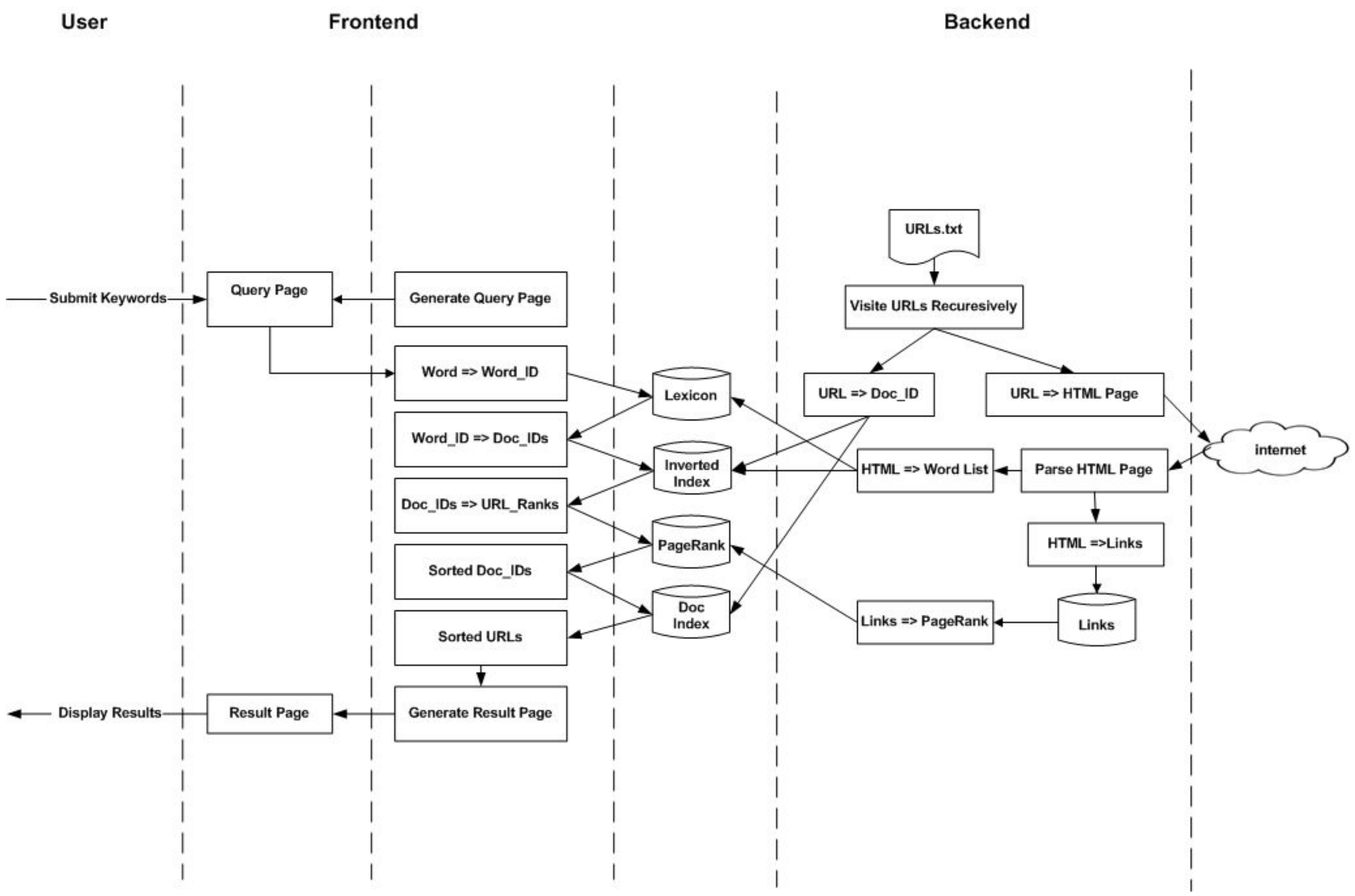
Lab 3 - Development Phase 2

In this lab, you will continue the development of 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. In this lab, you should store all the collected data, including the inverted index, lexicon, document index, PageRank scores to persistent storage.

By the end of Lab 3, the architecture of your search engine should be similar to those illustrated by the diagram below. 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. *To simplify the search algorithm, your search engine is only required to search against the* ***first*** *word in the query string*. For example, if an user searches "CSC326 lab3", your search engine is only required to look up "CSC326" from the databases.

The retrieved list of URLs should be displayed on the browser in *greatest-to-least order, sorted by PageRank scores.* The URLs listed should be clickable links, similar to Google and other search engines.

