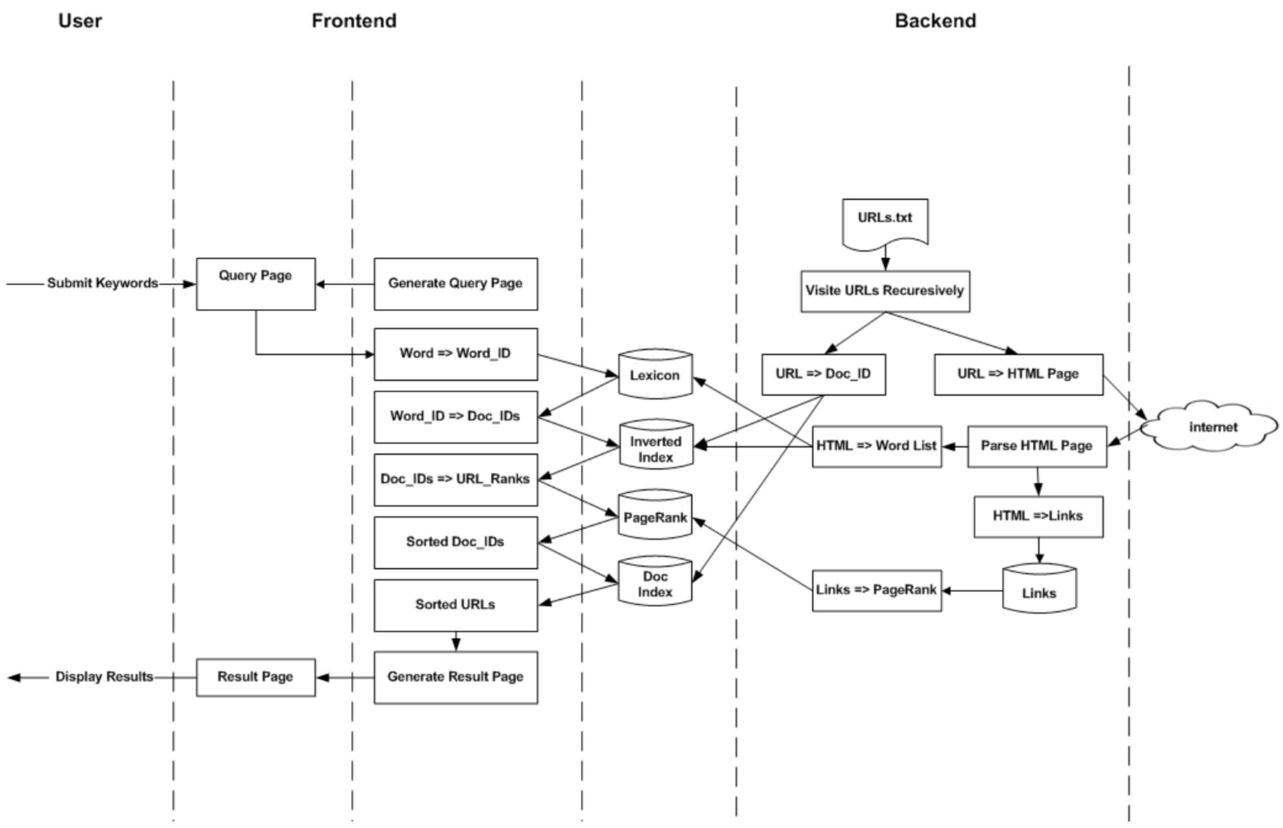
Lab 3 - Development Phase 3

Overview

In this lab, you will continue developing your frontend by integrating the data generated by the backend. For the backend, you will compute and store the PageRank scores for URLs in the document index. Store all the collected data, including the inverted index, lexicon, document index, and PageRank scores, in persistent storage.

By the end of Lab 3, the architecture of your search engine should resemble the one illustrated in Diagram . Note that the following architecture is not optimized and is only for reference. You are free to implement any architecture for your search engine as long as it preserves three main components: the frontend, the backend, and the persistent storage.



Frontend

F1. Search Engine Result Page

In response to the user query from the web browser, your frontend should search the keywords against the persistent storage generated by the backend. The retrieved list of URLs should be displayed on the browser in an order sorted by PageRank scores.

To simplify the search algorithm, your search engine is only required to search against the first word in the query string. For example, if a user searches "ECE326 lab3," your search engine only needs to look up "ECE326" from the databases.

On the result page, your frontend should also provide a query interface, i.e., the input form, for the user to submit a new query.

F2. Pagination

Your frontend should display a limited number of URLs per result page to avoid excessive processing time for the first result page. For example, if 1000 URLs are found for a keyword, only the first 5 URLs should be displayed on the first page. If more than 5 URLs are returned from the persistent storage, your frontend should provide a navigation bar for the user to switch between pages. Example: google.com.

F3. Error Page Handling

When a user tries to access a page or file that does not exist on your website, your frontend should return an error page indicating that the page or file is unavailable. The error page should also provide a link to the home page.

