

Group 9 Final Report

MOHAMAD Randitya Setyawan – 20316723

PRATAMA Kevin – 2036036

SUSANTO Adrian Prawira – 20369593

OVERALL DESIGN

The search engine consists of four major components: crawler, indexer, and retrieval engine. Most of our system is implemented with the Go programming language and can be run in the Linux operating system. The source tree is divided into packages that implement specific functions (e.g. crawling, database abstraction).

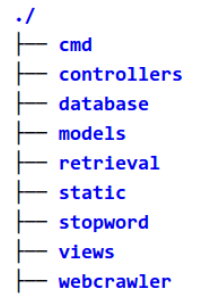


Figure 1 Source tree

Crawling

The crawling application is built using a http client and html parser which are part of the standard library. Building a web crawler presents a unique challenge as the crawler must be robust to handle variations that are beyond the control of the system.

The crawler takes a breadth-first approach when exploring the web by maintaining a single queue of URLs to be explored. The crawler would only insert new URLs into the queue if it does not exist in the index or if the indexed version is outdated. Additionally, the crawler inspects the site's robots.txt (robot exclusion protocol) to check if it can visit a specific part of the website.

When making a GET request to a URL, the html text is returned. The crawler parses the text to extract the words and child links. Any preprocessing such as stopwords removal and text cleaning is done at this stage of the pipeline. The crawler returns a document object which encapsulates data related to one web page (list of words and child links, last-modified time, page size). The document object is then passed to the indexer.

Indexer

The indexer is responsible for inserting documents into the index while updating the forward and inverted indexes accordingly. The indexer maintains the table structures internally (more in the file structure section), and only presents a simple interface for inserting a document. Any low-level operation related to the BoltDB file manipulation are abstracted. The indexer is thread-safe, multiple threads can insert into the index at the same time while guaranteeing correctness by using synchronization primitives such as read-write mutexes.

Because BoltDB only accepts key-value pairs in byte array format, we have to implement our own serialization and deserialization functions. For native data types, we simply take the binary representation and for classes, we use the protocol buffer serialization which is built-in inside Golang.

Retrieval Engine

The retrieval engine implements the vector-space retrieval model using cosine-similarity for the similarity function. The query undergoes preprocessing such as stopwords removal, identical to the preprocessing done in the crawling stage. The preprocessed query is converted into wordIDs and represented as a query vector.

The cosine-similarity scores are computed for documents with at least one query term. The documents are ranked by scores and the top 50 are returned.

Web Interface

The web interface handles all http requests directed at the search engine, obtains the query from the GET parameters and invokes the correct function from the retrieval engine. The result of the query is then sent as a http response. The web interface is implemented using the http server provided by the standard library, and the presentation of results is done using a template engine.

Choice of Language and Library

The choice to use the Go programming language to implement the search engine comes from its simplicity and performance. The language itself was designed to develop web applications, and therefore comes with a comprehensive standard library which includes a http client (for crawling) and http server (for serving web pages). No additional framework is required to build a fully functioning application. From performance point of view, the language has a simple concurrency model using lightweight threads and blocking queue. This feature is used extensively in GoSearch to speed up the indexing and retrieval process.

To build the index, we used the BoltDB library. The library implements key-value store on the local disk using B+ tree as the underlying data structure (similar to JDBM). We chose this library as it is relatively mature, has comprehensive documentation, and has good read speeds.

INSTALLATION PROCEDURE

Before installing the search engine, users will have to install Golang first according to the instructions written in the README document. Afterwards, users can easily install this search engine by running the simple command **make** in the root project directory. Golang will then handle all the project dependencies, allowing users to use the search engine directly.

After running the **make** command, two executables will be created: server and spider. Running **./server** will launch the web interface on port 8008. Running **./spider [-start=<start URL>] [-pages=<num of pages>]** Will crawl the specified page and insert into the existing index. The omission of the parameters will make the spider to crawl the CSE domain and retrieve 300 pages.

