CS 327E Final Project: Milestone 3

Prerequisites:

1. Completed Milestone 2.
2. Continuing to work with your partner.

Step 1. Download **the-numbers** dataset from this link: http://cs327e-fall2017-final-project.s3.amazonaws.com/the-numbers-with-headers.zip. This dataset contains 36 files (a-z and 0-9). Open a few files and become familiar with the structure of the dataset.

Step 2. Open a **psql** session and connect to your Postgres RDS. Create a new table **Title_Financials** in your IMDB database. This table will store the budget and box office numbers for a given title. The title should be stored as the usual **title_id**, representing the IMDB identifier for a title. The budget and box office values should be stored as `int` types and named **budget** and **box_office**, respectively. Create the primary key for this table, but do not create a foreign key constraint yet as this would slow down the data load. Copy the create table statement to a file named **create_title_financials.sql**. Add this file to your git repo.

Step 3. Download the starter code for this milestone from our snippets repo. The code is in a single file **movie_financials.py** accessible from this link: https://github.com/cs327e-fall2017/snippets/blob/master/movie_financials.py. Open the script in a Python editor of your choice and read through the code.

Step 4. Implement the **parse_line()** function for `map`. The function takes as input a single line from **the-numbers** files and parses the line as follows: extracts the year component from the **Release Date** column, the title from the **Movie** column, the genre from the Genre column, the budget from the **Production Budget** column, and the box_office from the **Box Office** column of the line.

Convert the title to upper case and remove all leading and trailing whitespace characters. Also encode the title as utf-8 using `string.encode('utf-8')` as several titles are foreign and contain international characters.

For the **genre** field, remove all leading and trailing whitespace characters. Since the genres have slightly different values in the database, also perform the following translations: If the genre == "Thriller/Suspense", set it to "Thriller". If the genre == "Black Comedy", set it to "Comedy". If the genre == "Romantic Comedy", set it to "Romance".

For the **budget** and **box_office** fields, remove the "$" and "," and "\"" characters from the values. Also, remove all leading and trailing whitespace characters. Cast the variable to a type `int` and set the value to -1 if it is empty.

Pass this function to **map** and save the output RDD as **mapped_rdd**. Verify the contents of **mapped_rdd** using the provided **print_rdd()** function.

Step 5. Implement the **save_to_db()** function for `foreachPartition`. The function takes as input a list of tuples where each tuple contains the elements of the mapped RDD. For each tuple in the list, the function should query the database to see if one or more **title_id** values exists for the year and movie being processed. Note that since the movie title was converted to upper case by the `map` phase, the **primary_title** field in the database should also be converted to upper case using the Postgres `UPPER()` function.

The number of **title_id** values returned by the query determines the next step in the logic. If there is a single **title_id** value returned, then proceed directly to inserting a record into the **Title_Financials** table using the **title_id** retrieved and the **budget** and **box_office** values from the tuple.

If the number of **title_id** values retrieved is greater than 1, then there is a nested conditional statement that refines the database query. If the **box_office** value is greater than zero, add a filter to the original query that excludes any records which are TV episodes. If the **box_office** value is **not** greater than zero, then add another condition to the original query that matches on the **genre** value in addition to the title and year.

Once the refined query has been executed, the returned value is evaluated. If a **title_id** is returned by the refined query, use this value to write the new record into the **Title_Financials** table along with the **budget** and **box_office** values from the tuple. If no **title_id** is returned, then no database insert should be done.

Step 6. Create an EMR cluster by cloning a previously terminated cluster. Configure the cluster and copy your script to the master node. Run the Spark job and debug any errors encountered during execution. Once the code has no syntax errors and is correctly adding records to the database, add your error-free **movie_financials.py** script to your git repo.

Step 7. At this point, your Spark job should be functional, but it is also excruciatingly slow. If you check the record count of the **Title_Financials** table, you will see that the job is performing < 5 writes per second. Stop the long-running Spark job (Control-C) since we are not interested in letting it run for hours.

Instead, we are going to work on optimizing the job. Review the select statements that are being run by the job and think of an index that might help speed up each statement. Please read the Postgres documentation page [5] to review the `create index` syntax and become familiar with some of the optional parameters. Also, read the documentation on `partial indexes` [6] and pay particular attention to Example 11-2 which shows how to exclude uninteresting values. Hint: indexes can contain expressions, so they can store the movie titles in upper case letters.

