HET Search Engine Design

In this project, the entire search engine compiles into a single program, which provide all the functionality. Including Indexing and HTTP server to display the UI.

The entire backend of the search engine is written in the Go Language. Its an excellent language released by Google for low level networking servers.

Data storage

The library we use for the database is called BoltDB. Its a key/value store that uses B-Trees internally. It is completely written in GoLang and gives strong transaction support. We utilize the transaction support to the fullest. Transaction support enables us to restart the search engine even while its indexing. IT will automatically recover, and begin exactly where it left without corrupting our schema or loosing any bit of data. BoltDB has a concept of a table called Bucket. Each bucket can have many Key/Values stored. In our search engine, we utilize the following buckets

Bucket Name	Description
pending	Contains all the pages that are pending to be crawled. Crawler will take a page from pending, index it and then delete from the pending table.
docs	Docs contain all the documents that we have indexed. It is hashed by the URL of the doc.
doc-keywords	Contains the keywords stored in each document. The reason this data is not stored in the docs bucket is for performance reasons, and to opmise the search result latency.
links	Contains all the links that have been indexed. We add an item every time an http link is found. It even stores the link if the link was to an image, or other resource which we dont index. It also keeps track of the outcomming links to a specific link and the incomming links. There is a partial implementation of PageRank which uses this bucket.
keywords	Contains all the keywords that have been found and indexed. The heart of the search engine. All entries are hashed by the keyword itself.
stats	Contains statistics about the entire search engine. How many pages still pending to be indexed. How many documents, links, keywords indexed.

BoltDB has no restriction on the schema of each bucket, it just accepts bytes and its up to the application to serialize their data structures to bytes. We internally use JSON to encode the golang structures. Schema for each bucket is described below.

Database Schema

Most of the schema lives in the het package. Which contain common types that all search engine packages can use. The database layer, also uses these types to store database in the filesystem. Most of the types live in het/types.go

Docs Bucket (het.Document type)

Property Name	Object Type	Descriptipn
URL	URL	URL parsed in golang URL type for the document. It is the final URL, not any redirects that point to the document
Title	string	The Title of the docuement. Extracted from the HTML title tag
Length	int	The Length of the document vector. Each entry is a unique word in the document. Computed when the crawler is parsing the doc html

Doc-keywords Bucket (het.DocKeyworde type)

This bucket is an array of the internal Keyword-Ref Type. Every doc has many Keyword-Ref type stored in the doc-keywords bucket. The key is the URL of the doc, and the value is a list of Keyword-Ref's. Following is the schema of Keyword-Ref

Property Name	Object Type	Descriptipn
Word	string	The keyword itself.
Frequency	int	The amount of times it appears in the document

Keywords Bucket (het.Keyword type)

Property Name	Object Type	Descriptipn
Frequency	int	The amount of times it appears accross all documents. Total times this keyword has been seen while indexing
Docs	array of { URL: URL, Frequency: Int}	Array of Documents that

Links Bucket (het.Link type)

Property Name	Object Type	Descriptipn
Redirect	bool	True if this link was a redirect to another link. If yes then the URL contains the redirected link and everything else is empty
URL	URL	URL of the link. If its a redirect then the final URL, otherwise the original URL.
LastModified	string	Last Modified date of the document. If not given in HTTP headers then it assumes the day it was crawled
ContentType	string	HTTP content type of a speccific docuemnt. It will always contian the HTML in case its a valid document which can be indexed. For the rest content types the links are stored but never indexed.
StatusCode	int	The HTTP status code when the link is retrieved. If the link was not found then the status code is 404 and it is still stored. This is a big optimisation to store all links to speed up the crawling
Rank	float64	The link rank. Currently it is filled up with the Page Rank of the link.
Outgoing	map[string]bool	All the links that are outgoing from the link. It is a map so that it is ensured that no duplicates are added, and checking if a link is part of outgoing links is fast. Maps are like hash tables in GoLang. String contains the URL of the outgoing link
Incomming	map[string]bool	Similar to Outgoing. Contains the incomming links from other docs.

Note: There is an initial page rank implementation that fills the rank of each link and keeps updating it until they converge. But it was buggy, and still not

shown in the UI of the search engine. We never got time to implement it, but it would be trivial to do so if we had more time with this architecture.

