Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Description of the crawling, parsing and indexing (methods or libraries used, data structures, description of the overall flow of the system):

The Crawler that was used is the WebSphinx, for the parsing Jsoup was used, and for the indexing our old Spimi code (from project 1 & 2) was used.
“The objective of crawling is to quickly and efficiently gather as many useful web pages as possible, together with the link structure that interconnects them”, which will allow us to index the links, for search purposes. In our case we ignore the interconnecting link structure, and only focus on retrieved document’s content.

To control what a crawler can see/get, the crawler looks for a “robots.txt” which contains instructions for the crawler as to what to retrieve and what to ignore.
A crawler should be robust (not get stuck in web traps = infinite loop for example) and polite (obey the rules)
Most crawlers obey the rules (as they should), but some don’t. Our program obeys the rules.

How a crawler usually works:
-It is given 1 or more root link to start with. (starts at a website) (seed set)
-It analyses the root link(s) and directory and follows every hyperlink on each page.

Detailed Crawler Explanation:
-“It picks a URL from this seed set, then fetches the web page at that URL.”
“The fetched page is then parsed, to extract both the text and the links from the page (each of which points to another URL)”
Usually that’s when the indexer comes in and uses the extracted text (in our case, at this point the only thing we do is give each URL an ID, and store the URL-ID in a TreeMap with the URL as key (as to avoid duplicates). The crawler is given a limit as to how many links to index. After the limit = 1000 docs are retrieved, the crawler is forced-stopped.

Our implementation relies heavily on the WebSphinx library (and http://sujitpal.blogspot.ca/2008/03/crawling-web-pages-with-websphinx.html ’s modifications to it), as well as modifications of ours.

The crawled docs (links-ids) data is retrieved from the TreeMap and inserted in another TreeMap<Integer, String> that has the document ID as key. The file is “1000 docs saved links.blob”
The new TreeMap is then saved to disk for re-usage without re-grabbing the links every time.

After the links are retrieved (crawled or read from file [previously crawled]), they are fed to the parser and indexer. The parser uses (Jsoup) to fetch each online link’s content (title, headers, paragraphs). A failsafe is included (so that if the socket times out, or an exception occurs, the document is re-tried until it works out). The Parser Indexer loads stop words from a file called “stopwords.txt” into an ArrayList<String> which will be used to check if a word is a stop word or not.

The retrieved content is then filtered and “cleaned”:

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Dashes, numbers, non words and white spaces are removed. Words are then retrieved (1 by 1), stemmed, checked that they are not stop words, added to a static vocabulary of ArrayList<String> type (which will be saved to disk, and used further down the road).

An inverted index TreeMap<String, TreeMap<Integer, TreeMap<String, Integer>>> [Term, DocID, Section (title, headers, paragraphs), and Term Frequency] to store the term frequency in a section of a document for a certain term.
We keep track of the number of tokens/words in each, the title, headers, paragraphs, and store for each document ID, the amount of tokens in a static TreeMap<Integer, Integer> (which will also be saved to disk). We will use that TreeMap for calculating an okapi BM25F score for documents later on. We also keep track of other static data (which I decided not to use later, like Okapi BM25F for the whole collection). I decided to do Okapi BM25 for the cluster.

At this point, we've got saved to disk a vocabulary (1000 docs Vocabulary.blob), the number of tokens per documentID (1000 docs DocsTokensAmt.blob) (which we will use for the Okapi BM25 Algorithm), and our inverted index (1000 docs INDEX.blob)

Here on out, we will no longer crawl, parse, and index at each program/project run. We load the essentials directly from file.

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Description of the clustering system:

The clustering system that was used is the K-Means clustering algorithm and was written from scratch.
A clustering algorithm groups a set of documents into subsets or clusters. Documents within a cluster should be as similar as possible and should be as dissimilar as possible from documents in other clusters.
K-Means is a flat clustering algorithm which creates a flat set of clusters “without any explicit structure that would relate clusters to each other.” Each document is a member of exactly 1 cluster, and cannot belong to more than 1 cluster (hard clustering). We also make a fundamental assumption, that documents within the same cluster “behave similarly with respect to relevance to information needs.”

