Group: 8

Li Shing Fung

Daniel Schlaug

Lui Tsz Ching

# 

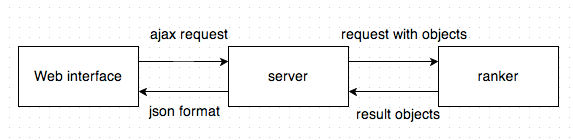
# 

# Gopher

Ambitious and enthusiastic we set out to implement every bonus feature in a superfast search engine made for concurrent processing in an ultra modern language developed by Google itself. And with a cute name to boot! We did learn a lot about Go, indexes of every kind and the huge challenges faced when trying to scale a system like this to handle billions of pages. We also learned that working with a young language like Go means you might have to take to arcane methods in order to overcome the difficulties otherwise handled by well-proven frameworks. Specifically we found no good database supporting nested trees and so had to depend on linear search and modification of the pages in the inverted index. When it was time to do the superfast searching this disadvantage was heavily felt.

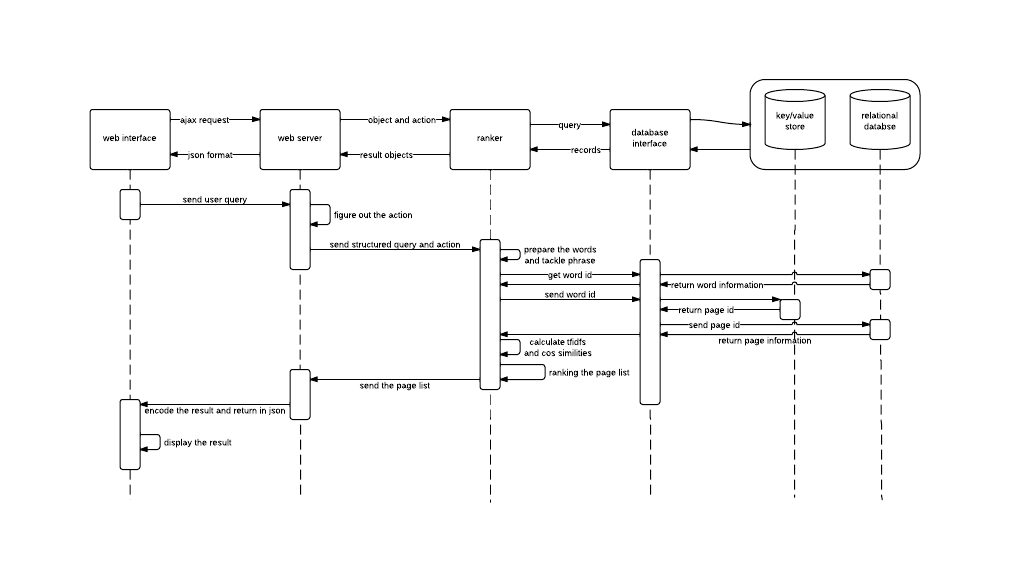
## Design of the system

The search engine can be divided into three parts: web interface, server and ranker.

The The web interface receives does its best to present the user with as convenient a way to convey their desired information that we could muster in the short time left after wrestling with the main tasks of the project.

The server is a full-featured http server and serves up the web interface as well as handling ajax api calls. It acts as a bridge between the javascript code running on the web interface side and the Go that the actual engine is implemented in. To do this we make heavy use of Go’s built-in JSON encoding functionality to send and receive objects over the http connection between the client and server.

The ranker takes a query in the form of a slimmed down page object from the server and compares it to relevant documents in the index. As required it uses tfxidf and cosine similarity to compute ranking scores and return pages and their rankings back to the server.

**The processing of a query**

## Database overview

## Algorithm

### Title weighting

Title’s are given extra weight on the fly in the ranker by comparing the query terms to the words in the title of each returned page and doubling the tfxidf of that word on a match. This could be optimized by letting title words give words a higher tf while indexing.

### Phrase search

We did little to optimize phrase search and it now relies on brute force to get it’s tf and df. We simple get all the pages including all the words in the phrase and remove those where the word positions are not adjacent. We then count the number of adjacent occurrences to get a tf value and the number of documents to get the df. Thereafter the search is handled as usual.

## Scoring functions

### Cosine similarity

### The document vector

## Install procedure

Written in go Gopher's components require a go environment to be compiled and run. Please refer to golang.org for instructions on installing this. Please make sure to put the gopher folder in the src folder of your gopath.

### Package dependencies

Install the packages required for Gopher by running the following:

* go get "code.google.com/p/go.net/html"
* go get "github.com/cznic/exp/dbm"
* go get "github.com/mattn/go-sqlite3"

### Spider

Standing in the gopher directory, run:

$ go run spider.go

### Server

Standing in the gopher directory, run:

$ go run server.go

With the server running you can access the web interface at http://localhost:8081.

## Highlights of features

### Autocompletion

Autocompletion is implemented in the search bar. It will suggest the words from the search history stored locally in the user’s browser.

### Similar Pages searching

The similar pages link extract the top five keywords from a search result and use them to make a new query. Webpages with high cosine similiarities with this query are considered as similar pages and sent back to the user.

### Query History

User’s input query would be stored in html5’s local storage so that user could take a look on them. This allows users to track back on their past queries and make use of them to recontruct queries.

