Web Intelligence - Crawler Miniproject

Erik Sidelmann Jensen
ejens11@student.aau.dk

Lasse Vang Gravesen
lgrave11@student.aau.dk

Dennis Jakobsen
djakob11@student.aau.dk

1 Crawler Miniproject

1.1 Crawler

The crawler retrieves the documents used for the search engine, it starts by using a list of seed urls. The seed site for ours was `http://en.wikipedia.org/wiki/Internet_of_Things`. Politeness was important, in that the crawler had to adhere to the 'robots.txt' file on websites and determine if the crawler was allowed to visit the url in question and at the same time the crawler should not visit a page too often, this was implemented by simply just waiting a second after each visit. Equality of content on pages had to be checked because there is no reason to have multiple pages with the same content in the search results, we did this using sketches with Shingling and Jaccard similarity. The general idea of shingling can be demonstrated using the following string: "This is a string", with a shingle size of 2, the resulting shingles would be "This is", "is a", "a string" and these strings are then hashed. These hashes can then be used to check equality against a different document using the aforementioned Jaccard similarity.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard similarity is then some value between 0 and 1, and if it's above the threshold say 0.9 the two documents are said to be roughly equal.

Sketching is then when you use multiple hashing functions and just get the minimum for each, and then run jaccard similarity on those.

The crawler then also needs to extract urls from the document using regular expressions, though this is somewhat limiting in that we only get the urls following inside an href attribute. These urls then are normalized, fixed and added to the frontier if they have not already been visited or they are not in the frontier already.

The architecture is somewhat similar to the architecture used by Bo in his slides, though there are some differences specifically with the 'robots.txt' check in that it does that check for each url it visits instead of performing the check before adding it to the frontier. See Figure 1.1.

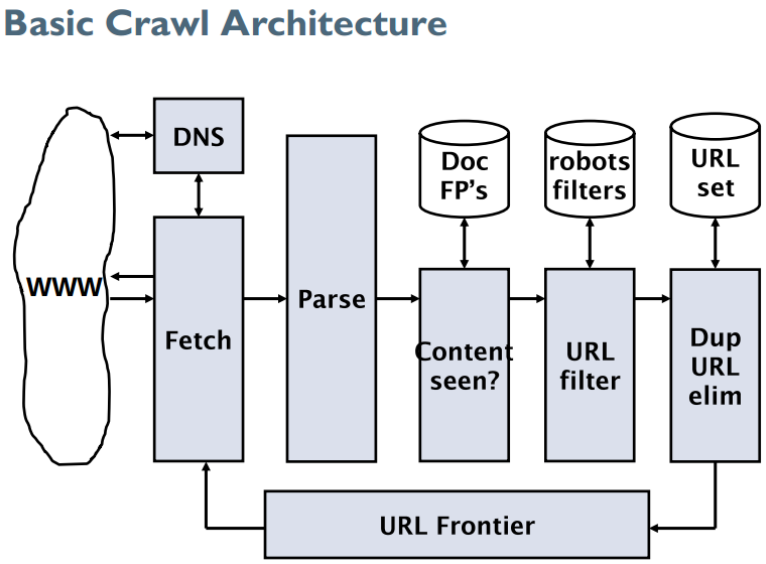

Figure 1.1: Basic Crawl Architecture from Slides

It gets an url from the frontier, fetches the content if it can get past the 'robots.txt' checker, parses out the word content and checks if that has already been seen, and adds it to the url filter after normalizing and checking if it does not already exist in the frontier or already visited urls.

Cut Corners

We could have used the Mercator scheme instead of just sleeping 1 second whenever the Crawler has finished with a page.

The Crawler does find every url in the visited urls, but rather just the ones that are inside href attribute.

The Crawler was limited to very few pages, 100-1000 primarily because the shingling with sketches is extremely slow for some reason.

To actually check for content it might be a good idea to remove all the tags from the HTML content retrieved, but we do not do that.