Usually K-Means requires length-normalized document vectors to compute the Euclidean Distance between a Document’s vector, and the K Centroids. But we were told not to normalize. (In case you’d like normalized vectors, the methods have been created [I had extra time])

To be able to use K-Means, we need to have our documents as document vectors, but our Index data structure is under this format:

TreeMap<String, TreeMap<Integer, TreeMap<String, Integer>>>,
which stands for
Term, DocID, Section (Title, Headers, Paragraphs), Term Frequency.

Our first step is to generate an Adjusted Inverted Index:
TreeMap<String, TreeMap<Integer, Double>,
Notice the change from Integer (Term Frequency) to Double. We will need to have floating point values to hold the correct distance values and scores for normalization or centroid computation.

This adjusted inverted index assigns each section a weight (inputed by the user), to calculate a final frequency value for the term in the whole document and not just a section.
(titleweight * tf in title) + (headerweight * tf in header) + (paragraphweight * tf in paragraph) = finalFrequency

To compute euclidean distances or cosine similarity (which has also been implemented for pleasure), document vectors have to be of the same dimensionality M (or length), and thus all document vectors have to contain all words in our vocabulary. And so our next step involves computing a document vector for each document, we aim to go from the previous
Term, DocID, TF structure to a DocID, Term, TF structure which will look like this:
TreeMap<Integer, TreeMap<String, Double>>

Thus we add each word-tf encountered in a same document (same docID) to a new TreeMap<String, Double> structure. Once we’ve processed all words and hence all documents, we go through each document vector and add all vocabulary words that do not exist, with a term

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

frequency of 0. We may optionally decide to normalize the vector, and divide each term frequency by the Euclidean Length (square root of sigma of each (term frequency squared)). Finally, we return a TreeMap<Integer, TreeMap<String, Double>> containing each documentID and the associated document vector.

Our K-Means implementation takes as input: the number of clusters requested/required, the number of iterations it should do, and the set of all document vectors. The inputed documentVectors is a TreeMap holding the documentID as key, and the documentVector as Value.
TreeMap<Integer, TreeMap<String, Double>>.

The document vector TreeMap<String, Double> holds all vocabulary words for a document and its adjusted term frequency for that word in the document.

How K-Means works is as follows:

K seeds or K initial centroids are selected at random (random DocIDs are selected). Many other methods for choosing seeds exist such as:
-selecting 10 random vectors for each cluster, calculating their centroid and setting the seed to that centroid
-heuristics: a) exclude outliers from seed set
b) try out multiple starting points and select the lowest cost
c) obtain seeds from another method

We decided to stick with something simple. We select K random DocIDs and make sure that those document IDs have not been selected before. Thus we select K distinct documents as initial seeds.
After having selected K initial seeds, we create K empty clusters. A 1-to-1 mapping exists between clusters and centroids. Each centroid correspond to 1 cluster, and each cluster has 1 centroid. To do that, I decided instead of having the documentIDs as key, to take those K seeds and set their keys to (1 to K). Each key (1 to K) maps to centroids (1 to K), doing things that way is more manageable.
Our centroids data structure is a static data structure (static access is easier for re-use later on). It is a TreeMap<Integer, TreeMap<String, Double>>, that contains clusterID as key, and centroid vector as Value.

A cluster stores all documents belonging to it:
TreeMap<Integer, TreeMap<String, Double>
(DocumentID, DocumentVector)

We store our clusters in a static
TreeMap<Integer, TreeMap<Integer, TreeMap<String, Double>>>
(ClusterID, Documents belonging to that cluster)
or
(ClusterID, DocumentID, DocumentVector)

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

We then iterate i times, each time destroying the existing clusters and recreating empty clusters (while maintaing the associations between centroids and clusters):
{
    We then iterate over all our documents
    {
        Find the closest centroid to a document by iterating over all centroids and getting the minimum centroid vector having the least Euclidean Distance with the document.