Stats Bucket (het.CountStats type)

Property Name	Object Type	Description
DocumentCount	Int	Contains the total docuemnts hat have been indexed
PendingCount	int	Contains the total numer of docs stored in the pending. Indexer needs to take them out and crawl them one by one
KeywordCount	int	Total number of keywords seen as of yet by the indexer.
LinkCount	int	Total links that are currently stored in the database

Main Controller

This is the first component that runs when the search engine start. It contains the Main function that runs when the search engine executable is ran. The code lives in the engine.go file. It handles the indexer, API Server and also the static HTTP server components.

It takes the following arguments from the commandline. Most of which have a default so they dont have to be explicitly stated unless a change is required.

Flag Name	Description	Default Value
db	Path to the local db used for indexing and searching.	"./index.db"
stopwords	Path to the stop words used to filter out common words	"./stopwords.txt"
index	Maximum pages to index by the search engine. If less pages are indexed then the indexer will start and index pages before starting the web server	300
drop	Set to reset DB and start indexing from scratch	FALSE

The main controller then tries to open the database with path given in the command line or the default one. If it doesn't find a database file then it creates one. If the drop command line flag is given then it resets the database to be empty.
If it creates a new database, then it puts (http://comp4321.cse.ust.hk/~your_account/comp4321proj.html) link as the first entry in the pending bucket.

After the database has been initialized, the main controller checks if enough docs have been indexed. By default 300 are enough, but this can be overwidden through the index command line argument.

If enough documents are not indexed, then it calls the CrawlPage function in the indexer package in a loop. CrawlPage will try to crawl a single page, and return if it was indexed successfully. The main controller calls CrawlPage until enough pages have been indexed.

Once enough pages have been indexed, the main controller starts an http server. It starts the API server for clients to get search results. It then also starts a static server to serve our UI (HTML/CSS/Javascript) to the client. The UI will display the final search enginge web app, and talk to the API server to search for certain queries.

Crawl Component

This is the main essense of the entire project. This component indexes a single page from the pending bucket and fills up the Docs, Keywords, Links and Doc-Keywords bucket.

It first tries to lookup the page in the Links bucket, if not found then it retrieves it and stores it in the Links Bucket. If it was the redirect then it follows the redirect for atleast 10 tries before giving up.

After link, it tries to see if this page has already been indexed, if it is then return early and index another page.

If no duplicate is found then it uses the internal Golang HTML parser to parse the html, and retrieves all of the title and anchor tags. Title is stored

in the Docks bucket, and the anchors are retrieved one by one from the links bucket.

If the content type of an anchor is HTML, then it is strored back in the pending bucket to be indexed later.

The Crawler also tokenizes the doc HTML TextNodes to collect displayable text. The text is broken by spaces (newlines, tabs ..) and also by non alpha characters. They are fully trimmied, and then stemmed to be strored in the Keywords Bucket. If the keyword is already seen then their frequencies are updated. The currenty doc is added into the list of docs in the keyword.

This entire process happens in a transaction. So if any error happends, or the search engine is killed, then no data is lost. When it resumes, it starts from where the transaction ends and completes the crawl process.

API Server

The API server runs on the local port 8080 and on path "/search". It takes one GET query parameter, query. It then returns JSON array for the search results.

So, to get search results for the query anaconda, you run:
http://localhost:8080/search?query=anaconda

The response returned is a JSON object. It contains a success field.

If success is false then it means that something went wrong and the error message is then stored in the message field.
If success is true then a results field is attached which contains the results for each document which matches the query. This is obtained by calling the Search Module which would be described later.

The schema of the JSON element in the result array is described below:

Property Name	Object Type	Description
Doc	het.Document	Contains the Title,URL, Size and Length of the docuement. Same type as the one stored in the Docs Bucket

Property Name	Object Type	Description
Link	het.Link	Contains the Link information for a specific Document. Contains LastModified, ContentType, StatusCode and also Incomming/Outgoing links from this document.
URL	string	The final URL to display to the user. Its the final URL and never a redirecting URL.
Keywords	het.DocKeywords	Contains the individual top 5 keywrods stored in a specific docuemnt. Contains the word along with their frequencies.