ADDITIONAL FEATURES

Relevance Feedback

In GoSearch, relevant feedback feature has been implemented, which allows users to easily find similar pages based on the search result of users' queries. Clicking the "Get similar pages" on a search result will retrieve the five most frequent keywords from that specific page. Using those terms, a query is now formed, which is then used to retrieve a new set of pages.

Site Search
http://www.cse.ust.hk/admin/search/
Date: 30 Apr 2018
Size: 14.4K bytes
Score: 1.2057258506998199
Keywords: research 8; postgradu 7; undergradu 5; alumni 3; cse 3;
Parents:
http://www.cse.ust.hk/
http://www.cse.ust.hk/pg/
http://www.cse.ust.hk/admin/search/
http://www.cse.ust.hk/admin/intrane/
http://www.cse.ust.hk/pg/research/labs/
Children:
http://www.ust.hk/
http://www.cse.ust.hk/admin/search/
http://www.facebook.com/hkust/
http://www.instagram.com/hkust/
http://www.youtube.com/channel/UC1Gky4HCpEOK5EmoUj6zhGg

Figure 2 Get similar page feature

Keywords List

To accommodate for easier query formulation, GoSearch allows users to examine the list of available keywords.

To form a query from these keywords, users can click on the words, which will then add the word to the search box. Clicking on several keywords will chain them, for instance, which forms a more accurate and relevant query compared to a manually created one.

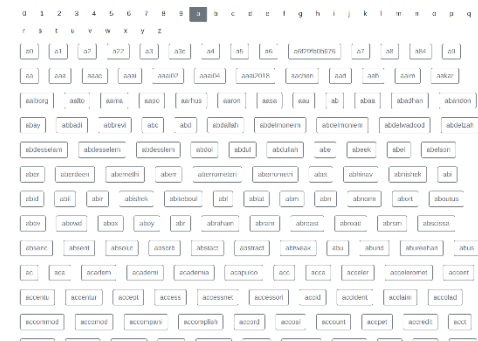


Figure 3 Indexed word list

Query History

Click individual row to add query into the search bar

Query	Time	Search within this query result
abdelmoneim	30 Apr 18 17:25 HKT	<input type="text"/>
search	30 Apr 18 16:35 HKT	<input type="text"/>

Figure 4 Past searches of a user

Sometimes, users may need to refer to past searches to gain better and more relevant queries. To fulfill this demand, GoSearch has implemented a feature, which allows users to view their past activities on the engine. Furthermore, from this list, users can also form a new query by clicking on the respective words to chain them in the query field, which will increase the chance of finding relevant documents significantly. Lastly, to bound the result of a query search, users can search only on the past result by utilizing this feature. Doing so should increase the Recall of a retrieval, because users limit the documents to the desired documents only. Search history is implemented using Cookie. Each user is assigned a unique Cookie ID and the search history for each user is stored in a key-value store separate from the main index.