        We then grab the cluster associated with that centroid and add the document to it
    }

    We recompute each centroid vector (since the documents have changed clusters)
}

The way we recompute each centroid vector is:
We pass it a cluster of documents
We create a new vector TreeMap<String, Double> toReturn.
For each term in our toReturn vector, we add up all the frequencies of that term in all documents in our cluster. (including 0 frequencies)

Once all terms have been added from all documents, we proceed to divide each term frequency for each term by the cluster length or size = number of documents in our cluster.

We return the new centroid vector toReturn;

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Description of the user interface:
The UI is built using Swing, the primary JAVA GUI widget toolkit. The code for the UI can be located in the UI package.

To run the UI, run the Query.java file.
The UI consists of two panels,
•The Search Engine Panel for all types of inputs
•The Search Result Panel for the output

The MainFrame.java class contains the two panels.

The SearchEnginePanel.java contains the layout of the search engine. It was put in another class just to be more organized. The panel is added to the MainFrame.java class.

The SearchEnginePanel contains a text input to allow the user to input the query and the following variables:
•K: represents the number of clusters
•it: represents the number of iterations
•k and b and constant variables that can be modified for the OKAPI function
•Tw, Hw and Pw represents the title weight, header weight and paragraph weights respectively
•top: represents the number of top k documents to be retrieved.

Listener.java is the interface that connects between the UI and the Java code

ReceiveEvent.java receives the data sent from the UI through the Listener interface.
It calls the methods needed to retrieve the relevant documents and returns the values to the UI, which will be displayed in the Search Result Panel

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Description of the ranked retrieval querying system:
An analysis of the clustering system:

How many clusters have been created:
In our system we used 5 clusters, but you can modify it by changing the K variable in the User Interface.

What can you tell us about the pages that are clustered together:
The pages that are clustered together are more similar to each other in content than those in other groups (i.e: undergraduate student and graduate student), which means that they may answer the exact or similar information need(s) [they share some information needs they answer, if not all]
We separated the documents into 5 clusters and noticed that the clusters are as following:
1- Graduate studies (software engineering, religion, social-cultural, media studies..)
2- Guides (financial support, tuition fees, academic integrity)
3- Undergraduate studies info (permanent code, excuses, government loans)
4- Information (academic dates, cancelled classes, undergraduate calendar, hospitality)
5- Everything else (not really specific)

For example:
Cluster 1:
http://www.concordia.ca/academics/graduate/calendar/current.html
Document ID: 4
Title: Current
Headers: Fall 2013 Graduate Calendar General Information Faculty of Arts and Science Faculty of Engineering and Computer Science Faculty of Fine Arts John Molson School of Business School of Graduate Studies Feedback forms Application Deadlines
Body: The Calendar is an official University document defining academic programs and regulations. It is accurate as of August?1,?2013?*. It includes all items approved at Senate up until April 19, 2013. ? *

http://www.concordia.ca/academics/graduate/calendar.html
Document ID: 6
Title: Graduate Calendar
Headers: Graduate Calendar Fall 2013 Graduate Calendar Archives About the Calendar Feedback forms
Body: The Calendar is an official University document defining academic programs and the regulations which pertain to them. It is accurate as of August 1, 2013. The University Senate reserves the right to modify the academic programs and regulations at its discretion after the posting date of the Calendar. In addition, the University reserves the right to modify the published scale of tuition and other student fees at any time before the beginning of an academic term. The most current information is available from the School of Graduate Studies.

Cluster 2:
http://www.concordia.ca/students/new/first-year-guide/2-offer-of-admission.html
Document ID: 522
Title: 2. Offer of Admission

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Headers: 2. Your offer of admission Related Links Join the Conversation Deadline for to confirm your Offer of Admission: Institute for Co-operative Education Colleges, Institutes, Schools Feedback forms Accepting your Offer of Admission: Questions About Your Offer of Admission Admission to Colleges and Co-operative Education What Happens if I do Not Meet the Conditions of Admission? What if I Have Questions About the Assessment of my Exemptions, Deficiencies and/or Transfer Credits? Can I Defer my Tuition to Another Term?
Body: Please read and review all of the information in your Offer of Admission once you receive it. If you have any questions regarding the specifics, please contact the individual who signed your Offer of Admission. Should you need any clarification about the terms used on your Offer of Admission, you can refer to our Glossary of Terms. The confirmation deposit is non-refundable but will be credited to your account and applied towards your tuition fees. Please confirm by the date indicated on your Offer of Admission. If you do not confirm by the specified date, your admission may be cancelled. If there is no

