CS 327E Lab 2: Data Loader

Learning Objectives:

1. Team up with your partner on this project
2. Create a Postgres database on RDS
3. Create tables based on your IMDB database design from Lab 1
4. Populate tables with IMDB dataset
5. Transform raw data into usable data for analysis and reporting

Prerequisites:

1. Lab 1 complete
2. Lab 2 setup complete [1]

Steps Outlined:

1. Perform any changes to your Lab 1 submission that were noted in your team's graded rubric. This includes changing the cardinality of the relationships between, field lengths, datatypes, keys and other constraints. Ensure that the conceptual and physical diagrams are in sync and that the data dictionary is also in sync with the physical diagram. For example, if a change is made to the datatype in the physical diagram, it also needs to be made to the data dictionary. Once all the necessary revisions have been made to the Lab 1 submission, commit the changes and push them to the **lab1** folder on Github. You may now start on Lab 2.

2. Create a new folder in your local git repository called **lab2**. All the work you will do for this lab will go into this folder.

3. Create a new **database** named `imdb` on your Postgres RDS instance. Run the command: `create database imdb;` to create the database.

4. Write the **DDL** for the IMDB tables. The data dictionary from Lab 1 contains the details required to write the initial `create table` statements. Once you have those statements, review the layout of the csv files and ensure that for each create table statement, the sequence of columns in the table definition matches the layout of the source csv file. In particular, if a column exists in the csv file, but doesn't exist in the create table statement, it needs to be added to the table. This is an artifact of the method we are using to load the data into the tables (see step 5).

Remember that each table must have a single primary key and child tables must have one or more foreign keys. Also, remember that the table creation sequence is important, namely parent tables must be created before their child tables. Refer to the ticket create database script for an example [2]. The DDL for all the tables should be in a script named `create_tables.sql`.

Here are a few helpful psql commands:

- run the command `\c imdb;` to connect to the imdb database

- run the command `\i create_tables.sql;` to run the script
- run the command `\dt` to see a list of tables in the database
- run the command `drop database imdb;` followed by `create database imdb;` to drop and re-create the database as needed

Once the `create_tables.sql` has been debugged and run successfully without errors, commit it to your team's local repo and push the commit to your remote private repo on Github.

5. Write the required `\copy` commands to import the IMDB dataset into the database tables. Each copy command imports the data from exactly one csv file. Refer to the tickit load data script for an example [3].

As alluded to earlier, the copy command does not have an option to omit columns from the csv file during an import. Therefore, each database table must match the layout of the source csv file (e.g. the layout of the `Actors` table must match `actors.csv`, etc.). This means that you may end up some additional columns in the tables that were not part of the original schema design. You will get rid of those extra columns in step 7.

Place all the copy commands in a script called `load_data.sql`. Connect to the IMDB database and run the script using the command: `\i load_data.sql;` Fix any syntax errors and re-run the script until it is error-free. Commit the script to your team's local repo and push the commit to your remote private repo on Github.

6. Using SQL, perform some simple verification checks of the imported data. Check the record count of each table to ensure that it matches the output from the copy command. Check some sample values to ensure that all the columns from the csv files were imported into their designated columns in the table. If you notice an issue with the imported data, fix the create table statement and re-load the table.

Place the select statements used for these checks into a script named `verify_data.sql`. Note: you do not need to include the output from each select statement that was run, just the select statement. Commit the script to your team's local repo and push the commit to your remote private repo on Github.

7. Remove all the *empty* columns from the database. An empty column is one that contains only null values. Run the command: `select distinct abc from xyz;` where `abc` is the name of the column and `xyz` is the table name, to check if the column is empty. If the column is empty, use the following command to remove the column: `alter table xyz drop column abc;` where `xyz` is the table name and `abc` is the column name. In addition to empty columns, please remove any extra columns that were required for loading the data into the tables, but are not part of your schema design.

Once you have removed the appropriate columns, consult the Postgres manual [4] and alter the datatype for the Actor's gender column to `char(1)`. Similarly, alter the datatype for the Movie's type column to `char(3)`.

Place all of the `alter table` statements run into a script named `alter_tables.sql`. Commit the script to your team's local repo and push the commit to your remote private repo on Github.

8. Using SQL, perform the following transforms to the data:

- set `Actors.gender` to 'M' (for male) if its current value is 1.
- set `Actors.gender` to 'F' (for female) if its current value is null.
- set `Movies.type` to 'V' (for video) if its current value is 1.
- set `Movies.type` to 'VG' (for video game) if its current value is 2.
- set `Movies.type` to 'FF' (for feature film) if its current value is 3.
- set `Movies.type` to 'TVM' (for TV movie) if its current value is null.
- set `Movies.type` to 'TVS' (for TV show) regardless of its current value if its `idmovies` value also exists in the `Series` table. Hint: use an inner join to find the `idmovies` values that match in both tables.

All transforms should be done using a SQL `update` statement. Read pp. 432-433 from our Database Design for a refresher on this topic. Note that the `update` statement can also contain an embedded `select` statement in the `where` clause [5].

Place all the update statements in a script called `update_tables.sql`. Commit the script to your team's local repo and push the commit to your remote private repo on Github.

9. Update the **Stache** entry that you created in Lab 1. The entry should have the following content and format in the **secret** field:

```
{
 "aws-username": "shouldbeAdmin",
 "aws-password": "mypassword",
 "aws-console-link": "myconsolelink",
 "rds-endpoint-link": "myrdsendpoint_without_port_number",
 "rds-username": "shouldbeMaster",
 "rds-password": "myrdspassword"
}
```

Make sure to **not** include a port number with the RDS endpoint. We will assume the Postgres RDS instance is running on the default port of 5432.

Check the option called **allow this item to be accessed by read-only API calls**. Save the changes to the Stache entry.

10. Locate the **commit id** that you will be using for your team's 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 team's repo. This is the URL to your private repo on Github (e.g. https://github.com/cs327e-spring2017/xyz.git where xyz is your repo name). Go back to the 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/62021",
 "stache-read-key": "ec1f815a603234eb8c5e2c02b474839f0b6d3b9e76b103f1ab0463b655e6661b"
}
```

Create a file called **submission.json** that contains your modified json string.

Click on the Lab 2 Assignment in Canvas and upload submission.json. This submission is due by **Friday, 02/17 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. **Note: there should be one submission per team.**

Teamwork:

1. We will use 2 class meetings (02/13 and 02/15) to work on this lab.
2. We expect each team-member to contribute equally to the assignment and we will be checking the commit history to ensure that this is happening as expected.
3. We want you to use the Github Issue Tracker to assign and track the status of tasks.

Resources:

[1] Lab 2 Setup Guide: https://github.com/wolfier/CS327E/wiki/Setting-up-Lab-Two
[2] Lab 2 Grading Rubric: http://www.cs.utexas.edu/~scohen/assignments/rubric2.pdf
[3] Tickit sample code: https://github.com/cs327e-spring2017/snippets
[4] Alter table commands: https://www.postgresql.org/docs/9.6/static/ddl-alter.html
[5] Update table examples: https://www.postgresql.org/docs/current/static/sql-update.html