Example response can look like this:


```
{
   "success": true,
   "results": [
      {
         "Doc": {
            "URL": {
               "Host":"www.cse.ust.hk",
               "Path":"/~ericzhao/COMP4321/TestPages/Movie/131.html",
             },
            "Title":"Arachnid (2002)",
            "Length":33.95585369269929,
            "Size":7156
          },
          "Rank":0.1931806897525771,
          "URL":"...",
          "Keywords":[
              {
                 "Word":"movi","Frequency":11
              },
              {...}, {..}
           ]
       }
   ]
}
```

Search Component

Search component is responsible for using the database (index.db) to lookup the index value of all the documents analyzed by spider. It receives the query then tokenizes and stems the query.

Then it looks up the Keywords Bucket to get the docs which refer to the keyword. For each document, it creates a tf-idf table to store the relative weight of each document for each keyword. If a document does not contain a keyword at all then it is pushed to the bottom dilebrately dispite the final weight as an optimisation.

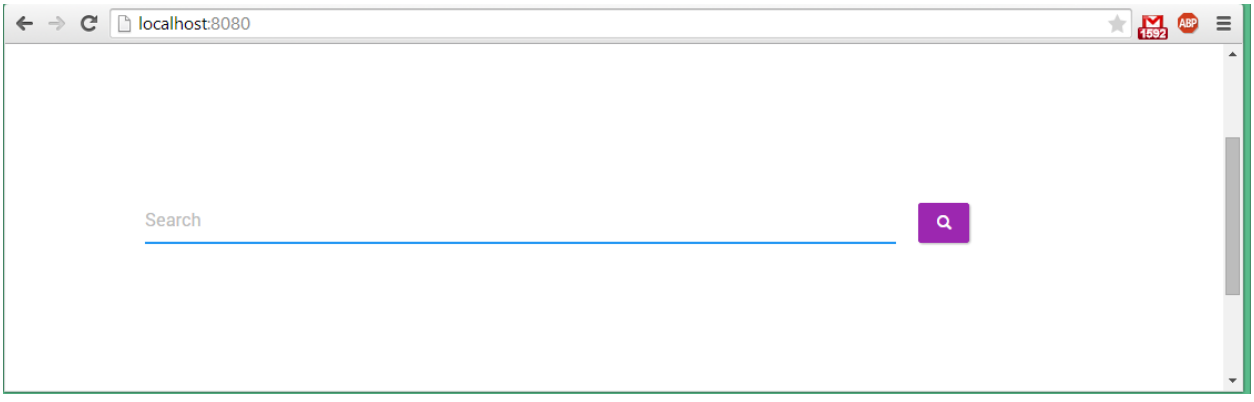
After calculating the tf-idf weights, it then measures the cosine similarity between the query vector and each docuement vector. This is the final rank of the document. Once ranks have been finalized, then the top 50 results are chosen, and extra metadata is retrived from the buckets. Which include the incomming, outgoing links, http headers such as last modified, content type and so on.

UI Component.

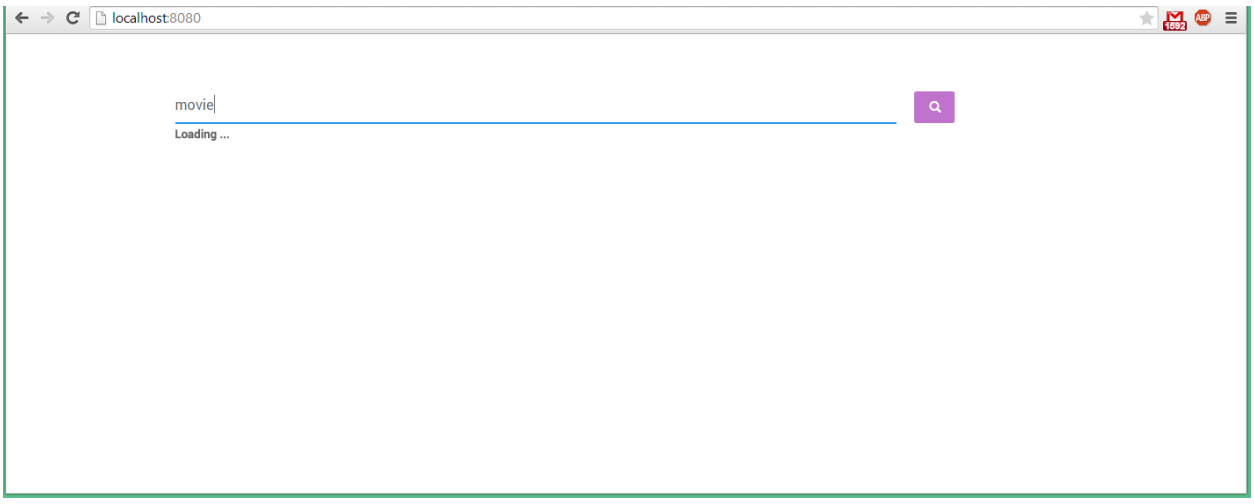
After calculating the tf-idf weights, it then measures the cosine similarity between the query vector and each docuement vector. This is the final rank of the document. Once ranks have been finalized, then the top 50 results are chosen, and extra metadata is retrived from the buckets. Which include the incomming, outgoing links, http headers such as last modified, content type and so on.

