Project 1B Specification

Computer Science 143 - Spring 2019

TuneSearch

Project 1 Part B

Due Friday, 04/26/2019 by 11:59pm

NOTE: If you have already downloaded the P1B assets ZIP file, use fresh copy from 1A to avoid issues: here.

Partners

Rules from 1A apply.

Overview

In this part of the project, you will use the data you loaded into PostgreSQL to create a simple search engine. In this part, you will accomplish the following:

1. Connect to the PostgreSQL database via the psycopg2 package for Python 3.
2. Use the parameters submitted by the user to form a SQL query that will fetch and rank results.
3. Gracefully pass the results to an HTML search results page.
4. Allow the user to page through the results in multiples of 20. (20 results shown at a time)
5. Protect the database against SQL injections, a common attack vector.

Project Layout

In the `www` directory in your home directory on the VM (your shared folder), you will see three directories: `SearchEngine`, `logs`, and `sql`. We will get to the logs directory later. The SearchEngine directory contains all of the files for this project, including files you will not need to touch that create a "glue" between nginx and Flask. This "glue" is uWSGI. There are a few different files in this directory, none which need to be touched:

1. `searchengine.ini`. While nginx handles the web connections over port 80 on the VM, nginx passes all requests to uWSGI. This .ini file configures `SearchEngine` for use with uWSGI. It mostly defines file permissions and the number of processes that uWSGI will run. There are a couple of parameters though that will be of interest to you.

`touch-reload` and `py-autoreload`. When a Python file in this project changes, uWSGI and Flask will reload them. Without these options, you would need to restart uwsgi or nginx every time you changed a Python file. `logto` specifies where the contents of stderr will be written on disk. This is where the logs directory comes in.

2. `wsgi.py` loads the `searchengine` module so that Flask can see it. It also sets up logging to stderr and appends the path of the module to the Python search path. It then runs the Flask app.

Inside the `SearchEngine` directory, there is another directory called `SearchEngine` which houses a Python module called `searchengine`. Got that? If you cd into that inner `SearchEngine` directory, you can tell it is a Python module because it contains a file called `__init__.py`. This directory contains all of the files that process the data for the search engine as well as the files that control the display of the website. There are several files and directories here:

1. `static` contains files that control the look and feel of the website, including the header image used for TuneSearch and a CSS file. *Feel free to play with these*.
2. `templates` contains an index.html file of plain HTML, and a results.html file which is a Jinja2 template. **You will need to modify this Jinja2 template to get pagination to work.**
3. `searchengine.py` has two functions you will need to modify: `dosearch()` and `index()` to handle the HTTP request sent by the user.
4. `search.py` which contains the main logic for connecting to the database, running the proper query and passing the results back to the Jinja2 template.

In the outermost `www` directory, there are two other directories: `logs` and `sql`.

1. `logs` contains one file, `errlog` which contains errors reported by Python to uWSGI.
2. `sql` is an empty directory, and you may wish to house your queries from Part 1A in this directory.

How TuneSearch Works

As stated in the project overview, a user issues a simple query containing words (tokens). These tokens may also have punctuation attached to them, such as when searching for specific lyrics "Could anything ever feel this real forever?" The punctuation must be stripped. I have already written some code to do this for you.

The user selects either the AND constraint, or the OR constraint. If the user selects AND, only songs that contain ALL of the words in the query are returned. If the user selects OR, songs containing ANY of the words in the query are returned.

When the user visits the main search page, the function `index()` in `searchengine.py` fires off and simply renders an HTML file called `index.html`. Whenever the user enters a search query and hits the search button, an HTTP GET request is passed to `/search`, which is handled by the `dosearch()` function in the same file. You can access data from the request using the `request.args` dict including the query and the query type (AND or OR). We then move to the `search()` function in `search.py` where we connect to a database (if needed) and run a Postgres query that finds songs that match the terms in the query. The results are then returned back to the calling function to be rendered in the `results.html` Jinja2 template.

This may sound like a lot, but we have broken down this project into a series of small tasks to keep your organized. Tasks 1, 4, and 5 will probably take the most time.

To view the website, you do not need to open the HTML file or run Python. Just open a browser to `http://localhost:1480`!

Task 0: Redo Part 1A with Our Solution

To prevent double jeopardy and have everyone start off on the same foot, please see our solution for project 1A which will be posted after the two day grace period. Please do not wait for that. If there are errors, you can always reload the data. We want all students to use the same schema.

Task 1: Search Ranking Query

Write a SQL query that computes the TF-IDF score for each token in each song. To do this, you will need to first compute the number of songs that each token appears in (the Inverse Document Frequency, or IDF). You already have the term frequency TF. There are several variations on how to compute TF-IDF. We will use this simple variation:

$$\text{tfidf}_{ij} = \text{TF}_{ij} \times \text{IDF}_i = \text{TF}_{ij} \times \log\left(\frac{|J|}{\text{DF}_i}\right)$$

where TF is the term (token) frequency of token *i* in song *j*, |J| is the number of songs and DF_i is the number of songs term *i* appears in. You will probably want to do this computation in a separate table, and as a one-time data processing step rather than as part of the app.

