COMP 4321 Project Report Group 6

SUHARTONO, Kristian | 20313207 | ksuhartono

CHENG Ho Kei | 12219689 | hkchengad

1. **Overall Design of the System**

Our system is fully built with Golang, an open source programming language created by Google. We chose this language because we liked its concurrency property and the design of our system has been capitalizing on that as the main advantage.

Our system consists of two parts, the Crawler which does the crawling of web pages and then indexes it into the database, and the Search Engine interface. We implement them as two different programs.

* Crawler Indexer:

The crawler makes use of the concurrent property that is available in Go. First, from the starting URL, we start a crawl into the site. Then for all the pages that are found as a child of the page, we implement a breadth-first search using what is called “channels” and we kick off crawling each of the child URLs concurrently. So that the crawling process take less time.

While crawling, we also feed the results to the indexer.

When finished, we close all the databases that are open and exit this program.

* Search Engine Interface:

With a server built with native support from the Go Programming language, we hosted a ‘router’ based system in order to express the directory of the search engine page. Using packages from the official Go library, we implemented an HTTP server to host our sites. We are locally hosting it in “<http://localhost:8080/>” and if the exe file to host the server has been run, accessing the URL will direct you to the landing page.

Each sites were implemented using HTML and CSS. They are mainly layouted using Bootstrap CSS stylesheet to make layouting and designing the page simpler instead of using raw HTML and CSS. The HTTP server handles receiving queries from the page, processing it, and then performing the retrieval.

During retrieval, the backend engine would first analyze the query, splitting it into terms. If the user wished to perform phrase search by using double quotes in the query, it will also be taken care of in this stage. The vectorized terms and phrases are then passed to the retrieval function which would access the database, computing tf-idf and the cosine similarity ,and return the ranked result at last. After that, it redirects the user to a another page containing the retrieval results.

1. **File Structures in Index Database**

All the databases in the project is stored with a Go library called bolt. It is a low-level key/value pair storage library (which is similar to JDBM). It does not provide serialization functions, so all the integer id, text string and relevant information are serialized into a sequence of bytes by hand-written functions. It uses only B+ tree internally, therefore all of our data structures are built with B+ tree.

Using all B+ tree in our projects make our data structure extremely scalable. As the web search engine need to surf the Internet which is growing at a high speed, storage options with speed worst than linear time is not acceptable. With B+ tree, the search and insertion time of our data structures can be logarithmic with a large log base. Besides, B+ tree is better than hash table (despite that hash table is faster) in the sense that we don’t have to estimate the data size and the hash function in advance.

With B+ tree data structure and 64 bit id, our project, from the database point of view, is highly scalable. It is able to handle the massive amount of pages and data in the Internet.

We mainly uses four databases in our project, two of them are mapping tables between id and string, and the rest is a document information metadata storage table and a database that contains all the inverted files respectively.

* Mapping Tables: Url <-> Url ID and Term <-> Term ID

The two mapping databases in our project are the engines of the entire data structure as all other databases are based on them. Both of them convert between a string (URL and the term) and a 64-bit integer (Url id and term id).

Inside each mapping database, there are two B+ trees which stores essentially the same information, but with different structure. The forward mapping tree uses the 64 bit id as the key, and the string as the value, while the inverted mapping tree uses the serialized byte string as the key, and the 64 bit id as the value.

Using two trees doubles the storage requirement, but it minimizes the time required to retrieve id from string or vice versa, as searching B+ tree is much more efficient to search with keys.

* Document Metadata Database

The document metadata database is the smallest and simplest database in the project. It only has one B+ tree, using the document ID as the key and the metadata struct as the value. The metadata struct includes the size of the document, last modified time, parent url ids, children url ids, and its own title. This struct has to be encoded and decoded by tailor-made functions before and after database accesses.

* Posting list

This is the most complicated and largest database in our system. It has five B+ trees inside.

* Title B+ tree

The title B+ tree uses the document id as the key, and the title as the value. It is not stemmed nor stopword removed, as we think the terms in the title carry high values and should not be changed. To entertain stemmed searches, the stemmed version of the title would also be inserted into the stemmed posting list (see below). The values of the terms in the title are also increased during indexing.

