linux web console you created. Run the first HiveQL script on it as follows (make sure you in the directory that contains the file):

```
$ hive -f t2-wordcount.hql
```

7. 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.

8. The program may take a short while to complete, so while you wait, take this opportunity look over the code of the t2-wordcount.hql file. The program creates the following three tables:

   - The ${studentId}_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 ${studentId}_mywords table. This table stores individual words extracted from the ${studentId}_myinput table. The table is created by expanding (using the EXPLODE function) each row of the ${studentId}_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 ${studentId}_wordcount table gets each word from the ${studentId}_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.

9. Once the program has completed successfully, the tables will be stored in Hive. Go into the Hive web console and see what each table contains by performing the following steps.

   - Type in the following to see the first 10 rows of the <student id>_myinput table (replace <student_id> with your student id):

     ```
     SELECT * FROM <student_id>_myinput LIMIT 10;
     ```

   - Repeat the above for the <student_id>_mywords and <student_id>_wordcount tables.

10. In the hive web console, lets list all the tables that belong to yourself using the following command. Replace <student id> with your own student.

    ```
    show tables like '<student id>*';
    ```

11. Please use the following command on the linux web console **whenever** you have created **new** tables. The command makes sure that other students can not see the tables you

create. This is especially important for the assignment. Replace `<student_id>` with your own student id.

```
hdfs dfs -chmod 700 /apps/hive/warehouse/<student_id>_*
```

12. Try to run the Hive script again in the linux console using the same command and see what happens.

```
$ hive -f t2-wordcount.hql
```

13. 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 `t2-wordcount.hql` script.

```
DROP TABLE ${studentId}_myinput;

DROP TABLE ${studentId}_mywords;

DROP TABLE ${studentId}_wordcount;
```

14. Once the program has completed successfully, browse to the `task2-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.

15. 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 ${studentId}_wordcount;` line:

```
ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t'

STORED AS TEXTFILE
```

Task 3: Subqueries and Hive MapReduce analysis

Using subqueries

Let's try to redo the word count program using just two tables: `${studentId}_myinput` and `${studentId}_wordcount`. We will do this by writing the

${studentId}_mywords table as a subquery within the ${studentId}_wordcount table.

1. Copy the file t2-wordcount.hql to t3-wordcount.hql so that we don't have to start from scratch.

2. Look at figure 1 below while following these instructions. In the command to create the ${studentId}_wordcount table (which begins with the line CREATE TABLE ${studentId}_wordcount AS), replace the line

   FROM ${studentId}_mywords

   with

   FROM (<subquery>) ${studentId}_splitwords

   where <subquery> is the second and third line from the command to create the ${studentId}_mywords table (SELECT EXPLODE ... FROM ${studentId}_myinput). This modification makes the ${studentId}_wordcount table take its input from the result of a subquery instead of from the table ${studentId}_mywords. The ${studentId}_splitwords is just a name we give to the table created by the subquery, it can be any valid name.

File Edit View Language

```
set hivevar:studentId=12345678; --Please replace it with your student id

CREATE TABLE ${studentId}_myinput (line STRING);

-- Load the text from the local filesystem
LOAD DATA LOCAL INPATH './Input_data/2/'
  INTO TABLE ${studentId}_myinput;

-- Table containing all the words in the myinput table
-- The difference between this table and myinput is that myinput stores each line as a separate row
-- whereas mywords stores each word as a separate row.
CREATE TABLE ${studentId}_mywords AS
SELECT EXPLODE(SPLIT(LCASE(REGEXP_REPLACE(line,'[\\p{Punct},\\p{Cntrl}]','')),' ')) AS word
FROM ${studentId}_myinput;

-- Create a table with the words cleaned and counted
CREATE TABLE ${studentId}_wordcount AS
SELECT word, count(1) AS count
FROM (<subquery>) ${studentId}_splitwords
WHERE word NOT LIKE ""
GROUP BY word;

-- Dump the output to file
INSERT OVERWRITE LOCAL DIRECTORY './task2-out/'
  SELECT * FROM ${studentId}_wordcount;
```

Figure 1

3. Modify the t3-wordcount.hql script so that the output is saved to task3-out instead of task2-out.

4. Execute `t3-wordcount.hql` (using `hive -f t3-wordcount.hql` in the linux web console) and check that you get the same output as for Task 2.

5. Now let's do a small experiment comparing the efficiency of `t2-wordcount.hql` and `t3-wordcount.hql`.

    1. Open the job browser in Hue, if you don't how to open Hue Job Browser, please read the lab instructions document LMS called *"Instructions for Hue used in lab 2 and some later labsFile"*.