The sketches for each document is kept in memory, which only works because we crawl very few pages.

1.2 Indexer

The purpose of the indexer is to sort through the crawled documents, and create something that can be easily utilized by the search engine. It does this by providing an inverted index of the terms and which documents contain them and their frequency in said documents. We use an inverted index because then we do not need to investigate every document for matches but rather we can just look at the individual terms and see what documents match. The indexer works by retrieving the crawled pages in the database as an object that contains the html document,

the url and the document id. The html document is converted into only words and these are tokenized by splitting them. These words are made into features, by stemming them using an English language stemmer. It also the English stopwords, such as ”the” or ”a”. It is important to note that the query gets the same treatment, in that its tokenized, stemmed and that the stopwords are removed because if you do not do that the terms in the inverted index will not match. That feature is then added to the inverted index along with the id of the document.

Adding to the inverted index works as such, it receives the term and the document id as mentioned, checks if the inverted index, which is a dictionary with a string key where the term is the key, and a posting list that contains a list of postings.

If the inverted index does not already contained the term, it creates a list of postings, adds a posting with the document id, and adds that to inverted index using the term and a new posting list.

If the inverted index does contain it, it checks if the document id already has a posting in that we might encounter the same word multiple times in the same document. It then sorts the postings.

Otherwise it increments the frequency the term has been seen in the current document.

The structure of the inverted index can be seen in Figure 1.2.

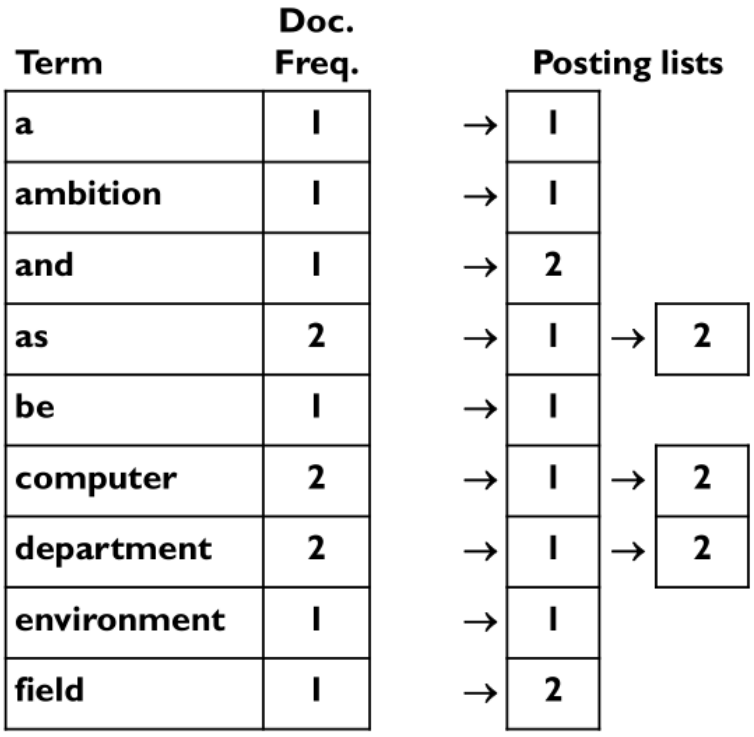

Figure 1.2: The inverted index. Source: From the Lecture 3 slides page 36.

This inverted index is kept in memory, which only works because only a small amount of sites are actually crawled.

The general idea of what the indexer does can be seen in Figure 1.3.

