Webintelligence - Mini Project 1

Lars Andersen, Mathias Winde Pedersen & Søren Skibsted Als

6. oktober 2014

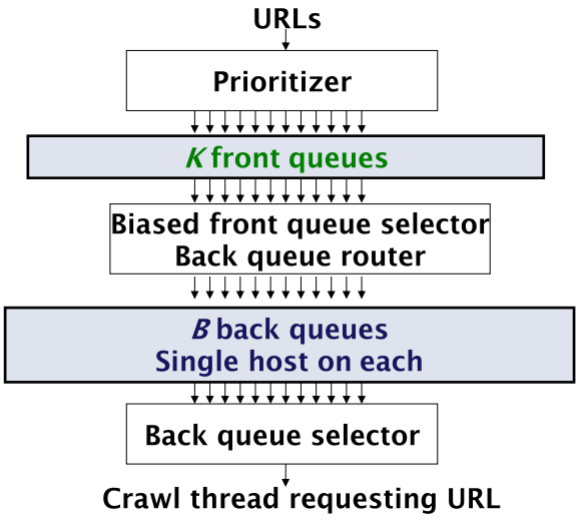
Crawler

The crawler serves as the obtainer of website links. When doing this, it has to make sure that it follows the robots.txt specification. The robots.txt specification file, is a file that is places at the root of website, who wants to indicate what pages they want to have crawled by different search engines. This ensures explicit politeness. Furthermore, it should not visit a site too often. In order to not consume too much space, near duplicate detection is also performed here. The reason near duplicate detection is performed is that a large percentage of the internet content is duplicate information.

We decided to implement the crawler with mercator. Mercator works by having several front and backqueues. The front queues serves as ranking of what websites gets crawled first. As an example, news sites can be put into a frontqueue with high priority, as such sites changes regularly, whereas a static site can have a lower priority for crawling. However, for the front queues we use a single queue, resulting in a uniform ranking, as we have no detection of what is static and what is dynamic websites. The backqueues makes you able to crawl different domains while still ensuring implicit politeness for the crawler. Implicit politeness being not hitting the same domain more than once every two seconds.

Furthermore, the crawling is single threaded, as a proof of concept, but could be expanded to a multi threaded one. This is also related to DNS lookup times, as it was found that some DNS lookups delayed the crawler severely.

An illustration of the Mercator scheme can be seen in Figure 1



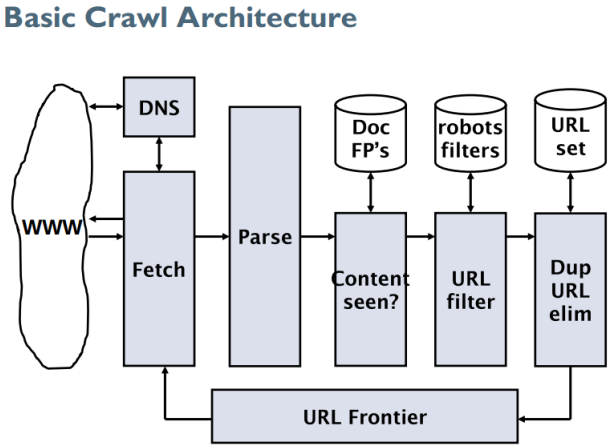
Figur 1: Basic structure of Mercator Scheme, taken from slides.

To check near duplicates we use shingles with sketches. Shingles ia a continuous set of words, which in our case is a set of 8 words per shingle, thus a shingle size of 8, which is the recommended size. Sketches is then to use hashing on these shingles, to reduce the storage necessary, as well as computation time, since the smallest of all the hashing values is chosen for each hash functions, and

compares on this. However, Google recommends 84 hash functions for sketching, but we only use 19 hashes, as we did it as a proof of concept.

Furthermore, tackling documents on the website which is not html or txt encoded is not handled, which gives crawled text encoded in a strange manner, e.g. pdf files. Image files are excluded, as some of those formats made the crawler otherwise crash, when trying to convert it to a string. A library could be used to translate such text into a readable format, to be used for the indexing, and also to make more sensible near duplicate detection.

For the seed we used `http://aau.dk/` and `http://stackoverflow.com/`, which is the starting frontier of the crawler. A figure of the basic structure of a crawler can be seen in Figure 2.



Figur 2: Basic structure of crawler, taken from slides.

Indexer

The indexer has the purpose of constructing the inverted index, in that regard it is important to keep statistics of term and doc frequency, such that this information can be used by the ranker to rank pages. An inverted index is just similar to the back of a book, when you lookup a term, the pages containing the term are referenced. This provides the advantage that only documents containing the term are listed, and thus greatly reducing the memory required for storage. An example is a term only occuring on 100 out of 1 million sites crawled, then you only have to list 100 documents in the index, instead of the whole million, containing a lot of zeroes.

For the indexer we cut corners as follows. We only stem on the English language. Stemming means cutting off unnecessary parts of the words to greatly reduce the index size. We do not develop our own stem method, but instead relies on an already developed one from the internet.

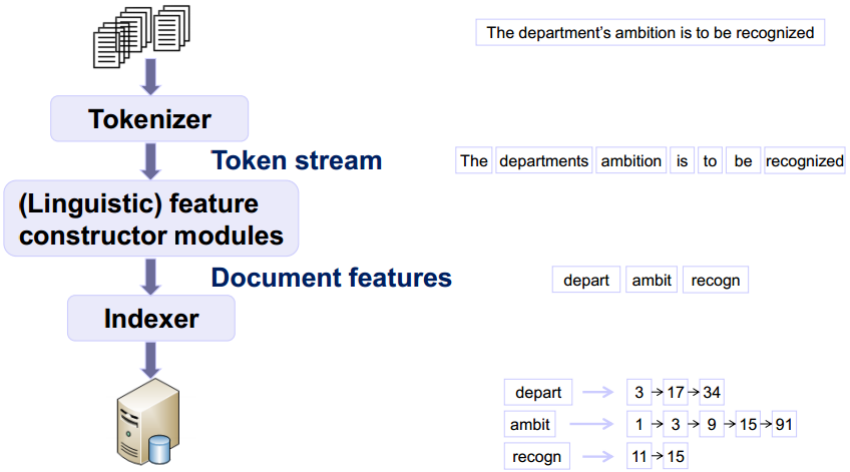
For stopwords it is used for English and Danish, Which are common words without much meaning, such as "the", "about", "then", "by"etc.

As we encode weird strings due to extracting pdf files without a library, we get some strange terms that fill the index unnecessarily. Services could be used

to tackle such file formats, in order to extract the text from the pdf, word, and other such files.

Furthermore, the postingslist are all kept in main memory. This works fine for a small size of websites crawler, however, if you were to crawl the whole web, you would have to have servers farms or write to disk(making it much slower).

A figure showing the basic steps of the indexer can be seen in Figure 3. Where for the inverted index shown in the lower left, we also keep track of the amount of times the term occurs in the document, as that is used for the ranker.



Figur 3: Basic structure of Indexer, taken from slides.

Ranker

The ranker serves the purpose of ranking the pages, given a search query. This means that

This is meaningful, as it contrary to boolean matching can weight different documents, and as of such can rank the results on what matches the query best, and not just if the terms occur in the document or not.

For ranking we used the algorithm provided from the slideshow, which means we used the CosineScore, based on the tf-idf weighting. For contender pruning we used a precomputed champion list, which is a list of the R documents for each term which has the highest weight.

This made it possible to pre-compute this, and significantly reduce the computations needed when ranking, as the champion list limits each posting to a list of r docs, the docs with the most occurrences of the given term the posting is associated with.