Following are some of the screen shots of the amaizng UI that we achieved:

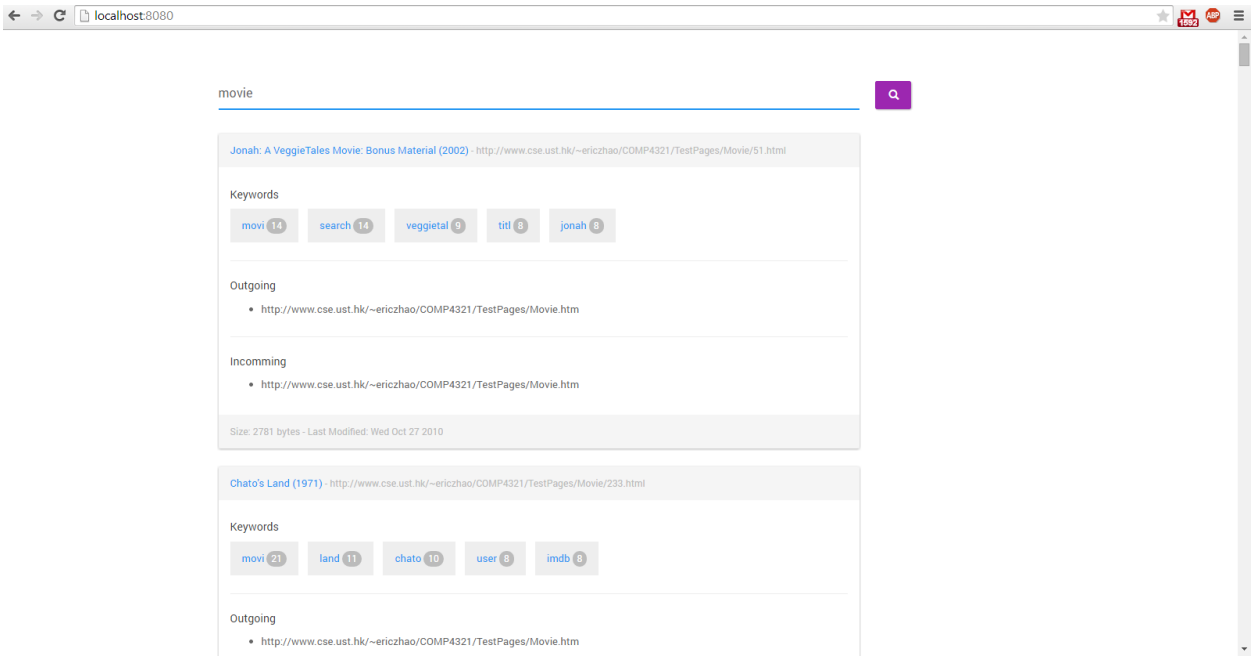
Initially, a simple UI is greeted to the user.