* Inverted index (stemmed and unstemmed)

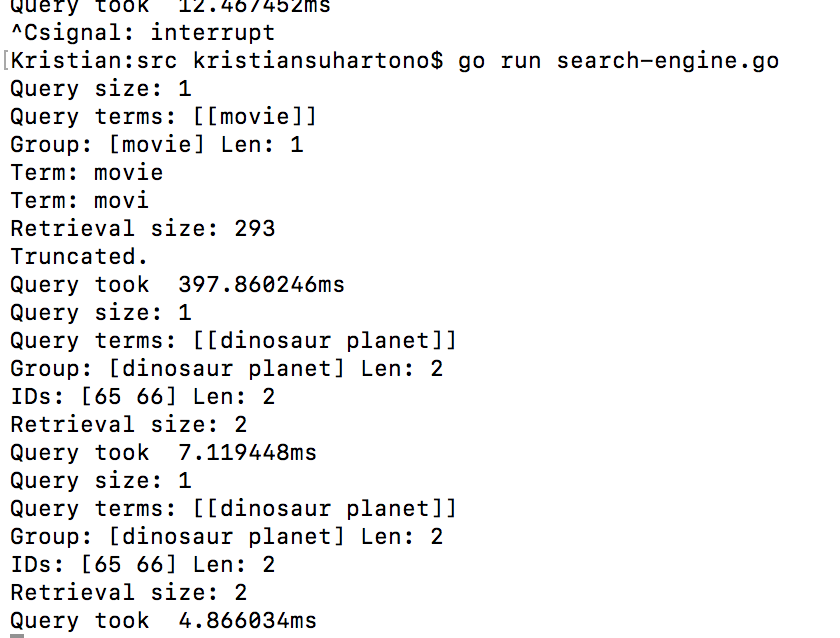
We have two version of inverted index with similar structure. One of them contained stemmed and stopwords removed terms, while the other contains the original terms. This is an extra function that we have implemented (explained in part 3), as we do not want to lose the information in the document (caused by stemming). Both of them are nested B+ tree, first with a term id as the key, and a B+ tree as the value. Inside the child B+ tree, the document id is used as the key, and the posting is the value. Both term frequency and term position (to support phrase search) are stored in the unstemmed file, while only the term frequency is stored in the stemmed file, as higher precision is expected from phrase search and we would only use the unstemmed file to perform phrase search.

The document frequency (df) is stored at a special position in the B+ tree, being the only value that is not another B+ tree in the outer tree. It has a key value of zero.

* Forward index (stemmed and unstemmed)

The forward index stores similar information as the inverted index, but in different structure. Therefore, it would also have two versions: stemmed and unstemmed. Similarity, both of them have a nested structure. The order is different in the forward index. The outer B+ tree uses the document ID as the key, and an inner tree as the value. In the inner tree, the term id is used as the key, and the term frequency in the term is used as the value.

Again, the value at key zero has a special meaning. It does not contain an inner tree, but the maximum term frequency in the document. This value is used in tf-idf computation.

1. **Features Beyond Specification**

Our retrieval speed is rather fast, as the moment the user inputs the query, to the user the results page would load instantly. Based on system timers, the time of queries (except the first time, due to the caching. This is not a problem as the database would normally be kept running in the server non-stop) are mostly under 20ms. This is due to the use concurrent property of Go. During each retrieval process, a number of goroutine will be initiated to access the database and compute the results simultaneously. This allows us to make full use of nowadays common multi-core processors. Besides, as disk accesses are slow, other goroutine can spend the CPU time to do computation while one wait for the disk to respond. Traditional single-thread approach will waste most of the time to wait for the disk to respond.

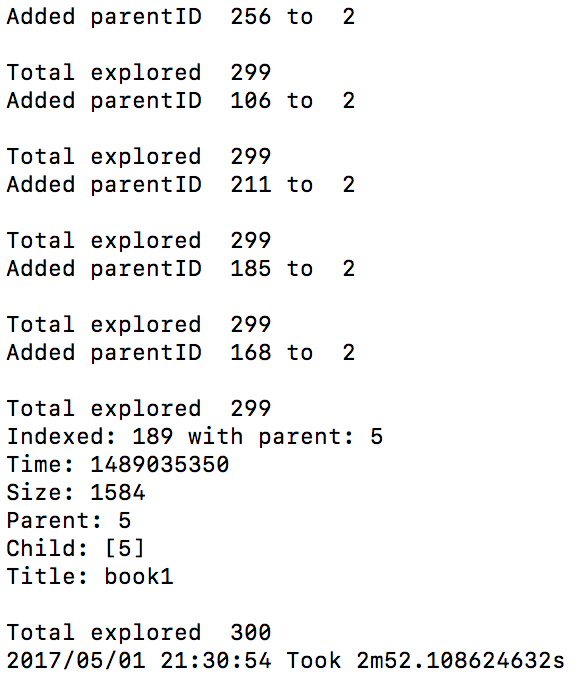
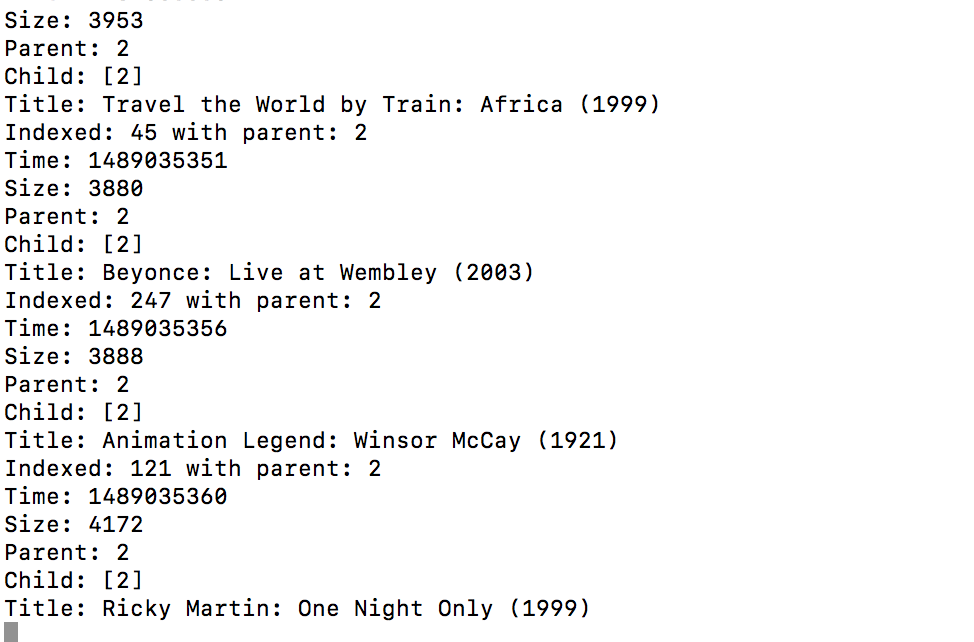
Also based on the results from the html page, during the indexing and computation of weights for the keywords, we added a bias on the importance of the terms for special elements in the page such as bolded text and titles in the page. We do this by extracting these elements from the html and adding extra weight to these terms importance.

