CS 327E Final Project – Spring 2017

Starting today, you will work on the final project through the end of the term. Just like with the labs sessions, you will work on the final project with your partner during class period. We will continue to have reading quizzes for assigned readings to ensure that we learn the relevant material for the project.

The goal for this project is to build a data warehouse. A data warehouse is a large analytical database that collects various independently produced datasets into a single repository. These independent datasets are then transformed such that they can be joined on their common attributes. We can write SQL queries against these transformed tables and produce reports that provide insight into a particular domain. A data warehouse typically receives incremental updates from its source databases on a daily basis so that the data in the data warehouse stays up-to-date. Nobody wants to look at reports that are produced from stale data!

For the final project, you will be building a data warehouse to collect a wealth of information about contemporary popular music. The three datasets that we will work with are Discogs [1], Million Song Dataset [2], and Music Brainz [3]. Discogs has information about music releases, artists, tracks, genres; Million Song has details on the artists, songs and lyrics; Music Brainz has information about music events, places, recordings, and releases. The three datasets have different formats, a few common attributes, and several unique attributes. The goal is to bring these datasets into a single database, create a unified schema that captures the unique entities and attributes from each dataset while letting us join the datasets, and then answer some interesting queries that wouldn't have been possible had we just examined one dataset.

Software Stack:

S3 will store the raw data for each dataset. Each dataset will be stored in a separate S3 bucket.

Amazon Redshift will be used for the data warehouse.

The raw data will be copied into Redshift directly from S3 using the copy command.

SQL will be used to perform transforms on the raw data and to populate the unified schema.

SQL will be used to write the analysis queries.

Amazon QuickSight will be used to create interactive reports.

QuickSight reports will be produced from SQL queries run against the Redshift tables in the unified schema.

Git will be used for version control of the code through the project. GitHub and GitHub Issue Tracking will be used for collaborating with your partner and for grading project milestones.

LucidChart will be used for diagramming the database design (both staging schemas for each database and unified schema).

Project Milestones:

All project milestone submissions are due at 11:59pm on the due date listed unless otherwise noted. Late submissions will be penalized by 10% per late day. Milestones must be submitted to receive a non-zero project grade.

M1. Environment Setup due Friday 03/24. Follow the steps in the Final Project Setup Guide [4]. Once the setup is complete, connect to your group's Redshift cluster from your local psql client and take a screenshot of your connection test. There should be two connection tests, one per group member. Make note of any issues that your group encounters for this milestone in a m1-readme.txt file.

Create a **final-project** folder in your git repo and add the two screenshots and optional m1-readme.txt to this folder. Commit and push the changes to your remote repo on GitHub.

Open your Stache entry from Lab3 add the elements shown in bold:

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

Please use JSONLint [5] or an equivalent checker to ensure that your JSON document is properly formatted. Create a **submission.json** file using the same format as from Lab 3. If you are copying the one you used for Lab 3, remember to update the **commit id** with the one you are submitting for this milestone. Select the **Final Project Milestone 1** assignment in Canvas and upload **submission.json** by the milestone deadline.

M2. Database Design due Friday 03/31. Download the 3 dataset subsets [7] and look at the sample data using Excel or another editor of choice. Read the documentation on each dataset by visiting their web sites [1, 2, 3]. For example, the schema for Music Brainz is available online [8]. Decide as a group which Entities and attributes you want to work with and make a list of the files you plan to load into the database. You must choose a **minimum of 30 files from 2 datasets**. Note that all song_metadata*.csv from Million Song count as only 1 file because they belong to the same Entity. You can work with all 3 datasets if you find something you like from each one, but you can't work with only one dataset.

Create a physical diagram for each dataset that you chose. You should only represent the files that you chose to work with (as opposed to all the files in the dataset). Note that some files are

very wide and have hundreds of columns in them (e.g. song_metadata_*.csv in Million Song has 388 columns!). You do not need to represent each attribute in the diagram when there are so many. Only capture the most prominent ones (i.e. the ones you plan to use).