Then write a query against this table that handles both the AND case and the OR case. We can determine how well a song matches the query by summing across the TF-IDF scores in the

search query and then ordering in descending order by TF-IDF. You will want to write your query such that you can test it using some of your own sample queries at the `psql` command line.

Task 2: Write Code to Connect and Disconnect from the Database

In `search.py`, there is a comment with some instructions. You must connect to the Postgres instance running on your VM. Whenever you open a connection, you must also close it including when a failure occurs. Take a look at try/except/finally exception handling in Python.

Task 3: Integrate your Database Query in the `search()` Function

In the `search()` function in `search.py`, enter your PostgreSQL query or queries to handle the user's input and extract the results from the query. See the helpful hints later in this document.

Each result lists the name of the song (which links to the corresponding LyricsFreak page), as well as the artist that performed it.

Task 4: Enable Pagination

You now have a TF-IDF table, and a Postgres query that ranks the search results. Enough Python code is written for you already to output all results in one page. But now we are going to add a Previous and Next button at the bottom of the search results page, if there are more than 20 search results. Each page of results should contain 20 results.

To enable pagination, we can rerun the query you just wrote, once for each page of results, but this is very slow and wasteful. Instead, we should use some other construct we learned in lecture to store the search results, and then write another query against those results to pull the results for each page. Do you know what construct that is? How will you maintain state as you move among pages of results? You may want to play with the `request.args` dict as well as embedding some information in the search results page that will indicate where we are in the list of full results.

Finally, the Previous/Next buttons should only show up when they are necessary. For example, The Previous button should not show up on the first page of results, and the Next button should not appear on the final page of results. Additionally, Next button should not be displayed if there are no more results to display.

Task 5: Protect Your Database from SQL Injections

Finally, handle user input in such a way that the user cannot commit a SQL injection. Note that because some text processing is done on the query, you won't be able to actually do a SQL injection yourself. We just want your query to be protected from them.

Late Submission Policy

Part 1B must be submitted the deadline. See the syllabus for more information about the late policy.

Submission Instructions

Preparing Your Submission

Please create a folder named as your UID, put all your files into the folder, then compress this folder into a single zip file called `P1B.zip`. If you are working with a partner, **one** partner will use their UID as the name for the folder. We will figure out who the partner is from `TEAM.txt`.That is, the zip file should have the following structure.

```
P1B.zip
 |
 +- Folder named with Your UID, like "904200000" (without quotes)
     |
     +- SearchEngine (the outermost directory, just like in the ZIP file)
     |
     +- sql
     |
     +- TEAM.txt
     |
     +- README.txt
(We do not need your logs directory)
```

Please note that the file names are case sensitive, so you should use the exact same cases for the file names. (For teamwork, use the submitter's UID to name the folder)

- The `sql` directory should contain the following:
    - `load.sql` containing the query you used to populate the TF-IDF table.
    - **Please make the paths absolute so we can check them automatically.**
- `TEAM.txt`: A **plain text** file that contains the UID(s) of every member of your team. If you worked by yourself, just include your UID. If you worked with a partner, write both UIDs on one line separated by a comma (,).
- `README.txt`: A **plain text** file that contains anything thing else you feel may be useful to us. If you had any problems getting something to work they way we want it to, include it in this

file. While you may receive a point deduction, it is better than 0. This is also the file to include any citations.

Projects are submitted to CCLE, and only to CCLE.

Testing of Your Submission

Grading is a difficult and time-consuming process. In order to help you ensure you have the correct files for submission, you can test your packaging by downloading this Test Script.

In essence, this script unzips your submission to a temporary directory and tests whether or not you have included all your submission files. Once you download the test script, it can be executed like:

```
~$ ./p1b_test <Your UID>
```

(Put your P1B.zip file in the same directory with this test script, you may need to use "chmod +x p1b_test" if there is a permission error).

You **MUST** test your submission using the script before your final submission to minimize the chance of an unexpected error during grading. Significant points may be deducted if the grader encounters an error during grading. When everything runs properly, you will see an output similar to the following from this script:

```
Check File Successfully. Please upload your P1B.zip file to CCLE.
```

Helpful Hints and Troubleshooting

When you make a Python or SQL error, nginx will throw an unhelpful HTTP 502 error. If this happens, your first course of action is to look at `logs/errlog` as this is where most errors from Python and Postgres will be recorded. If you catch exceptions, you need to be very careful or else you may never be able to see the error message that is causing you grief.

You can also look at the nginx error log at `/var/log/nginx/error_log` but that is generally more helpful for configuration problems.

The file `search.py` contains a small `main()` function that you can use to pass search queries to Postgres via your Python code. You can then embed `print` statements, use a debugger, or other methods to see what is being returned from Postgres. You can change the `main()` function as you see fit for your testing.

The goal of this project is to do something interesting with Postgres. We are not interested in testing a bunch of edge and corner cases on the parsing of the search query (except for preventing SQL injections), so don't go overboard. :-)

Resources

`psycopg2` Postgres Driver for Python 3

Jinja2 Documentation

Flask Documentation

Textbook, 7th Edition, **Chapter 9**
This chapter focuses on Java, but the concepts are highly relevant.