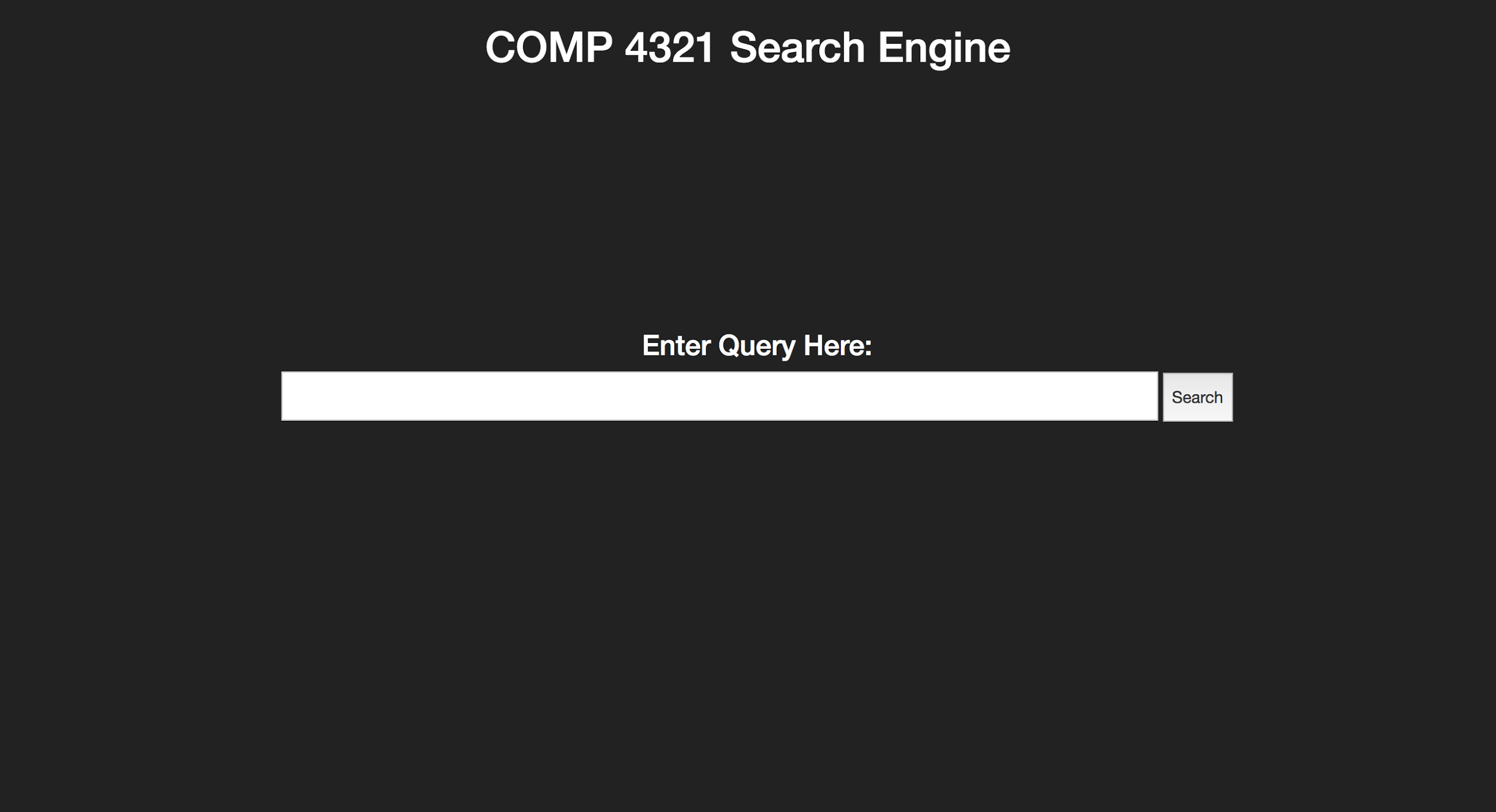
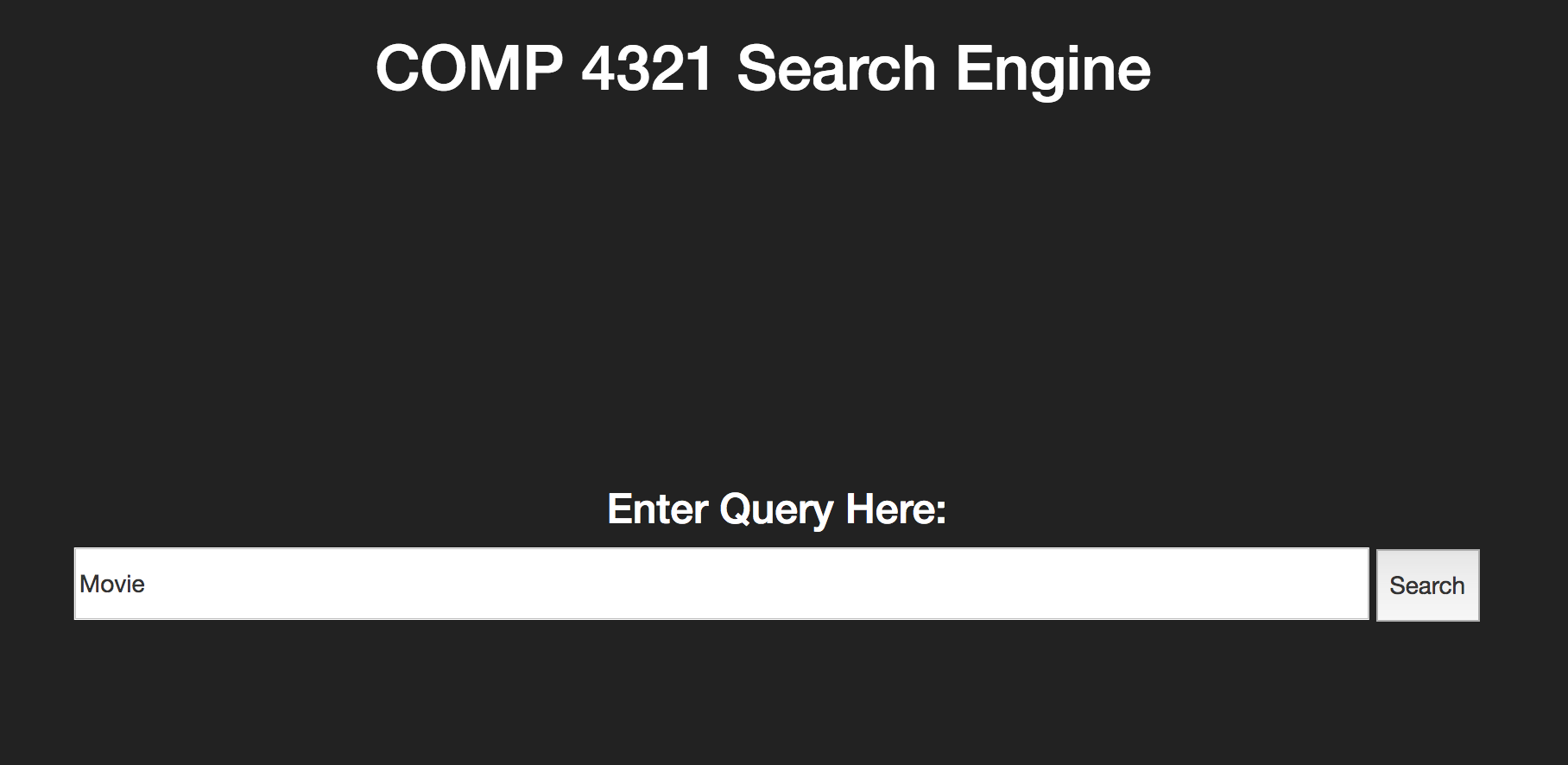
As mentioned in part 2, we have kept both the stemmed and the unstemmed version in our databases. Although this increases the index time and the storage requirement, it helps us to preserve much more information in the document, while making use of the increased recall by stemming. When a search is initiated, we will search in both the stemmed and the unstemmed version of the posting lists, compute their cosine similarity separately, and add the results up at last. This pretty much means the document with the original term will get a higher score, due to a lower max tf in the stemmed version and the combined sum from both searches. Yet, it gives the documents with the stemmed version of the words some scores, so as to increase recall. Such two-channel retrieval would normally slow down the search by a lot, but thanks to concurrency, our retrieval speed is still remarkably fast.