Once you have produced a physical diagram for the datasets, think about the analysis that you want to be able to run over the combined datasets. Perhaps you want to analyze the common songs / artists across the datasets? Perhaps you want to analyze the releases by music genres or locations? Whatever they are, write up those queries in plain English in a queries.txt file. You should come up with 10-12 queries. These queries should have good coverage of the tables in the unified schema and consist a mix of single dataset and multiple dataset queries. They should also use a variety of SQL operations including aggregations, group-bys, joins, etc.

Using these analysis queries as a guide, design a conceptual diagram for the unified / combined schema. The unified schema represents how Entities relate not only within a dataset, but across datasets. In addition to new connections, the unified schema shows the results of parsing and transforming the raw data e.g. type casting varchar to date columns, removing unnecessary columns, combining tables that have one-to-one relationships, etc. All required transformations will be done as part of M4. For now, our goal is to produce a design that is suitable for answering the analysis queries that your group has identified.

Next, create a physical diagram and data dictionary for the unified schema. The physical diagram and data dictionary should have only the most prominent columns for each table. For each of those columns, provide a brief description of what the column is intended for (e.g. joining on table x, filtering, aggregating, grouping, etc.).

As you go through the design process, make note of any issues that your group encounters with in a m2-readme.txt file. These notes will come in handy when working on the technical report for M6.

All diagrams should be done in LucidChart. Download the diagrams as pdf files and copy them to the final-project folder in your repo along with the queries, data dictionary, and the optional m2-readme.txt file. Commit the changes and push them to GitHub. Create the standard **submission.json** file and upload it into the **Final Project Milestone 2** assignment in Canvas by the milestone deadline.

M3. Data Load due Friday 04/07. The goal is to create the staging tables and load the CSV data into these tables. Due to the number of tables that we are working with and the tedious work involved in hand-coding create table statements, we are going to automate some of this process using software:

Pull the script `generate_DDL.py` from our snippets repo [10]. This script takes as input a CSV file and produces a SQL file with the create table statement. The script determines the database column names from the CSV column headers. It determines the database types and lengths based on a single row of CSV data. It is important to review the output before creating the table since the script may not be able to determine the correct type and length based on one row.

To run the script, use the command: `python generate_ddl.py xyz.csv`, where `xyz.csv` is the input CSV file from which to generate a SQL file. An alternative option for Music Brainz, is

to pull the published DDL from their GitHub repo [11] and modify it for Redshift. Note that the Music Brainz DDL contains several types of constraints that are not supported by Redshift.

To keep the staging tables organized in the database, we are going to create a separate schema for each dataset. This is instead of storing all the staging tables in the default public schema.

To create a schema, use the command: `create schema discog;` where `discog` is the name of the schema. Once the schema is created, you can create the tables in the schema either by adding the schema name as a prefix to the create table statement (e.g. `create table discog.Releases;)` or by setting the search path to the appropriate schema: `set search_path=discog;`. Note: when you set the search_path, you do not need to qualify the table with the schema. To list all the schemas in the database, use the psql command: `\dn`. To query the tables from multiple schemas, set the search_path to those schemas using the command: `set search_path=discog,millionsong;`

There should be one main DDL script per dataset. This script should contain the following sequence of actions: drop the schema if it already exists, create the schema, and create all the tables in the schema. All this code can either be in one giant file or the main script can call the individual create table scripts. The main script should be named after the dataset: `create_discog.sql` for the Discog dataset, `create_millionsong.sql` for Million Song, and `create_musicbrainz.sql` for Music Brainz. Run the main script from the psql shell using the `\i` option and debug any problems until it runs error free.

Once all the staging tables have been created, write the copy command to load each staging table. The copy command takes several input parameters including:

bucket name: `cs327e-final-project-datasets`
folder name: `discog-csv, million-songs-csv, million-songs-metadata-csv, music-brainz-csv`

The `million-songs-metadata-csv` folder contains the files `song_metadata_A.csv – song_metadata_Z.csv` whereas `million-songs-csv` contains all the other Million Song files. Remember that all 26 `song_metata*.csv` files should be loaded into the same table.