### Database Index exposure

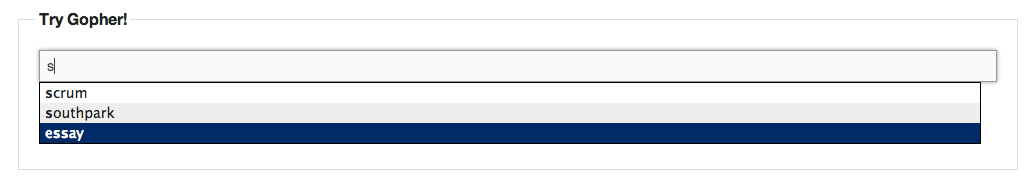
User can get all the indexed words in the database. Superb for TA’s wanting to see that we applied stemming properly.

### User Friendly Interface

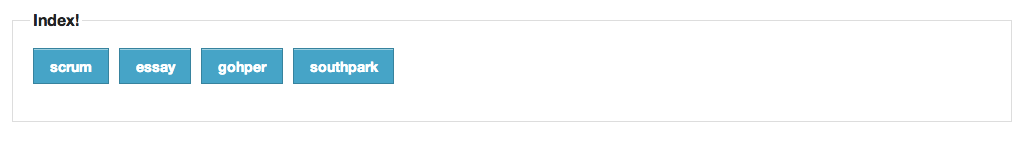
The interface is well designed with prominent buttons and style. User could use the website pretty well without any practise. Friendly features such as “Clear History” is provided in case the history become too messy.

### Example of functions

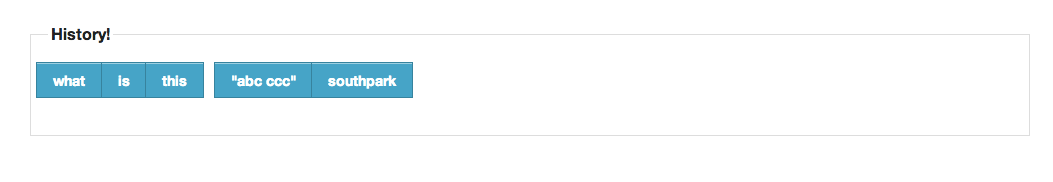
Autocompletion



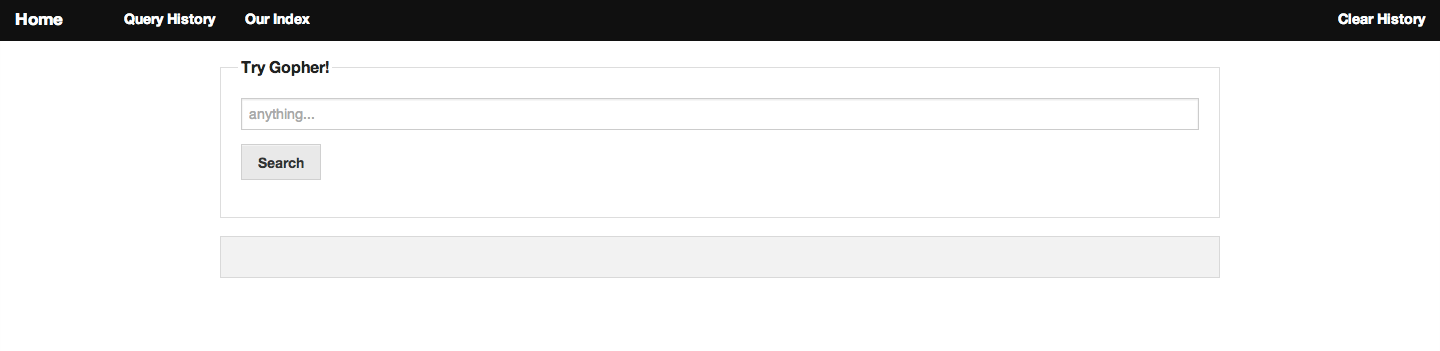
Database Index



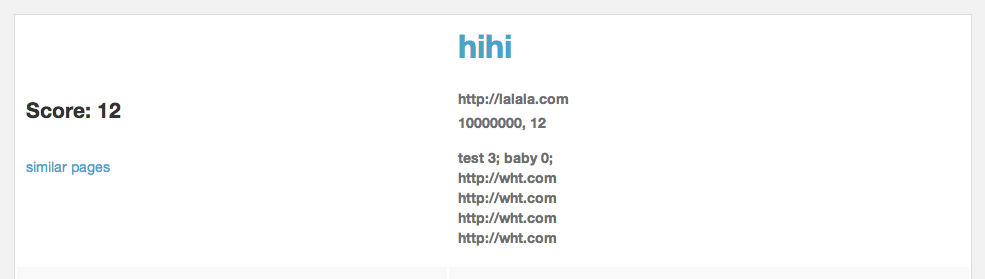
Query History



User Friendly Interface



Similar Pages searching



## Conclusion

### Strengths

The system provides searching service mainly based on term frequency-inverse document frequency, cosine similarities and vector space model. Written in go the project has good potential for concurrency which is currently being used for the spider to request and download an arbitrary number of web-pages simultaneously.

The fact that we are constantly running our own compiled server means that we can respond to api calls super-quickly without having to fire up a process each time a request comes in as in a cgi case.

### Weaknesses

Our system have not been tested enough to be very reliable and should be regarded as highly experimental.

Weighting could be heavily improved with fancy hits such as meta tags and anchor text. Most of modern ranking methods are not implemented.

Our current database implementations will not allow us to scale very well and good solutions to this are not yet available for go but would have to be patched in through bridges to other languages.