On the result page your frontend should provide the same query interface (i.e. search box) as the starting page, so that the user can submit a new query immediately from the results page.

F2. Pagination

Your frontend should display a limited number of URLs per result page to avoid excessive processing time to display the first result page. For example, if 1000 URLs are found for a keyword, the first 5 URLs should to be returned for the first page of the result page. In the case when more than 5 URLs are returned from the persistent storage, your frontend should provide a means of navigation from one page to the next (i.e. sidebar with list of page links, next/previous buttons). Example: google.com

Note that this kind of *static pagination* will require an additional GET query parameter (i.e. &page_no=3) for the page number. For example searches that have more than one page of results will be redirected by your server to page one (e.g. url/&keywords=airplane --> url/&keywords=airplane**&page_no=1**).

Alternatively, you may use Javascript to load the page result dynamically as user scroll down the result page. This approached is not recommended for those without prior knowledge of javascript, **and is not required**. Example: duckduckgo.com

F3. Error Page Handling

When a user is trying to access a web page or a file that does not exist on your website (i.e. 404 Error), your frontend should return an error page indicating that such page or file does not exist, and provide the user a link to the home page of your website. See "**Error Pages**" in the Bottle documentation for more details.

F4. Requirements

- When a keyword is submitted, your frontend should return a page with a list of clickable URL links, or a page indicating results for such 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 request sent to the server. Each page should have a maximum of **5**

links.
- An Error page should be returned when user is trying to access a page that does not existed, or using a HTTP method that is not supported. The error page should provide a link for user to visit the valid query page.

Backend

B1. PageRank Algorithm

PageRank is a link analysis algorithm that rank a list of documents based on the number of citations for each document. PageRank algorithm gives each document a score, which can be considered as the relative importance of the document. When a page has high PageRank score, it may be pointed by many other pages, or it may be pointed by some pages that have high PageRank scores.

For details about Google's original PageRank algorithm, see the paper at
http://infolab.stanford.edu/~backrub/google.html

B2. Compute PageRank Scores

To compute the PageRank score, the link between pages, identified by anchor tags, must be discovered. A reference implementation of the PageRank algorithm is provided (see below) ; it is up to you to adapt it to meet the requirements for the lab (i.e. persistence). It is also ok to implement your own PageRank function if you wish to do so, but no extra credit is assigned for this.

Your crawler from Lab 1 should be modified to implement a method to capture the link relations between pages. 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 its relations are 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 to persistent storage.

Reference implementation of PageRank can be found at pagerank.pys or
http://www.petergoodman.me/courses/2011/csc326/project/pagerank.pys

B3. Persistent Storage

The data collected by the crawler and the PageRank scores must to be stored in a persistent storage (i.e. in a file on disk), such that the frontend can retrieve this data later when needed.

In this lab, you are free to choose any type of persistent storage, e.g. sqlite3, redis, leveldb, MongoDB, Amazon RDB, etc. Note that persistent storage implies all transactions to the database are flushed to disk storage as soon as they are committed. Storing in-memory structures to files every once in awhile (i.e. via pickle) ***is not persistent storage***. Please ask TA's if you are unsure you are using an appropriate library/approach.

Below is an example for using sqlite3 in python, taken from the python sqlite documentation here.

```
>>> import sqlite3 as lite
>>> con = lite.connect(“dbFile.db”)
>>> cur = con.cursor()
>>> cur.execute("CREATE TABLE stocks (word_id integer, trans text, symbol text, qty
real, price real)")
>>> cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
>>> cur.execute("SELECT * FROM stocks WHERE symbol = '%s'" % "RHAT")
>>> printcur.fetchone()
(u'2006-01-05', u'BUY', u'RHAT', 100.0, 35.14)
>>> con.commit()
>>> con.close()
```

Revision

The Crawler/PageRank data structures are relatively challenging for a beginner to map to a relational database model (i.e. SQL, sqlite). For this reason we recommend using PyMongo (standard Python MongoDB interface) *if you are unsure of how to create a database schema or use SQL*.

B5. Requirement

- Compute the PageRank score for each page that is visited by the crawler, given a list of URLs specified in "*urls.txt*".
- There should be a single backend testing script (i.e. python **run_backend_test.py**) that runs the crawler and PageRank algorithm and pretty prints the PageRank scores in an easy-to-read format (see pprint module) in *greatest-to-least PageRank sorted order*.
- Your code should generate and store required data, i.e. lexicon, document index, inverted index, PageRank scores, in persistent storage.