The copy command for loading the Million Song file `words.csv` into the `Words` table is given below:

```
copy words from
's3://cs327e-final-project-datasets/million-songs-csv/words.csv'
iam_role 'arn:aws:iam::531237488045:role/redshift_s3_role'
region 'us-east-1'
csv quote '"' ignoreheader 1 trimblanks compupdate ON
maxerror 50;
```

The copy command for loading the `song_metata*.csv` files into the Song_Metadata table is given below:

```
copy song_metadata from
's3://cs327e-final-project-datasets/million-songs-metadata-csv/'
```

```
iam_role 'arn:aws:iam::531237488045:role/redshift_s3_role'
region 'us-east-1'
csv quote '"' ignoreheader 1 trimblanks roundec compupdate ON
maxerror 50;
```

Note that in the second example, we do not specify a file name since we want copy to load all files from the folder `million-songs-metadata-csv`.

The ARN string in the copy command provides Redshift the authorization to access S3. It is a unique string and you'll need to replace it with your own. You can use the IAM console to look up your ARN string. For an explanation of the other parameters used, see the COPY command reference page [11].

As with the DDL scripts, create one copy script per dataset: `copy_discog.sql` for the Discog dataset, `copy_millionsong.sql` for Million Song, and `copy_musicbrainz.sql` for Music Brainz. The script should set the `search path` to the appropriate schema before running the copy commands.

Run the copy scripts from psql using the \i option and debug the problems. If the copy command succeeds, it will return the number of records successfully loaded into the table as follows:

```
psql:C:/utcs_work/cs327e_spring_2017/final_project/datasets_subset/cop
y_millionsong.sql:50: INFO:  Load into table 'song_metadata'
completed, 1000000 record(s) loaded successfully.
COPY
dev=#
```

If the copy command fails, it will return an error like this one:

```
psql:C:/utcs_work/cs327e_spring_2017/final_project/datasets_subset/cop
y_millionsong.sql:50: INFO:  Load into table 'song_metadata'
completed, 1 record(s) could not be loaded.  Check 'stl_load_errors'
system table for details.
```

To debug the load problem, run the following diagnostic query:

```
select line_number, colname, type, raw_field_value, err_reason
from stl_load_errors
order by starttime desc;
```

Review the output and fix the problem. You will likely need to edit the DDL script to fix the problem. For example, if the error is `Overflow for NUMERIC` or `String length exceeds DDL length`, it means that the datatype length is too small for the value in the file. For a complete list of copy error codes and descriptions, see the COPY Load Error Reference page [12].

Each time the copy command fails, the load errors will be appended to the `stl_load_errors` system table. Since the query is sorting the records by timestamp descending, the errors from the most recent run will bubble up to the top.

Once you have successfully loaded all the staging tables, sample the records from each table to ensure that the data was loaded into the proper columns. Use the query: `select * from schema.table order by random() limit 10;` to retrieve 10 random rows of data. Store the output from each query in a text file. Again, there should be one file per schema/dataset. Name the files `check_discog.txt`, `check_millionsong.txt`, and `check_musicbrainz.txt`.

As you go through the data load process, make note of any issues that your group encounters in a m3-readme.txt file. These notes will come in handy later when you start working on the technical report for M6.

Add all the DDL scripts, copy scripts, check scripts, and the optional m3-readme.txt to your local git repo and commit and push the changes to GitHub. Create the standard **submission.json** file and upload it into the **Final Project Milestone 3** assignment in Canvas by the milestone deadline.

M4. Data Integration due Friday 04/14. The objective of this milestone is to create and populate the unified schema. Unfortunately, we can't simply copy the existing tables from the staging schemas to the unified schema in their current state. The raw data has some consistency problems which present some serious limitations for cross-dataset analysis and reporting. We will therefore spend some time transforming the data to make it less inconsistent. We will then copy to the unified schema only a vertical subset of the tables that includes the cleansed columns and finally we will test the analysis queries over the unified schema.