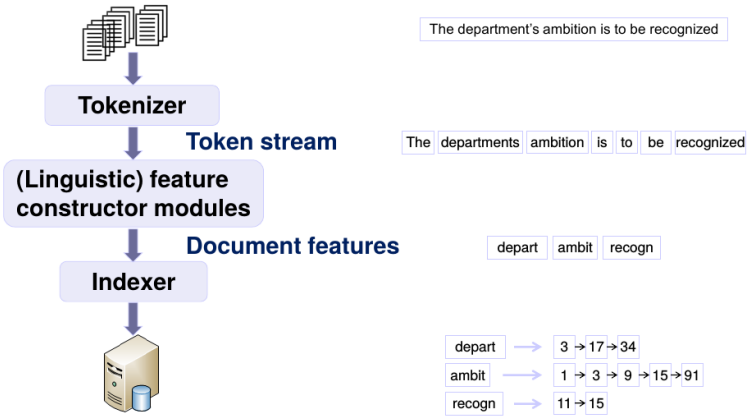

Figure 1.3: Indexer. Source: From the Lecture 3 slides page 7.

1.3 Ranker

The purpose of the ranker is to get the most relevant results for the user given some query. For the ranker we used the cosine score algorithm based on tf-idf weighting. The cosine score algorithm uses cosine similarity between the angle of tf-idf vectors.

The general idea behind that can be seen in Figure 1.4

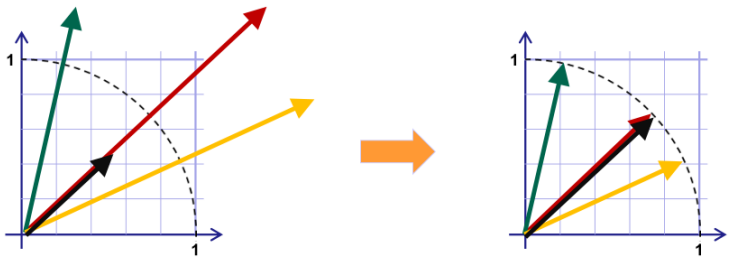

Figure 1.4: The idea behind cosine similarity. Source: From the Lecture 4 slides page 32.

What the ranker does specifically is that it takes a query, tokenizes it and constructs features using the stemmer and stopword removal because the query needs to be treated the same way as the documents were in the indexer.

The tf(Term Frequency) weight is the amount of times a document contains a term.

The tf* weight is

$$log_{10}(1 + tf_{t,d})$$

because relevance does not increase proportionally we need to log10 it.

The df(Document Frequency) weight is the amount of times a term is mentioned in all the documents.

The idf(Inverse Document Frequency) weight is

$$log_{10}(\frac{N}{df_t})$$

where N is the total amount of documents.

The tf-idf weight is then

$$tf * idf$$

Using tf-idf we can then calculate a score for each document using the cosine score algorithm mentioned previously, and this is then used sort the results and provide a good output for the user, for example 10 documents.

tf-idf weighting has different variants, like ltc or lnc. The one we use is ltc.

We also implemented contender pruning using 'Champions Lists' where you precompute for each term t, r amount of docs of the highest weight(the tf weight) and this list is then looked at first instead of the default list of postings as a champion list.

All this comes together to form a search engine, which can be seen in Figure 1.5

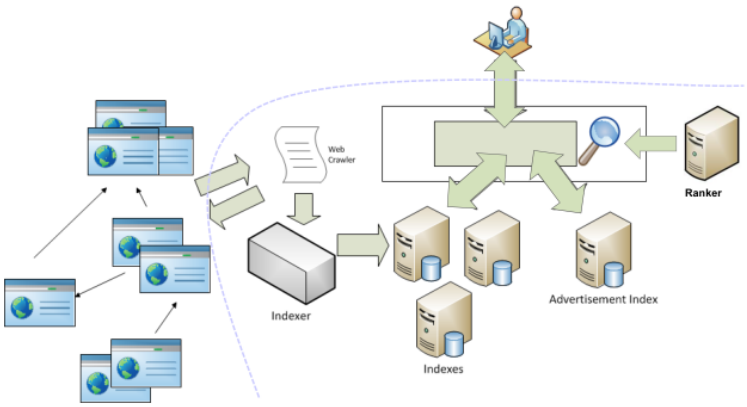

Figure 1.5: Search Engine Architecture. Source: From the Lecture 1 slides page 27.