F4. Requirements

- When a keyword is submitted, your frontend should return a list of URLs or indicate that results for that keyword cannot be found.
- The returned list of URLs should be sorted by the PageRank score of each URL.
- Static pagination or dynamic page loading with AJAX should be used to limit the number of URLs returned by each server request.
- An error page should be returned when the user tries to access a page that does not exist or uses an unsupported HTTP method. The error page should provide a link to the valid query page.

Backend

B1. PageRank Algorithm

PageRank is a link analysis algorithm that ranks a list of documents based on the number of citations for each document. The PageRank algorithm gives each document a score, which represents the relative importance of the document. When a page has a high PageRank score, it may be pointed to by many other pages or by some pages with high PageRank scores. For details about Google's original PageRank algorithm, see the paper at here.

B2. Compute PageRank Scores

To compute the PageRank score, the links between pages, identified by anchor tags, must be discovered.

Your crawler should implement a method to capture the link relations between pages, and you need to design a data structure to store these links. Make sure your data structure is compatible with the interface for the PageRank function if you use the reference implementation.

Once the links are traversed and their relations are discovered, the PageRank function can be invoked to generate a score for each page or link, and the generated scores must be stored in persistent storage. A reference implementation of PageRank can be found at pagerank.pys.

B3. Persistent Storage

The data collected by the crawler and the PageRank scores must be stored in persistent storage so that the frontend can retrieve the data when needed. In this lab, you are free to choose any type of persistent storage, e.g., file, sqlite3, Redis, etc. If you do not have a preference for a specific persistent storage, you may use sqlite3. Below is an example for using sqlite3 in Python.

```
>>> import sqlite3 as lite
>>> con = lite.connect("dbFile.db")
>>> cur = con.cursor()
>>> cur.execute('CREATE TABLE IF NOT EXISTS dummyTable(id INTEGER PRIMARY KEY, value TEXT);')
>>> cur.execute('INSERT INTO dummyTable(value) VALUES("foo");')
>>> cur.execute('SELECT id FROM dummyTable WHERE value="foo"')
>>> id = cur.fetchone()
>>> print(id)
>>> con.commit()
>>> con.close()
```

B4. Requirements

- Compute the PageRank score for each page visited by the crawler, given a list of URLs specified in "urls.txt".
- Generate and store the required data, i.e., lexicon, document index, inverted index, and PageRank scores, in persistent storage.

Deployment on AWS

D1. Frontend

Similar to Lab 2, the frontend should be deployed on AWS.

D2. Backend Data on Persistent Storage

For this lab, the data generated by the crawler should be stored in persistent storage before deploying the frontend of the search engine on AWS. For example, if SQLite is being used for persistent storage, you may run the crawler on the EECG machine to generate an SQLite data file. To deploy your search engine, the frontend files and SQLite data file should be copied to your AWS instance, and “crawler.py" does not need to be uploaded to the AWS instance. When the frontend is executed, it reads database tables from the SQLite data file directly without invoking the backend crawler.

D3. Baseline Benchmarking

To evaluate the performance of your search engine, run a benchmark similar to what was done in Lab 2. Collect the benchmark results and compare them with the results from Lab 2. In one paragraph, briefly discuss the differences in the benchmark results between Lab 2 and Lab 3 and the causes of these differences.

Deliverables

- **Frontend:**
  - Source code of your frontend.
- **Backend:**
  - Source code of the crawler with your implementation of the PageRank algorithm.
  - Unit test cases for testing your implementation of the crawler and the data generated for persistent storage.
- **AWS Deployment:**
  - The search engine should be active online for 3 days after the due date of this lab.
  - The public DNS of the server should be included in a README file.
  - Data generated by the backend should already be stored in persistent storage at the time of frontend deployment on AWS.
  - Persistent storage should contain at least the data crawled from www.eecg.toronto.edu with a depth of one.
  - Benchmark setup and results in the README file.

Hints

- You may disable the Google login feature temporarily for Lab 3 and re-enable it in Lab 4 when needed.
- You may choose not to display the recently searched words from history and their counts on the search page. However, you may keep it to implement extra features for Lab 4, such as search recommendations based on search history.
- For pagination, you may display a list of pages or buttons for "prev-page" and "next-page."
- “TODO” statements in the reference crawler implementation should be considered hints, not requirements. Implement only the functions relevant to the features you need.
- The crawler implementation from Lab 1 assumes all data fits in memory, which may no longer be true in Lab 3. However, you may still use your Lab 1 implementation as a cache layer for persistent storage to improve performance. Note that disk performance is orders of magnitude slower than memory.
- When a page contains multiple links to a document, only the first link should be counted.

Submission

Compress all files, including source code and text files, and name the archive ‘lab3_group<group-number>.tar.gz‘ or ‘lab3_group<group-number>.zip‘ and submit it under Lab 3.