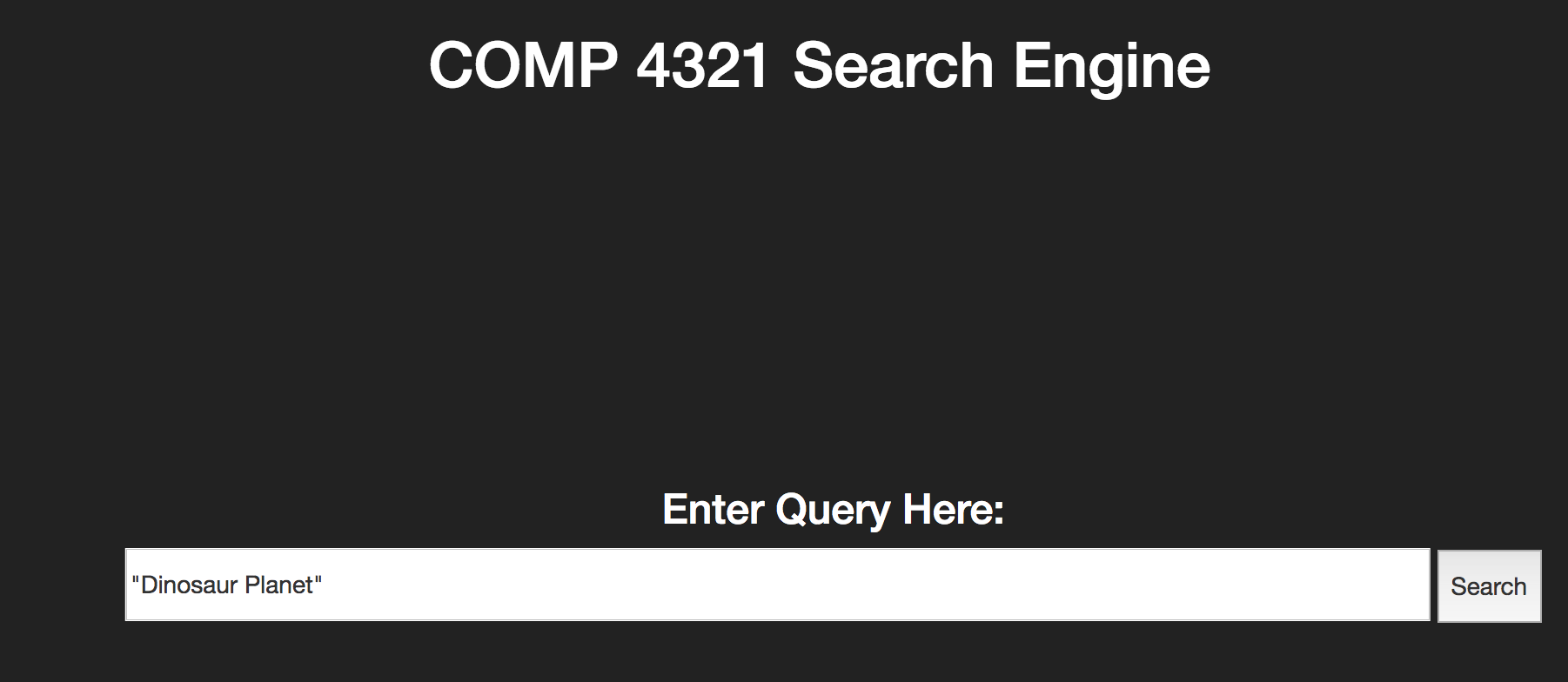
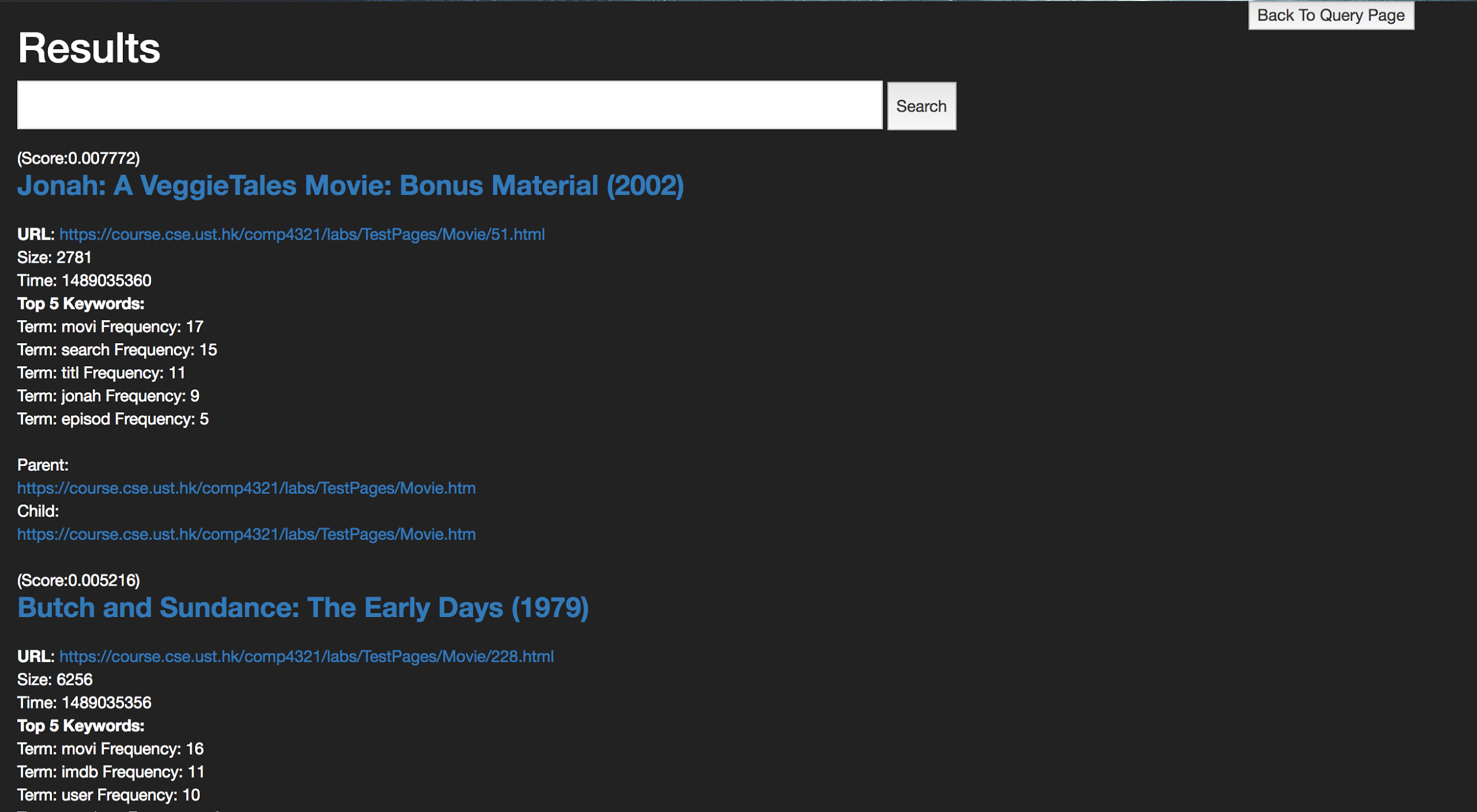
An exception is phrase searches. We assumed that the user demanded a higher precision when they use phrase search, therefore we would only search in the unstemmed posting list.