Cluster 3:
http://www.concordia.ca/admissions/tuition-fees/permanent-code/about-the-permanentcode.html
Document ID: 143
Title: About the permanent code
Headers: About the permanent code Permanent code surcharge Existing permanent codes New permanent codes Contact us & next steps About the surcharge Surcharge deadlines What a permanent code looks like ? Find your permanent code Permanent codes assigned by francophone universities Required official documents Where to submit official documents Check status and processing time Feedback forms
Body: Everyone needs a permanent code if they study in Quebec. There are no exceptions. If you do not have a permanent code, you will pay a surcharge on top of your tuition fees. Concordia adds a surcharge of?$319 per credit?...

http://www.concordia.ca/students/academic-integrity/excuses.html
Document ID: 170
Title: Excuses won't work
Headers: Excuses won't work Related link Feedback forms I accidentally sent the wrong file to the professor. I don't know what plagiarism means. I come from another country and we do assignments differently there. I don't read my course outlines. I was sick the day the professor talked about plagiarism. My professor is very mean. He/she will deduct marks if assignments are late. I accidentally copied the answer from my friend's exam. My friend showed me his answer to that exam question - I didn't ask for it. Those are not cheat sheets (for an exam), they are study notes. It is the professor's fault. She didn't have to read that book (or article, or website). I had been ill/There was a death in my family/I was going through a divorce/My children were in trouble in school/I was evicted from my apartment... My mother/father/sibling/ cousin/girlfriend/boyfriend was visiting and I had to look after them and take them shopping/ golfing/to the theatre, etc. I asked my friend to hand in the assignment and he handed in the wrong thing. But it is only worth a few marks! I have done this before and have not been caught. Lots of people cheat!
Body: Here are some examples of non-legitimate excuses: This seems to be a big problem with assignments being submitted electronically.

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Cluster 4:
http://www.concordia.ca/events/academic-dates.html
Document ID: 1
Title: Academic dates
Headers: Academic dates Feedback forms
Body: Thanks for your time. Please select your area of feedback.

http://www.concordia.ca/events/cancelled-classes.html
Document ID: 2
Title: Cancelled classes
Headers: Cancelled classes Feedback forms
Body: Thanks for your time. Please select your area of feedback.

http://www.concordia.ca/academics/undergraduate/calendar/current.html
Document ID: 3
Title: Undergraduate Calendar 2013-14
Headers: Undergraduate Calendar 2013-2014 Feedback forms
Body: The?Undergraduate Calendar?is also available in Adobe's Portable Document Format (PDF): Undergraduate Calendar 2013-14 Please note that the online version of the Undergraduate Calendar?is up to date as of February 2013. The online Calendar is an official University document.?It defines academic programs and the regulations that pertain to them. The University Senate reserves the right to modify the academic programs and regulations at its discretion after the posting date of the Calendar….

Cluster 5:
http://www.concordia.ca/students/academic-resources/courses-registration-redirect.html
Document ID: 91
Title: Course registration
Headers: Course registration How to register in a few easy steps: Step 1 ? Academic Advising Step 2 ? Organize your registration materials Step 3 ? Plan Your Timetable and Complete the Registration Worksheet Step 4 ? Registration start dates* Step 5 ? Registration System Availability Step 6 ? Registration Step 7 ? How to Obtain Help Step 8 ? Verify your courses Step 9 - Obtain your Account Balance and Verify Payment Deadlines Connect with us Step 1 ? Academic Advising Arts and Science: John Molson School of Business : Engineering and Computer Science: Fine Arts: Step 2 ? Organize your Registration Materials Qualifying programs Time Conflict English Language Requirements Prerequisites Step 3 ? Registration System Availability Step 4 ? Registration Adding courses Course Section changes Dropping courses Step 5 ? How to Obtain Help Step 6 ? Verify Your Schedule Step 7 ? Obtain your Account Balance and Verify Payment Deadlines Connect with us Feedback forms Chart of Registration Start Dates for SUMMER 2013 Chart of Registration Start Dates for FALL/WINTER 2013 Adding courses Course section changes Dropping courses Netname and Password Instructions Program of Study Verification Address Verification