User Interface

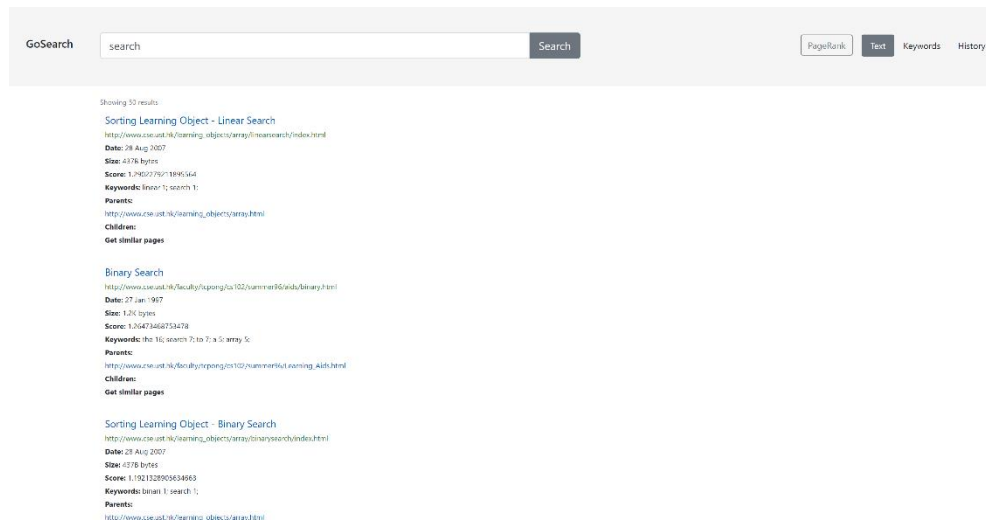


Figure 5 Interface of GoSearch

GoSearch's user interface and experience were designed to be highly intuitive and friendly. By displaying the results neatly, users can easily identify which are relevant. The relevance feedback feature, which finds similar pages, is also located at a visible space and integrated into the result interface.

To develop this user interface, Golang HTML template engine is utilized, with CSS to improve the aesthetics of this interface. In addition, Javascript is also used to make the interface more smooth and interactive. In the keywords list, especially, users can change the displayed list by clicking on the respective initials, which will then switch the list smoothly without having to load new pages. On this page and the query history page, Javascript is also used to fill the query field whenever users click on the respective keywords, allowing for more intuitive and relevant query formulation.

Link-based Retrieval

To avoid low-quality results in web-based retrieval engine, link-based ranking algorithm is essential. Therefore, GoSearch has also integrated the PageRank algorithm for ranking the search results. The PageRank algorithm judges the relevance of a page based on the number of links (referral) it has received from other pages. This model follows the same principle in academic papers where the more important papers would receive more citations from other works. The PageRank scores are query independent and therefore, they are precomputed during indexing time. We utilized the iterative method to compute the PageRank until the values converges.

Special Implementation Techniques

We employed concurrency in the spider, indexer and retrieval engine to improve the performance of the GoSearch. The performance gain from concurrent vs. single threaded execution is significant. For

instance, crawling and indexing 300 pages from CSE website would take around 13 seconds, and 5000 pages take around 12 minutes 13 seconds. The retrieval speed for a single word query would take around 50 milliseconds for a single term query on a 5000 pages document collection, and no longer than 300 milliseconds for a 5 terms query.

```
Indexing 5000 pages took 12m12.975358538s
Updating term weights...
Updating adj list...
Updating page rank...
```