A basic outline of the work for this milestone is given below:

1. review analysis queries (M2 queries.txt) and identify the columns that will be used in cross-dataset joins as well as the columns that are used in other query operations such as aggregations, group-bys etc.

For each column that appears in a cross-dataset join:

2. sample 100-200 random values from this column to see if they contain "punctuation" characters. "Punctuation" characters are defined as:
   `;, /, (, [, :, -, ..., with, Vs.`
3. if the sampled values contain "punctuation" characters, add a new column `ccolumn` to store the cleansed column data. Use UDF to determine the length of `ccolumn`.
4. set `ccolumn` to current column (e.g. set cartist = artist, ctitle = title, etc.)
5. remove punctuation characters from `ccolumn` using `split_part()` and standardize the formatting using built-in functions `btrim()`, and `initcap()`.

For all other columns that appear in the analysis queries:

6. if column is of type varchar, but it represents a date or boolean or number (e.g. `discog.Releases.released`), add new column `ccolumn` with corrected type (e.g. `discog.Releases.creleased` of type date)
7. convert values from varchar to new type (e.g. varchar to date).

For each table accessed by the analysis queries:

8. identify subset of tables columns which are needed by the queries.
9. create table in `unified` schema based on the subset of columns, substituting the

uncleansed columns with the cleansed `ccolumns`.

For all analysis queries:

10. translate query description (queries.txt) to SQL using the tables in `unified` schema
11. run SQL query and verify its output.
12. refine and iterate until you have produced 10 SQL analysis queries that are suitable for reporting.

A few implementation details for these steps are given below. Please feel free to use this section as a reference.

For step 1, review your analysis queries and make a list of the tables and columns that you'll need to answer those queries. Feel free to use the data dictionary from M2 to prepare this inventory.

For step 2, sample 100-200 records from the staging tables using: `select column from table order by random() limit 50;` If the output contains any of the following "punctuation" characters: `;, /, (, [, :, -, ..., with, Vs.,` perform steps 3-5 to standardize the data. If no "punctuation" characters were found, proceed to step 6.

For step 3, get the actual used bytes for the varchar column in question to determine what the appropriate size of the new `ccolumn` should be. Define a user-defined function (UDF) to retrieve the number of bytes as no built-in function in Redshift exists for this task.

```
create or replace function get_utf8_bytes(col varchar(max))
returns int
 stable AS $$
 import sys
 reload(sys)
 sys.setdefaultencoding('utf-8')
 return len(col.encode('utf-8'))
$$ language plpythonu;
```

This UDF is also available from our snippets repo [13]. Once the UDF has been created, call it as follows:

```
select max(get_utf8_bytes(column)) from schema.table;
```

For example:

```
select max(get_utf8_bytes(title)) from discog.Releases;
```

The output from the UDF determines the length of the new varchar column `ctitle`:

```
dev=# select max(get_utf8_bytes(title)) from discog.Releases;
 max
-----
 290
(1 row)


dev=#
```

```
alter table discog.Releases add column ctitle varchar(290);
```

As you'll see, the Python UDF is very slow compared to the built-in SQL functions in Redshift. It takes ~30 minutes to run on a table with 20 million records. That is why we are going to use a UDF only when a SQL function doesn't exist. Fortunately, for us, this is the only UDF we are going to use.

For step 4, first populate `ccolumn` with the values from the current column using a simple update statement. For example:

```
update discog.Releases set ctitle = title;
```

For step 5, remove the substrings beginning with the "punctuation" characters found in step 2. Use the function `split_part()` to remove those characters. The following example shows how to remove the substring starting with `' -'` from `discog.Releases.ctitle`, the output of which is trimed, and then capitalized:

```
update discog.Releases set ctitle = initcap(btrim(split_part(ctitle,
' -' , 1)));
```

Notice that when removing the hyphen, we add a leading space to ensure that we don't chop off valid titles that happen to contain hyphens. For example, we want to keep the title 'Wide-Eyed Angel' as is, but we want to replace the title 'Harthouse Compilation Chapter 2 - Dedicated To The Omen' to 'Harthouse Compilation Chapter 2'.