    2. While keeping the job browser open, first run `t2-wordcount.hql` and then `t3-wordcount.hql`. What do you notice?

    3. The processing for `t2-wordcount.hql` uses **three** separate MapReduce jobs:

        1. MapReduce job 1: Create the `${studentId}_mywords` table.

        2. MapReduce job 2: Create the `${studentId}_wordcount` table.

        3. MapReduce job 3: Output the `${studentId}_wordcount` table to the local directory.

    4. The processing for `t3-wordcount.hql` uses **two** separate MapReduce jobs:

        1. MapReduce job 1 : Create the `${studentId}_wordcount` table.

        2. MapReduce job 2: Output the `${studentId}_wordcount` table to the local directory.

    5. If you add up the total time taken by jobs for each of the two scripts, you should see that `t3-wordcount.hql` is faster. This is because by having one less MapReduce job `t3-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 `${studentId}_wordcount` data ordered by the count in descending order (that is, the words with the most occurrences will appear first). As a secondary order, make the words also appear is ascending order.
Hint: You may want to use the familiar SQL syntax shown below at the end of the query which creates the `${studentId}_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 `t3-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 4: 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 `t4-stoplist.hql`. This file currently just creates the two tables `${studentId}_myinput` and `${studentId}_mywords` from task 2 and dumps the output to the directory `task4-out`. You will modify this file in order to filter out a set of stop list words from the `${studentId}_mywords` table.

2. Create another single-column table called `${studentId}_stopwords`. Then read the `stoplist.txt` file ("`Input_data/4/stoplist.txt`") into your newly created `${studentId}_stopwords` table. Refer to how the `${studentId}_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 `${studentId}_stopwords` table, to make sure it has the correct data.

4. Next, create an interim (temporary) table called `${studentId}_stopjoin` that contains two columns. The first column is called `mword` and the second column is called `sword`. Take a look at Table 1 below for an example of what the `${studentId}_stopjoin` table should look like. The first column (`mword` column) of the `${studentId}_stopjoin` table just contains all the words inside the `${studentId}_mywords` table. For each `mword`, the second column, `sword`, contains the matching stop word. If an `mword` does not exist in the `${studentId}_stopword` table, then its corresponding `sword` is `NULL`.
Your job now is to create the `${studentId}_stopjoin` table. You will need to `JOIN` the `${studentId}_mywords` table with the `${studentId}_stopwords` table to create the `${studentId}_stopjoin` table. Since we want to keep all the words in the `${studentId}_mywords` table, even the ones that do not match the `${studentId}_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>);
```

${studentId}_mywords	**${studentId}_stopwords**
the	a
treasure	is
is	the
my	
treasure	

${studentId}_stopjoin	
mword	**sword**

the	the
treasure	NULL
is	is
my	NULL
treasure	NULL

Table 1: Example ${studentId}_mywords and ${studentId}_stopwords tables, and the expected ${studentId}_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 `${studentId}_stopjoin` table.

6. Currently the `${studentId}_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 (Hive web console):

```
SELECT COUNT(1) FROM ${studentId}_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 `${studentId}_stopjoin` table. To do this, add a `WHERE` clause to the end of the `CREATE ${studentId}_stopjoin ...` query that only keeps the words that do not match the empty string, `""`:

```
WHERE ${studentId}_mywords.word NOT LIKE "";
```

   Run your updated script, then use the Hive interpreter (same as step 6 above) 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 `${studentId}_stopjoin` table, do the following in the Hive interpreter:

   1. 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.

   2. 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 ${studentId}_stoplistOut, which contains only the rows in the ${studentId}_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 ${studentId}_stoplistOut table **should only contain a single column**, which includes each kept mword. See Table 2 for the contents of the ${studentId}_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.

${studentId}_stoplistOut	
Mword	sword
Treasure	NULL
My	NULL
Treasure	NULL

Table 2: The ${studentId}_stoplistOut table only considers rows from ${studentId}_stopjoin where sword is NULL.

Exercise 2. Extend the ${studentId}_stoplistOut query to produce word counts for each unique word. Table 3 shows the contents of the new ${studentId}_stoplistOut table for our running example.

1. The ${studentId}_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 3.

3. Limit the output to 10 rows.

4. Edit the "Dump output to file" part of the script to save the table ${studentId}_stoplistOut instead of ${studentId}_mywords.

Run the program and compare the output with that of task 3. You should see many of the top words from task 3 are absent from the output of task 4, as these were included in the stop list file.

${studentId}_stoplistOut	
mword	**count**
treasure	2
my	1

Table 3: The updated `${studentId}_stoplistOut`, which contains word counts instead of duplicates.

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.

Task 6: Include list (Bonus Task)

8. 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		

9. Copy your completed `t4-stoplist.hql` from Exercise 2 to a new file called `t6-includelist.hql`. Modify the script so that it now loads data from the file located at "`Input_data/6/includelist.txt`" and saves output to "`task6-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.