Figure 6 Indexing performance

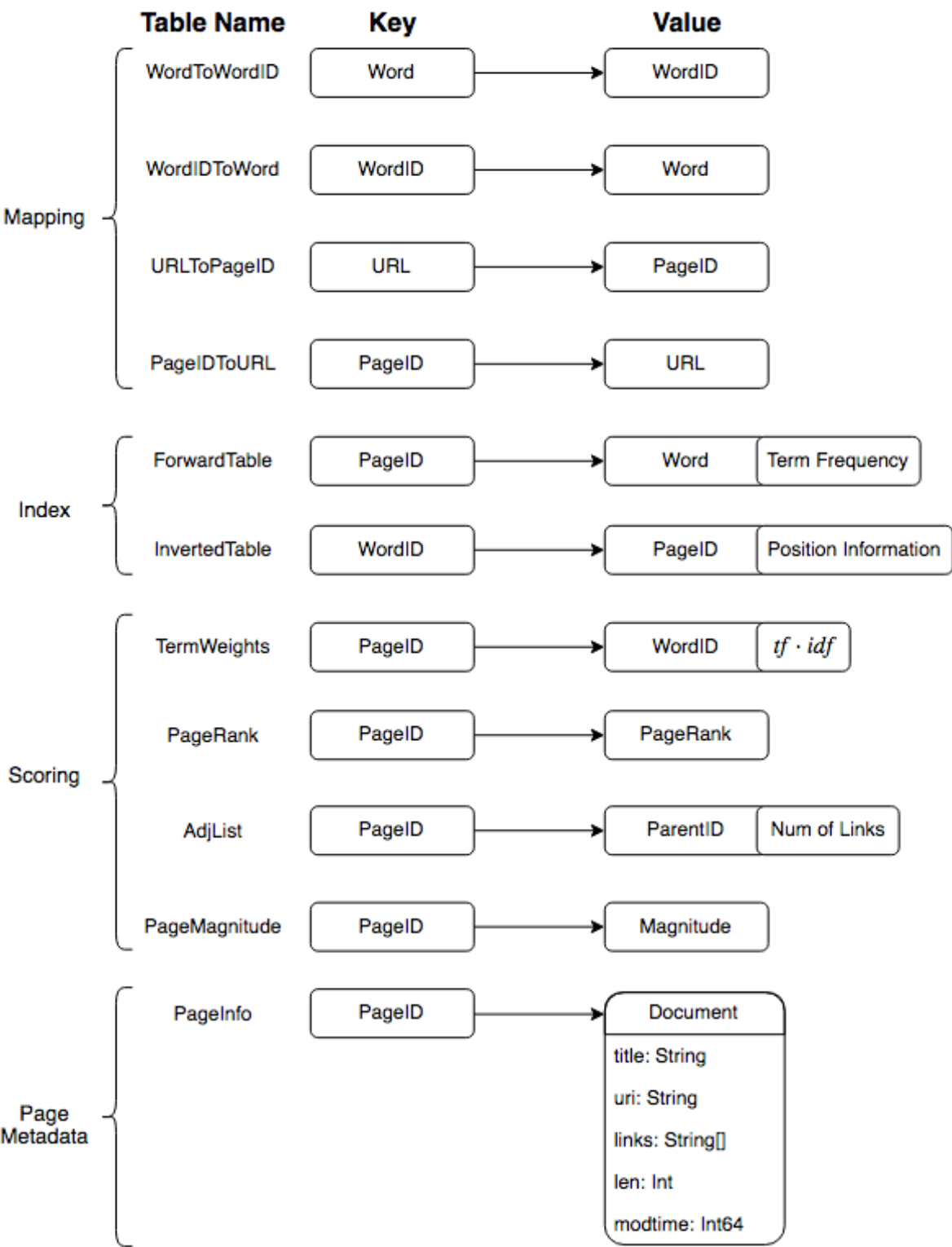
Concurrency is mainly done using Golang's lightweight threads (goroutines) that communicate using a blocking queue. The idea is to divide the work into smaller subroutines and have a thread accomplish each subroutine. For example, during the crawling process, a new thread would be created when a new URL is fetched, and another thread would be used for inserting the page into the index. In the retrieval process, different threads are responsible for reading the documents from the index, and at the same time cosine-similarity scores are computed. In the single-threaded execution model, most of the CPU time would be wasted while waiting for disk or network (IO) access. But in the multi-threaded model, other tasks can be accomplished while one thread waits for IO access.

To speed up the construction of the inverted index, we used the batch insertion algorithm that was introduced in the lecture. Inserting into the inverted index one document at a time is rather inefficient because it incurs large number of random writes. Instead, an in-memory inverted index is created, and once all documents are indexed, the in-memory index is merged into the on-disk index in sorted order. This yields two advantages. First, disk writes will be done in sequential order instead of random order since the keys are sorted. Sequential writes are much faster than random writes in BoltDB. Second, we avoid repeated writes into the same place.

The retrieval speed is made faster by precomputing the tf-idf scores of all the terms in the forward index and storing the magnitude of the page vector on a separate table. By doing so, the computation of cosine similarity score only needs database reads equal to the number of query terms, as opposed to reading all the terms for calculating the normalization term.

FILE STRUCTURES

Table Definition



The database consists of several key/value tables that are stored locally on disk. The package used for our database manager is BoltDB, a persistent key/value storage for the Go programming language. BoltDB's underlying structure for the key/value table is B+ tree, and therefore our tables are implemented as B+ trees. As for the schema itself, the tables are grouped into four main categories.