Another observation is that the `split_part()` function does not support multiple patterns in one invocation, it needs to be called once for each punctuation character. An alternative to `split_part()` is to use `regexp_replace()` which takes a regular expression. Only use this function if you are comfortable working with regular expressions.

For steps 6 and 7, to convert a varchar to a date type (e.g. `discog.Releases.released`), parse each component of the date (e.g. year, month, day) and store it into its own varchar field. Next concatenate the 3 fields and cast the output to a date type using `to_date()` [14]. The following example demonstrates how to extract the year from the `released` column component and store the output in a `year` column:

```
alter table discog.Releases add column year char(4);

update discog.Releases set year = btrim(split_part(released, '/' ,
3));

update discog.Releases set year = btrim(split_part(released, '-' , 1))
where length(btrim(split_part(released, '-' , 1))) = 4 and year = '';
```

The month and day components can be extracted and stored using the same technique as year:

```
alter table discog.Releases add month year char(2);
alter table discog.Releases add day year char(2);

update Releases set month = split_part(released, '-', 2);
```

```
update Releases set month = split_part(released, '/', 1) where
length(split_part(released, '/', 1)) <= 2 and month = '';

update Releases set day = split_part(released, '-', 3);
update Releases set day = split_part(released, '/', 2) where
length(split_part(released, '/', 2)) <= 2 and day = '';
```

Finally, convert to a date and store the results in the new column `creleased`:

```
alter table discog.Releases add column creleased date;

update Releases set creleased = to_date(day || '-' || month || '-' ||
year, 'DD-MM-YYYY') where day <> '' and month <> '' and year <> '';
```

Include all the error-free alter table and update statements in a transform script organized by schema (`transform_discog.sql`, `transform_millionsong.sql`, `transform_musicbrainz.sql`). The transform scripts should contain only the alter table and update commands, not the select statements that were run as part of steps 2 and 3.

For step 8, return to the inventory that you prepared in step 1. Update the inventory as follows: Replace all the columns with "punctuation" characters with the cleansed columns `ccolumns` created in steps 3-4.

For step 9, create a new schema called `unified` where you will store the tables needed to answer the analysis queries. To create a table based on a subset of the columns, use a create-table-as-select statement. The following example shows how to do this for the `discog.Releases` table:

```
create table unified.D_Releases as select release_id, ctitle as title,
num_tracks, creleased as released, country from discog.Releases;
```

Note the columns renamings from `title` to `ctitle` and from `creleased` to `released` fields. Also, note the table renaming from `Releases` to `D_Releases`. Since we want to keep track of which dataset a table came from, prefix the table name with a dataset abbrevation (e.g. you can use `d` for discog, `ms` for millionsong and `mb` for musicbrainz). Keep track of all the create-table-as-select statements for the `unified` schema in a `create_unified.sql` script.

For steps 10-12, translate the query descriptions from M2 into SQL queries using the tables in the `unified` schema. Ensure that you have a good mix of queries containing a diverse set of operations as well as a diverse set of tables. Finally, ensure that you have a few queries that span across datasets. Test and verify the results for each query. If the query produces some unexpected results, debug the query by breaking it up into smaller units, review the output from each unit, and correct the problem by rewriting the query and/or performing additional transforms. Put all the validated queries into a `queries.sql` script. This script will be used for creating the reporting views in milestone 5.

If changes were made to the design of the unified schema during this milestone, please update the unified schema diagrams and data dictionary so that the documentation is in sync with the code. Also, make note of any issues or unexpected challenges that your group encountered during this milestone in a `m4-readme.txt` file.

Commit the following files to your local and remote code repository: `transform_*.sql`, `create_unified.sql`, `queries.sql`, and the optional `m4-readme.txt`. Create the standard **submission.json** file and upload it into the **Final Project Milestone 4** assignment in Canvas by the milestone deadline.