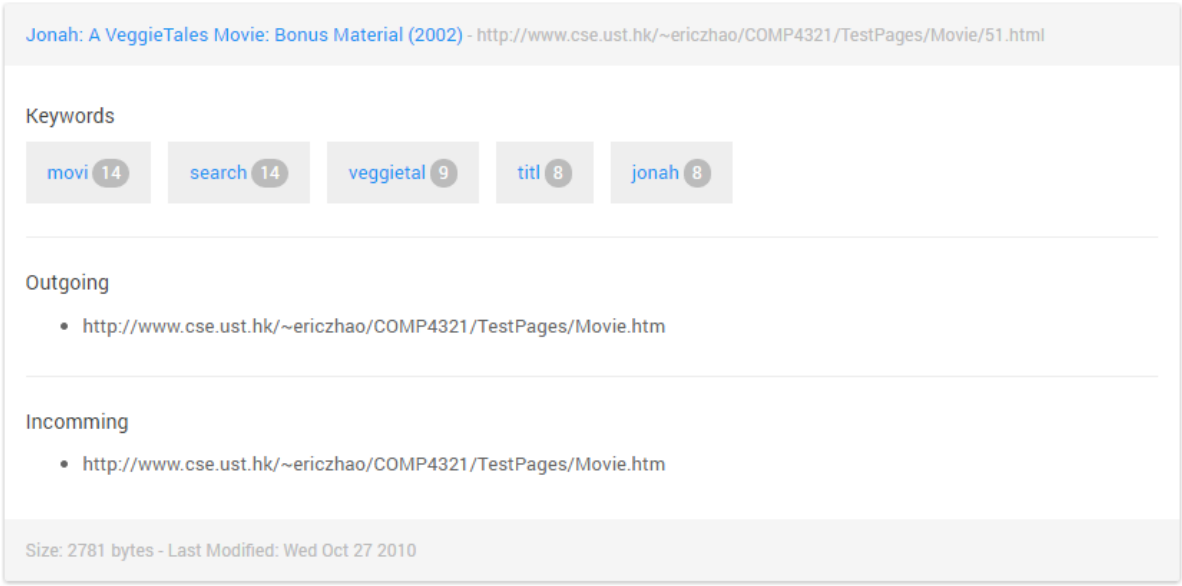
As the user types a query, the search box and the button slide up, and a loading sign is shown. The button is disabled as the search is happening in the background.



Then even bofore the user finises the query, partial results start to show up in the UI. The UI very smartlyand beautifully displays the link keywords, incomming and outgoing links and the link headers such as last modified and size:



The individual search box is very carefully designed to highlight the keyword frequencies. It lightens the last modified and the result page size as they are not normally useful to an average user. The document link is displayed with link to open it into another window so that the search screen is not lost.



Installation procedure

The project build already lives in the main project folder. To start the search engine, just go to the root of the project and run the right executable to launch the search engine. It will start indexing and launch the http server to start searching.

The executable file name to start is dependent on each platform. Currenlty we have the following executables:
Windows: search_windows_amd64.exe / search_windows_386.exe
Mac OSX: search_darwin_amd64 / search_darwin_386 for Mac OSX
Linux: search_linux_amd64 / search_linux_386
FreeBSD: search_freebsd_amd64 / search_freebsd_386

To build these executables if the codebase has been modified, the golang tol chain has to be installed. Along with an excellent Gox utility to cross compile it for tother platforms

Highlight of features beyond the required specification

A lot of features are included which make it a perfect candidate for the project Bonus.

1- We deliver AJAX, which allows webpages to get results in the background without any browser refreshes.

2- Simple and Beautiful UI is delivered. Animations are used in the front page for a very pleasing UI.

3- The entire search engine is real time. The user get the results as they type in the search text box!! This is due to phenominal performance tweaks in the entire system to get the latency down to 20-50 ms per query which is instant fast.

4- It is very ressiliant and recovers from crashes very neatly. The entire indexer has been battle tested with real world links, and has scalled and successfully parsed GB's worth of indexes from the internet. We can give a demo for this and show the scalability of the project on a single machine.

5- Performance. as said in the real time feature, its super fast and utilizes the entire bandwidth of the local computer for indexing and comuting search results.

6- Portable. Due to usage of the go language, the project is very portable. We have static executables for all platofrm in our codebase inclding Windows, Mac OSX, Linux, FreeBSD and Plan 9. And it has *zero* dependencies. Just open the project folder and run the specific executable to start indexing and launch the search engine.

Testing of the functions implemented

There is a tests folder that includes many tests that we run on the project ot make sure we never break things. The search engine is always resiliant as we build it and every change is reliably pushed in our Git repo which we use as a team to build this.

Conclusion

It was an amazing experience building this search engine. We tried a lot of new technologies and algorithms to build a really neat search engine that has the same feel as Google. We believe this project really stands out as a local machine search engine and we may even publish it online for others to try out and try to index most popular web pages to deliver results.