Mapping

Each word and page URL is represented by a 64-bit unique identifier. Four tables are used to maintain the mappings of (**word** ↔ **wordId**) and (**url** ↔ **pageId**), in which a one-to-one relationship uses two tables for forward and inverse mapping. A separate table is created for the inverse mapping so that the lookup from an ID to its string representation can be done quickly without doing a linear search. Such lookup is done frequently when presenting the search results to the user.

Indexes

These tables provide an integral part of our search functionality which consist of four tables: **InvertedIndex** and **ForwardIndex** for each title and body text. The indexes allow for both forward lookup (**pageId** → **words**) and backward lookup (**wordId** → **pages**). Title and body texts are put into separate indexes to allow different scoring for matches in the title and body.

Scoring

These tables contain scoring related information and must be updated after a set of pages has been added to the index. The scores are precomputed to allow faster retrieval.

- **TermWeights** stores the term weights (**tf • idf**) of all terms in each page.
- **PageMagnitude** table stores the sum of term weights for each term in that page.
- **AdjList** stores the parent IDs of each page, along with the number of pages pointed by that parent.
- From this **AdjList**, the **PageRank** scores are precomputed, which are then stored in the **PageRank** table.

Page Metadata

The page metadata only has one table, **PageInfo**, that stores the additional information of **Document** model (which contains title, child links, last-modified date, maximum term frequency, and page size). The information stored in this table is not used directly to execute searches but are important for the presentation of search results

ALGORITHMS

Vector Space Model

In retrieving relevant terms that match the query information, we used a cosine-similarity Algorithm to determine document-query similarity. Before performing cosine similarity ranking, we first perform a boolean OR retrieval using the query terms. This limits the number of documents we must evaluate since any document without any of the terms would yield a score of 0 and therefore are irrelevant to the query.

To favor title matches, we put the terms present in the index and the terms present in the body in separate indexes. The terms in the body and title are represented as different vectors and the cosine similarity is computed for both. The scores are combine according to the weighting:

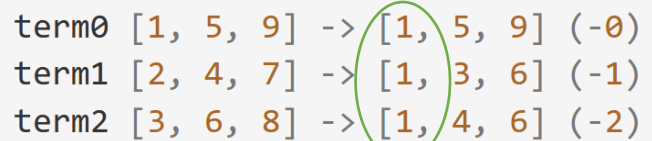
$$Score_{overall} = Score_{body} + 1.5 * Score_{title}$$

Stopwords Removal and Stemming

The implementation of stopwords detection uses the hash table data structure. The stopwords list is constructed into a hash table, and to find out if a string is a stopword, we simply perform a lookup into the hash table. Stemming is also performed on the terms that enter the system. We use the *porter2* algorithm to perform the stemming.

Phrase Search

The inclusion of positional information in our inverted index allows phrase queries to be evaluated. By enclosing query terms in quotes, the search engine will consider the ordering of terms when finding the relevant documents.



```
term0 [1, 5, 9] -> [1, 5, 9] (-0)
term1 [2, 4, 7] -> [1, 3, 6] (-1)
term2 [3, 6, 8] -> [1, 4, 6] (-2)
```

Figure 7 Phrase search illustration, the green circle represents phrase detection

To find out if the terms “**term0 term1 term2**” appear in the specified order, the following actions are done by the search engine. First, a boolean retrieval is carried out using the phrase terms. This narrows down the scope of search from the entire document collection to a select subset. For each document in the intermediate result, we take the positions array of each term, namely **pos0**, **pos1**, and **pos2**. Each position is subtracted by the order of the corresponding term in the query. For example, the positions of **term0**, **pos0**, are subtracted by 0, since **term0** appears first in the query. Likewise, **pos1** are subtracted by 1, and **pos2** by 2. Finally, the set intersection of all positions arrays is evaluated. The resulting set contains the positions of the phrase. If the resulting set is empty, the phrase is not present in the document.