M5. QuickSight Visualizations due Friday 04/21. There are two parts to this milestone. The first is to create the database views and the second is to create a visualization for those views in QuickSight.

Part 1:
Create a virtual view for each analysis query developed in M4. These views should be created in the `unified` schema. Give each view a descriptive name with a prefix of `v_` for ease of identification (e.g. `v_artists_from_iceland`). The basic command for creating a view is: `create or replace view $view_name as $analysis_query;` (where `$view_name` = the name of the view and `$analysis_query` = select statement for the query from `queries.sql`).

Once you have created the view, check the runtime of the view as follows:

```
\timing
\pset pager
select * from $view_name;
```

The output returns the execution time in milliseconds. The view should be interactive and return some initial results within a few seconds. If the view is executing for longer than 30 seconds, you should tune the view's query to reduce the runtime.

The first optimization you can perform is to pre-sort the tables being accessed by specifying a sort key. Pre-sorting the table(s) can help speed-up several operations including group-by, joins, and order-bys. However, if you have a massive query, pre-sorting may not be enough to obtain a runtime of <= 30 seconds. The second and more aggressive optimization is to pre-compute one or more of the join(s) performed by the query. This means creating a derived table that stores the results from the join(s) and then changing the analysis query to access the derived table. The third optimization technique is similar to the second and involves pre-computing the group-by(s) and aggregation(s). The goal is to pre-compute as few operations as possible while obtaining the desired runtime. In order to choose which operation to pre-compute, review the execution plan for the problem query from the Redshift Dashboard. Identify and pre-compute the higher cost operations first, then re-run the view to check the new runtime.

Once you have a sufficiently fast view, place the create view statement (and any create table and insert into table statements) in a `create_views.sql` file. Add a comment above the view to report the run-time obtained during your testing. For example:

```
-- Time: 5252 ms
create view v_artists_from_iceland as select …
```

The `create_views.sql` file should have a total of 10 views, one for each analysis query.

Part 2:
Follow the setup steps for Amazon QuickSight as documented in the QuickSight Setup Guide [15]. Make sure that your QuickSight account name is as close as possible to your group name (add a suffix if the exact name is not available). Once you have created a data source for Redshift and successfully connected to your Redshift database, start creating the visualizations in QuickSight.

Create a visualization (or what QuickSight calls a "visual") for each of the 10 views that you created in part 1. QuickSight requires that a dataset be created for each accessed table or view, so you will end up with 10 datasets.

To create a dataset, choose the data source for your Redshift cluster and select the `unified` schema and a view (e.g. `v_artists_from_iceland`). From the Analysis screen, select the fields to visualize and the visual type. Feel free to experiment with different visual types (e.g. bar charts, line charts, scatter plots, pivot tables, etc.) until you find one that is appropriate. Note that if you have a view that returns a scalar value or a set of values from a single column, you should adjust the query to return a set of values from multiple columns so that the output can be visualized in QuickSight.

You should now have 10 visuals, one corresponding to each database view. Take a screenshot of each visual and save it to a `quicksight` folder. Create the `quicksight` folder under your `final-project` folder in your local repo. Grant read-only access to your QuickSight account to the professor and both TAs by following the steps in the QuickSight Setup Guide [15].

Make note of any issues that your group encounters while working on this milestone in a m5-readme.txt file.

Commit the create_views.sql, quicksight folder with images of your visuals and dashboard, and the optional m5-readme.txt to your git repo and push the commits to GitHub. Create the standard **submission.json** file and upload it to the **Final Project Milestone 5** assignment in Canvas by the milestone deadline.

M6. Technical Report due Friday 04/28. Write a technical report that summarizes your group's experiences with this project. The report should describe your approach, including the advantages and drawbacks, as you experienced them while working on the various project milestones. It should cover the exploration of the datasets and challenges of modeling the data; the data loading approach, the quality checks and transforms applied to raw data and schema; the analysis queries developed over the unified schema and the refined queries and views for generating simple visualizations. It should also describe the technical challenges that your group encountered along the way and how you worked through those challenges. For example, describe the performance tables that you created to speed up a long-running view. The report should conclude with lessons learned, unexpected results, unsolved problems, or other issues that remain open. If you are not familiar with technical report writing, please consult these resources: technical writing guide [16] and two sample reports from previous semesters [17] [18].