<u>How does your system use said clusters during retrieval of documents:</u>
Our system takes in the query, filters and “cleans” it:

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

Dashes, numbers, non words and white spaces are removed. Words are then retrieved (1 by 1), stemmed, checked that they are not stop words, and added to a new ArrayList<String> of terms. The ArrayList of terms is then passed to the generateDocumentVectorForQuery method, which will create a query vector for us to calculate similarity/distance with the K centroids.
Our query processor only runs the K-Means clustering algorithms on the first query, as to gain computation time. K-Means is run once each project run. To try different clustering settings, close the project, and restart
If K-Means has not been ran yet (1st query), K-Means is ran with the input configuration settings, clusters & centroids are generated (static, which will allow us to re-use them for other queries). The minimum centroid is found, the cluster associated with that centroid is retrieved, the cluster is converted to an Inverted Index (for the Okapi BM25F algorithm to function).
We want to go from this format:
TreeMap<Integer, TreeMap<String, Double>>
(DocID, Term, Term Frequency)

to

TreeMap<String, TreeMap<Integer, Double>>
(Term, DocID, Term Frequency)

Each term is retrieved from each document, added to a new
TreeMap<String, TreeMap<Integer, Double>>

Every document that contains the term (with a frequency of non-zero) is added to the TreeMap<Integer, Double> of that term. (0 frequencies are ignored)

The Inverted Index for that cluster is returned.
The Average Tokens Per Document for this cluster is then calculated, so that we can run Okapi BM25F (on the retrieved documents from the cluster, and score their relevance according to the cluster to which they belong, instead of scoring according to the whole collection N)
NB: clustered documents are all adjusted indexes (that's the only difference between BM25 and BM25F)
The scores are then calculated for each document.
The documents are ranked, and the top x documents requested are returned to a list.

Our ReceiveEvent calls Retriever_Clusterer.retrieveDocumentsWithIDs(list) which will retrieve the docID, score and the "p"s (paragraphs) of each document and return the output (as a String) to the UI textField.

<u>How would this work on a much larger document collection such as the entire world wide web:</u>
The efficiency of K-means in the world wide web will depend on the number of clusters it implements. Limiting our whole collection to just 3 clusters is clearly not enough. Elements are sparsed (dispersed) into billions of topics. Joining the whole world wide web into just 3 clusters, limits our contents to 3 general topics. This renders all our search results useless; a document could belong to many different "general" topics. Hence we lose the guarantee that documents within a cluster answer the same information need, and are of the same relevance. Also hard

Peter El Jiz - 6250661
Wael Eddy Daouk - 1455818
Chris Caldareri - 9085580

clustering on a huge collection may also be inefficient: limiting a document to just 1 field will result in users not finding what they are looking for, unless they use the correct terminology or the right combination of words to match the cluster's inverted index.
Also going through billions of documents (1 cluster) for each query and ranking it, is less efficient and takes more computation costs, than going through a specific cluster for a specific query. (The clusters are precomputed once, so we gain computation costs). So choosing the right amount of cluster will improve latency, scalability and sparsity.

How you decided to weight the terms and how that influences the results you see:
We chose a value of 5 for Tw, 3 for Hw and 1 for Pw. Since it is more reasonable for a document to be relevant if the Title contains some query terms than a header or paragraph containing the terms. Same thing applies for headers and paragraphs (Headers are more important than Paragraphs). By using these weights we have received more relevant pages for the queries than those received using identical or different weights.

Comparison with Google:

As you can see in the following image, some of the results are similar to the ones displayed on Google's search engine. This proves the efficiency of our ranked retrieval system