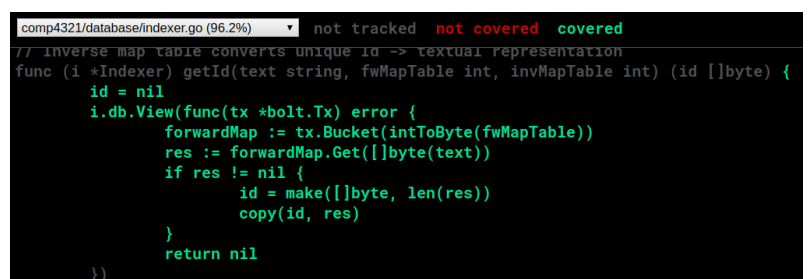
TESTING

Testing is primarily done through unit testing to ensure that non-trivial functions were implemented correctly. The functionalities covered by unit testing include: multi-threaded indexing, constructing adjacency lists for parent-child relations, **PageRank** and **tf-idf** calculations, vector-space and phrase retrieval, and serialization functions. The tests can be run with the **make tests** rule, which invokes Golang’s unit testing suite. In addition, the **make tests_report** rule will display the coverage report in the default web-browser.

In addition to unit testing, we also developed debugging tools to peek into the index itself. This allows us to quickly perform sanity checks on the indexer, and since the tools print into **stdout**, we can use Linux utilities such as **grep** and **diff** for further checking. Another tool we developed is the command line interface (CLI) which allows us to use the search engine without running the webserver. The CLI produces a more verbose output useful for debugging.

```
$ make tests
go test ./database/ ./models/ ./retrieval/ ./stopword/ -cover
ok      comp4321/database    4.105s coverage: 59.2% of statements
ok      comp4321/models      (cached) coverage: 74.5% of statements
ok      comp4321/retrieval   3.366s coverage: 60.7% of statements
ok      comp4321/stopword    (cached) coverage: 91.7% of statements
```

Figure 8 Unit testing



```
comp4321/database/indexer.go (96.2%) not tracked not covered covered
// inverse map table converts unique id -> textual representation
func (i *Indexer) getId(text string, fwMapTable int, invMapTable int) (id []byte) {
    id = nil
    i.db.View(func(tx *bolt.Tx) error {
        forwardMap := tx.Bucket(intToByte(fwMapTable))
        res := forwardMap.Get([]byte(text))
        if res != nil {
            id = make([]byte, len(res))
            copy(id, res)
        }
        return nil
    })
}
```

Figure 9 Coverage report

CONCLUSION

GoSearch is a very powerful and capable search engine, able to index and retrieve with rapid speed and precise accuracy, delivering highly relevant results to users in no time at all. This is possible because of great management and implementation in data structures used, and the effective utilization of concurrency through Golang's goroutines.

However, it is not the best engine existing currently because of some limitations in the development. One of these is the amount of data collected and indexed in the search engine currently. To gather sufficient data, huge amount of resources, both financial and time are required. In addition, huge amount of useful data on the internet are private, which GoSearch team lacks access to. Although it is limited currently, given enough time and resources, in the future the team believes that GoSearch will be able to function and compete against other search engine.

Additionally, GoSearch still has a lot of opportunities to be improved. One feature the team is interested in developing for the engine is the implementation of learning algorithms into the system. Current technologies have proven how powerful learning algorithms can be, if developed and trained correctly. GoSearch, on the other hand, although lacking the algorithms, has implemented functions to gather data from users, which can then be feed into these algorithms, leading to a more sophisticated search engine. Another improvement that the team would like to see is the implementation of image search in the engine. The team believes that image search is an essential feature in a search engine nowadays, which GoSearch is still lacking in. It is highly possible to integrate image search into the system, and it will highly improve GoSearch's advanced engine.

To conclude, GoSearch is currently still in its budding phase, however, it has the capabilities to compete with other search engine given enough time for improvement. The team believes that all the issues that still exists in GoSearch, including the lack of data, can be solved easily with time and resources in the future, leading to a more advanced and modern, yet friendly, simple, and interactive search engine.

CONTRIBUTION

Randitya	Webcrawler, database design, indexer, UI, phrase search	33 %
Kevin	Tf-idf calculations, positional index, indexer	33 %
Adrian	Boolean model, vector space retrieval, pagerank, robots.txt	33 %