The length of the report should be 10 pages, double-spaced. Feel free to include diagrams, code samples, and screenshots, but those will not count towards the page length. The report

should be formatted as a pdf file and named `final_report.pdf`. Make note of any issues that your group encounters during this milestone in a m6-readme.txt file.

Commit the final_report.pdf and the optional m6-readme.txt to your git repo and GitHub. Create the standard **submission.json** file and upload it into the **Final Project Milestone 6** assignment in Canvas by the milestone deadline.

M7. Presentation 05/01 – 05/04. Present a live demo of your project. Each group is given 10-minutes to present their work to the professor and TA. This is not intended to be an in-depth review of your project. You should pull out the key points from your solution and present only those, leaving 4-5 minutes for Q & A. Make sure to practice your presentation and be right at the 5-minute mark, as there will be a hard cut-off.

The presentations will be held from Monday 05/01 to Thursday 05/04 from 6:30pm to 8:30pm. Consult the presentation schedule [19] to find out your group's assigned timeslot. Also, note the location of your presentation since the room varies on a daily basis.

Teamwork & Collaboration:

Except for milestones 1 and 7, each group should use Issue Tracker to record a milestone's tasks, assignments, and status. We will be looking at GitHub regularly to check that both partners are contributing to the project and that frequent commits are occurring throughout the project. If your partner does not contribute to a milestone and you are stuck doing all the work, your partner will not receive credit for that milestone. If you faced issues with your partner when working on the previous assignments, please discuss openly and reach out for help if you need. For inspiration, you can read about Google's views on high-performance teams and how those teams resolve conflicts [6].

Final Grade:

Your grade on the project will be determined by how well you do in these two areas:

1. Project Milestones
2. Teamwork & Collaboration

References:

[1] Discogs: https://www.discogs.com/
[2] Million Song: https://labrosa.ee.columbia.edu/millionsong/
[3] Music Brainz: https://musicbrainz.org/
[4] Final Project Setup Guide: https://github.com/wolfier/CS327E/wiki/Setting-up-your-Final-Project
[5] JSONLint: http://jsonlint.com/
[6] What Google Learned From Its Quest to Build the Perfect Team: http://tinyurl.com/zev3s78
[7] Dataset subsets (download link): https://tinyurl.com/mb8qxby
[8] Music Brainz Schema: https://musicbrainz.org/doc/MusicBrainz_Database/Schema#Schema

[9] Music Brainz DDL: https://tinyurl.com/aoxn875
[10] Generate DDL script: https://tinyurl.com/ltsapm8
[11] Copy Command Parameters: http://docs.aws.amazon.com/redshift/latest/dg/r_COPY-alphabetical-parm-list.html
[12] Copy Load Error Reference: http://docs.aws.amazon.com/redshift/latest/dg/r_Load_Error_Reference.html
[13] get_utf8_bytes() UDF: https://tinyurl.com/n2vbk7h
[14] to_date() documentation: https://tinyurl.com/l9y2xdy
[15] QuickSight Setup Guide: https://github.com/wolfier/CS327E/wiki/Setting-up-QuickSight
[16] Valduriez et. al, Some Hints to Improve Writing of Technical Papers: http://www.cs.utexas.edu/~scohen/assignments/writing-hints.pdf
[17] Ong and Neotia, Final Project Report (Fall 2016): https://github.com/cs327e-spring2017/snippets/blob/master/sample-report1.pdf
[18] Kothari, Larsen, Nelson: Final Project Report (Spring 2016): https://github.com/cs327e-spring2017/snippets/blob/master/sample-report2.pdf
[19] Presentation Schedule: https://tinyurl.com/luo6xxe