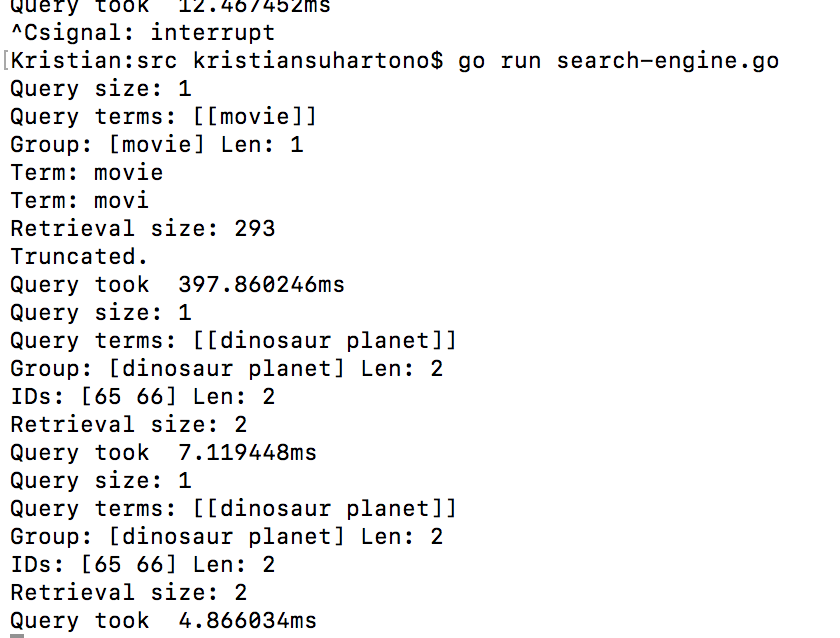
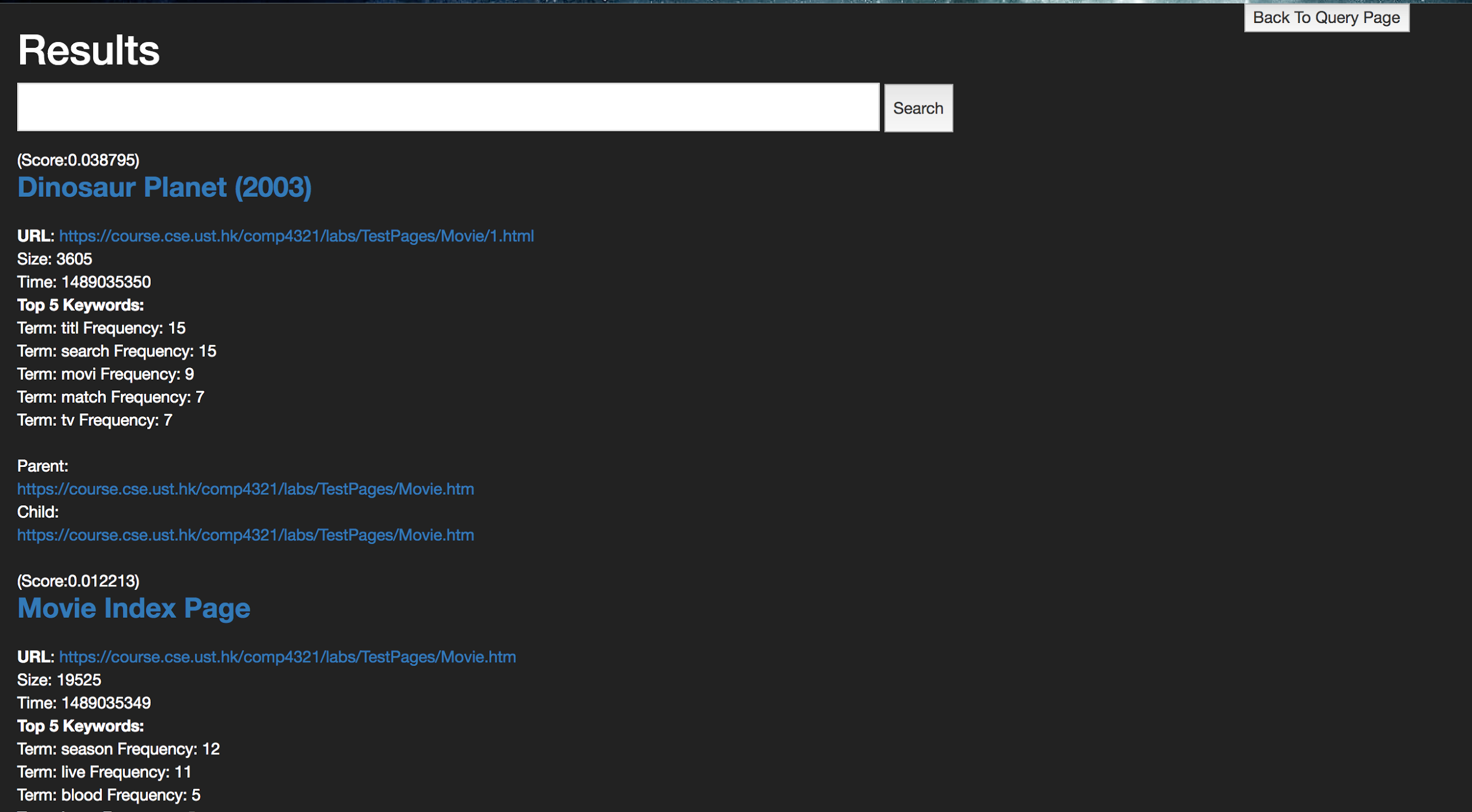
1. **Testing of Functions**