B6. Bonus

- Implement a multithreaded crawler. The reference crawler provided in Lab 1 uses a single thread. Since the list of URLs is not required to be visited sequentially, performance of the crawler may be improved significantly if it is implemented with multi-threads.
- *Implementation of this part is not trivial, and you should do it only after the rest of this lab is completed and verified.*
- In case you have an incomplete version of this implementation, you should submit the crawler with required features from B5 in a separate to avoid any instability introduced by this extra feature.

Deployment on AWS

Similar to Lab 2, the frontend should be deployed on AWS (see **Deliverables** for details)..

D1. Backend Data on Persistent Storage

For this lab, the data generated by the crawler should be stored to persistent storage *before* deploying the frontend of the search engine on AWS.

Supposed you used sqlite3 for persistent storage. When the frontend is being executed, your frontend actually reads *precomputed* database tables from a SQLite data file to determine the search results. The crawler is run ahead of time, perhaps even on a different machine, as a batchjob to generate the SQLite data file. The crawler.py does not need to be uploaded to or run on the AWS instance.

D2. Baseline Benchmarking

To evaluate the performance of your search engine, you should run the frontend benchmark test from Lab 2 (i.e. use **ab** tool to send many concurrent requests).

Collect the benchmark results and compare it with the results from Lab 2. In one paragraph, discuss briefly for the difference of the benchmark results between Lab 2 and Lab 3, and the possible cause of the differences.

D3. Bonus

The AWS free tier provides a limited amount of free usage for relational databases, Amazon RDB, and NoSQL databases, Amazon DynamoDB. You can use AWS RDB and DynamoDB to build a more scalable search engine. However, *you are responsible for monitoring the actual usage of these services to avoid extra charges caused by your implementation*.

Using RDB or DynamoDB is not required; any database is fine. Bonus marks *may* be rewarded to those who implement extra features beyond the basic requirements. Note that bonus mark may not be given to implementation without correct basic features.

Deliverables

Frontend

- Source code of your frontend

Backend

- Source code of the crawler with your implementation of PageRank algorithm *and* persistent storage.
- A single backend testing script (i.e. python **run_backend_test.py**) that
    - Performs any needed setup
    - Runs the crawler and PageRank algorithm and
    - Pretty prints the PageRank scores in an easy-to-read format (see pprint module) in *greatest-to-least PageRank sorted order*.
- A urls.txt file populated with a suitable test case.

AWS Deployment

- The search engine should be deployed on AWS and reachable online for one week after the due date of this lab.
- Persistent storage used by your frontend on AWS should contain *at least* the data crawled from www.eecg.toronto.edu with depth of one.

Documentation

- README file containing
    - Brief documentation stating how to run your backend to demonstrate the required functionality.
    - How to access your frontend on AWS; Public DNS of your frontend on AWS should be included.
    - Description of your frontend benchmark setup and your results.

Bonus Work

- To claim bonus marks, you must **explicitly indicate** the bonus features that you have implemented and whether they are complete or incomplete in the README file.
- Include a simple test demonstrating performance increase of the bonus backend component, and specify the observed performance gain in the README file.

Hints

- You may disable Google login feature temporarily for Lab 3, and re-enable it in Lab 4 when it is needed.
- The recently searched words and history tables are **not** required for this lab. However, you may want to keep them in order to implement extra features for Lab 4, such as search recommendation based on search history.
- For pagination, you may display a list of page links (perhaps in a sidebar), or buttons for prev-page and next-page.
- *"**TODO**" statements in the reference crawler implementation should be considered as hints*, and not requirements. You need only implement functions that are relevant to the features you need.
- The crawler implementation from Lab 1 assumes that all data fits in memory, which is no longer true in Lab 3. However, you may still keep your Lab 1 implementation as a cache layer for the persistent storage to speed up performance. In other words, the in-memory Lab1 implementation can be used as a subroutine to compute *partial* results, with the full results gathered and stored persistently on disk.
    - Note that disk performance is orders of magnitude slower than memory, so using database to store all data all of the time will result in a very slow program.
- When a page contains multiple links to a document, only the first link should be counted.

Submission

Compress and archive all your files, including the source code and the README file, and name it lab3_group_<group_number>.tar.gz with tar.

For example, for group 4, use the following command. **Please check that your compressed archive is non-empty and is not corrupt before submission.**

```
$ tar -zcvf lab3_group_4.tar.gz <files>
```

To submit your package, use the following command on EECG machine:

```
$ submitcsc326f-lab 3 lab3_group_<group_number>.tar.gz
```

Only one member of the group is required submit the package.