Formulate an index for each select statement and create the indexes in **psql**. Now use the `explain` command to generate a query plan. Note that the select statement must have a value for each query parameter. Generate a query plan for each query and review the output.

Ensure that the query plan is doing an index scan using the intended index (as opposed to a sequential scan of the whole table). If an index does not appear in the query plan, drop it from the database and try to come up with an alternate index that is more specific to the search criteria. Hint: an index may be relevant to more than one query.

Once the indexes have been created, use the `\timing` option in psql to check the runtime of each query. This is an important step because an index can sometimes hurt performance. If an index is not providing any time savings, drop it from the database.

Save the create index statement for each verified index. Place the statements in a file named **create_indexes.sql**. Similarly, for each query, save the explain output that contains the index scan and copy it to a file named **explain.out**. Add both files to your git repo.

Truncate the **Title_Financials** table to avoid primary key violation errors. Now re-run the Spark job with the new indexes in place. You should notice an immediate speedup if the indexes are relevant. The actual runtime for the optimized job will vary, but it should complete in less than 30 minutes.

Step 8. Connect to your IMDB database and retrieve the number of records from the **Title_Financials** table. The number should be in the 15,000 range. Copy the SQL query and its output into a new file. Save the file as **title_financials.out** and add it to your git repo.

Step 9. Create a `foreign key` on the **title_id** column of the **Title_Financials** table. The `foreign key` should point to the **title_id** column of its parent table. The ALTER TABLE command page [4] in the Postgres manual has an example on how to add a foreign key constraint to an existing table. Copy the foreign key statement to a file named **alter_title_financials.sql**. Add this file to your git repo.

Step 10. Write an aggregate query that accesses the new table in some interesting way and wrap this query inside a view. This query should join **Title_Financials** with **Title_Basics**. Name this view **v_title_financials**. The view should be virtual if it executes in < 10 seconds or materialized if takes longer. Create the view in your IMDB database and add the view definition to a file. Name the file **v_title_financials.sql** and add it to your git repo .

Step 11. Create a QuickSight analysis that visualizes the output from your view. Create a dashboard for the analysis and share it with the IAM admin user. Also, take a screenshot of the analysis and save it as a png or jpg format. Add the screenshot to your git repo.

Step 12. Locate the commit id that you will be using for your submission. This is a long 40-character that shows up on your main GitHub repo page next to the heading "Latest commit" (e.g. commit 6ca6f695bca36f7fc2c33485d1080ae30f8b9928). Locate the link to your GitHub repo (e.g. https://github.com/cs327e-fall2017/xyz.git where xyz is your repo name). Go back to your existing Stache entry and locate the read-only API endpoint and read key.

Replace the commit id, repo link, API endpoint, and read key in the json string below with your own:

```
{
  "repository-link": "https://github.com/cs327e-spring2017/xyz.git",
  "commit-id": "6ca6f695bca36f7fc2c33485d1080ae30f8b9928",
  "stache-endpoint": "/api/v1/item/read/61515",
  "stache-read-key": "b2eacb0387a919e33b27e7c03a6c5d84b71234795732be33eb28711ec16f0e21"
}
```

Create a submission.json file that contains your modified json string. Click on the Final Project Milestone 3 in Canvas and upload submission.json. **Do not add submission.json to your git repo**.

This submission is due by **Friday, 11/17 at 11:59pm**. If it's late, there will be a 10% grade reduction per late day. This late policy is also documented in the syllabus.

Additional Notes:

- Remember to treat your EMR as a disposable resource as opposed to a persistent cluster. Terminate it whenever you are not doing any active development. Otherwise, you will quickly run out of AWS credits!

- Continue to develop your code locally, using **scp** to transfer each new version of the code to the EMR master node. This avoids the whole situation of losing your code when your cluster gets destroyed.

References and Additional Resources:

[1] Snippets Wiki: https://github.com/cs327e-fall2017/snippets/wiki
[2] Milestone 3 Grading Rubric: http://www.cs.utexas.edu/~scohen/projects/m3-rubric.pdf
[3] Spark Programming Guide: https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html

[4] Postgres ALTER TABLE command: https://www.postgresql.org/docs/9.6/static/sql-altertable.html
[5] Postgres CREATE INDEX command: https://www.postgresql.org/docs/9.6/static/sql-createindex.html
[6] Postgres Partial Indexes: https://www.postgresql.org/docs/current/static/indexes-partial.html