Crawling + Indexing Process: tested by running “go run crawl.go”. Here we tested the main crawling function that fetches pages and gives the contents to the indexer. The indexer then proceeds to index the pages, and also does the processing of the contents of the page (stemming, stopword removal)

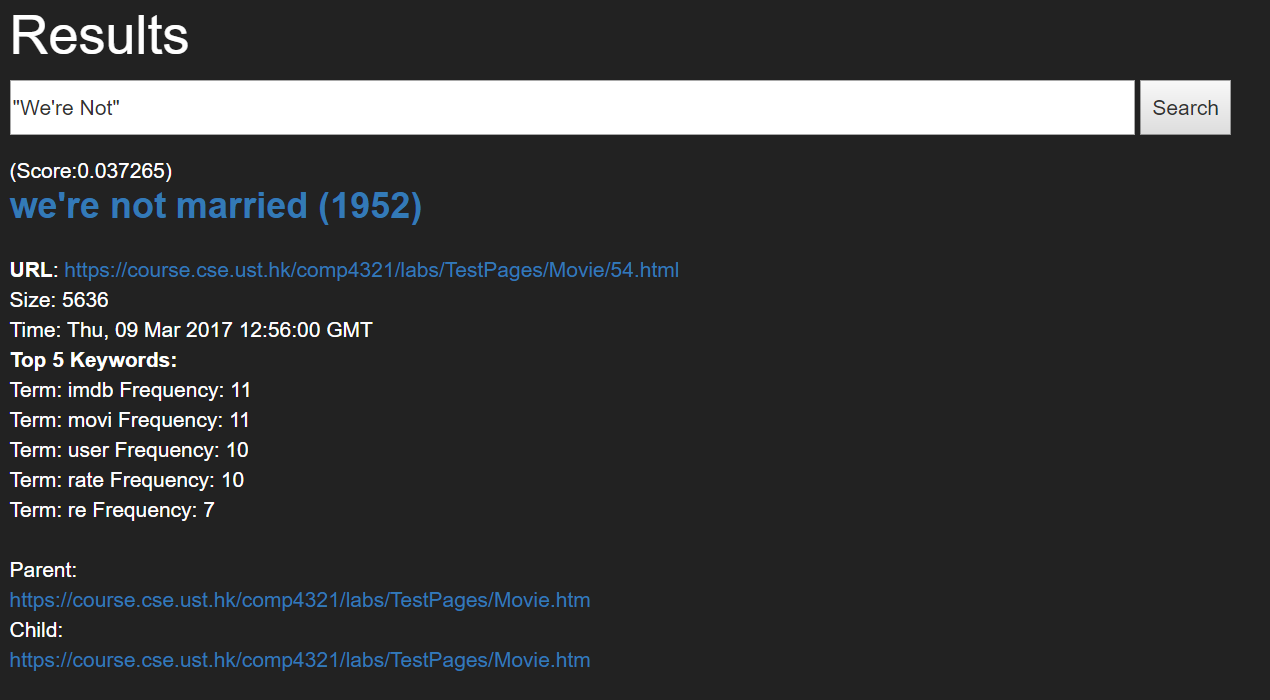


Search Engine : tested by running “go run search-engine.go”. Here, the http server will load up the page and host it. We can then access the landing page of the search engine. 

Above: Landing page and submitting a query;Below: results page, phrase search

Results of phrase search and terminal output

As can be seen, the output in the results follow the format requirements that are displayed in the project description. Also, we attempted to make it as user friendly as possible by allowing users to submit more queries even from the results page. Even though we still have a back to query page button if the user wants to go back to the main page.



A phrase search of “We’re Not” successfully return one and only one result that is relevant. This is because we have kept the unstemmed and non-stopword removed version. “We’re Not” would otherwise be converted to an empty string by stopword removal.

1. **Conclusion**

We believe that the main strength in our systems lies in its utilization of the concurrency that is provided by the Go programming language. Using this we were able to do things at a faster rate when compared to doing it line by line for each function.

The main weakness of our weakness is in the speed of our indexing process. The crawling process is able to run fast, however the indexing process cannot run as fast. Due to it having to keep up and do stemming and stopword removal on all the keywords found it takes longer. The fact that we are keeping two versions of the posting and forward list almost doubled the indexing time. Not only that, our database structure only allows a single writer writing into the database at a time. Therefore making us unable to utilize the concurrency property to speed up the process. We did try to use it however it resulted in data loss due to some data not being actually written to the database as something else is writing into the database at the moment.

If we could reimplement the whole project, we would try to counter the one writer issue by implementing the database differently, like with more files and thus less locks in the file system. So that we are able to utilize the concurrent property better such that we can crawl and index faster.

Interesting features to add would be the implementation of extra bias to pages with links and pages that are pointed to by links that are also relevant to the query (PageRank) so that we can have a better ranking system for our query results. Also, implementation search suggestions as can be seen in Google would be interesting, so that upon typing the query, our search engine would attempt to complete the word and give some suggestions. This is possible in our system due to the fast retrieval speed. At the moment that the user have typed the query before submitting, we can already access the database and return suggestions (like terms that would return a lot of results) or spelling corrections (if the typed terms return no or very little results) at a speedy manner. It would not be practical if the retrieval needs more than half a second to complete.

**6. Contributions**

Contributions:

CHENG Ho Kei: Database design, indexer, retrieval, 50%

SUHARTONO, Kristian: Program structure and database design, crawler, web interface, 50%