Aarhus Universitet
Datalogisk Institut
Ny Munkegade
8000 Århus C

January 30, 2003

Algorithms for Web Indexing and Searching
Indexing group report

by
#991836, Jesper Hedegaard
#991834, Martin Stig Stissing
#970541, Aske Simon Christensen
#980117, Mads Brøgger Enevoldsen
#980471, Karsten Strandgaard Jørgensen

1 Introduction

The goal of the project in the course "Algorithms for Web Indexing and Searching" is to create a web search engine. This report describes the process of constructing an index for such a search engine.

2 Overview

The task of indexing is to create a data structure that given a word returns a list of documents containing that word. Having a word and document you should also be able to look up the positions of the word in the document.

To avoid having one gigantic data structure containing all of the above facilities, we have split it in two. First we have an index server that given a word returns a list of documents containing that word; and secondly we have a position server that given a word and a document ID returns the positions of that word in the document.

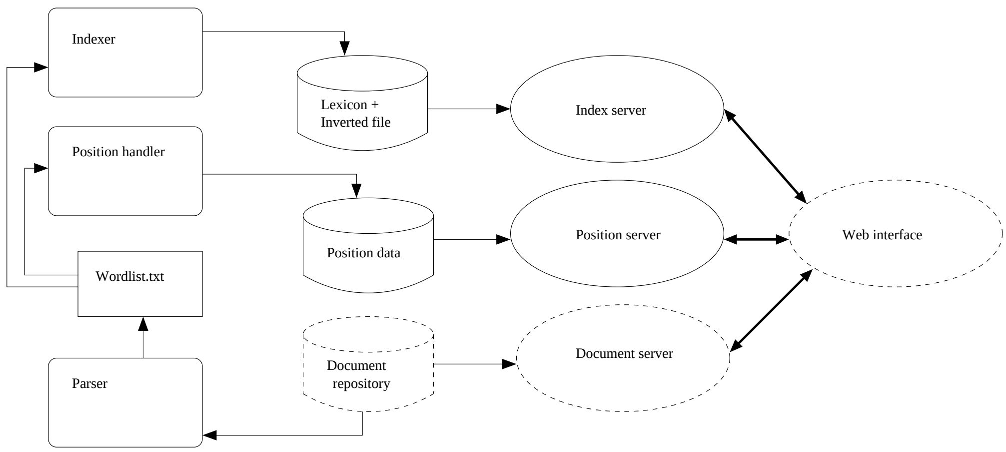

Figure 1: Overview of the code modules, files and databases. The parts created by other groups are dashed.

An overview of the indexing modules and the resources with which they interact can be seen in figure 1

3 Parsing

First attempt was to do it the easy way - use Java's built-in HTML-parser to construct a parse tree that could be traversed to extract the text information. This was an easy approach, but constructing the parse tree was much too time consuming. When processing more than a million documents, time is clearly of the essence.

Instead we implemented a simpler "tags remover" that scanned through the document and skipped all tags, javascript and whatever information that wasn't text. This resulted in a string containing the text we were interested in. But this text had to be normalized making it contain only lowercase words separated by space. This was achieved by scanning through the string once more doing the following:

- Converting upper-case letters to lower-case
- Removing accent marks.
- Skipping special characters.
- Converting HTML character entity references (e.g. , Å etc.) to the letter that they represent or skipping it if it represents a symbol (e.g. <).
- Replacing consecutive spaces with one space.
- Changing numbers with more than four digits in the following way: the first four digits remain unchanged, but from the fifth digit and onwards the symbol # is written instead.

When normalizing text another issue arises: what encoding is the document written in. In a HTML document the `meta`-tag can specify what encoding the document is written in; or it can be specified in the HTTP header. We don't read this information and simply assume that the encoding is ISO-8859-1 or latin 1.

Some tags of the HTML has attributes that could be considered part of the text we were interested in. For instance the "alt" attribute in a <IMG>-tag. We discovered that web pages tend to contain erroneous HTML and trying to fetch the contents of the "alt"-attribute would be complicated. Therefore we decided that our search engine could live without this extra information.

4 Inverted files

This part of the task includes building an inverted file plus an associated flat lexicon. Basically an inverted file contains, for each word in the lexicon, an inverted list, having the IDs of the documents in which the corresponding word occurs. The lexicon then consists of all the words and the position of their corresponding inverted list in the inverted file. Ultimately, an inverted file is a mapping from words to sets of document IDs.

4.1 The initial data

The result of parsing the documents is a mapping between document IDs and sets of words. The format we chose at first, primarily for easy identification of postings (a pair consisting of a document ID and a word), was:

```
document ID, word, document ID, word, ...
```

This was an easy first attempt for all parties involved, but seemed a bit foolish since a lot of redundant information was both written and read (since the documents typically have a lot of words). Therefore we decided on using an alternate format:

```
document ID \n number of words \n word \n word \n ...
```

that doesn't exhibit the above problems and at the same time gives a speedup during reading.

4.2 The Algorithm

The conversion of the mapping from document IDs to words into a mapping from words to documents IDs is basically accomplished by sorting the prior mapping with respect to the words.

The problem is that even small subsections of the internet will produce parsed files of unimaginable size. This implies that one can not hope to do this sorting in memory. Therefore we sort less-than-memory-sized parts of the file, one after the other. These, so-called, sorted-runs, which are basically just minor inverted files, are then merged together, using a multiway merge algorithm, to produce the final inverted file.

Performance and settings of the algorithm for the final run can be seen in section 7.2.

4.3 Compression of the inverted file

A first attempt at compression was not using compression. This didn't compress the file to any larger degree, so it wasn't given any more attention, and was only used for preliminary testing. The second attempt used the Local Bernoulli scheme [1], which is generally considered one of the best compression schemes in the information retrieval literature.

One of the encountered problems using the Bernoulli compression was to understand the exact algorithm given in [1]. The book is not very precise in its description of how to save some bits by using $\lfloor \log b \rfloor$ instead of $\lceil \log b \rceil$ in the algorithm. This expression describes the number of digits that are used to write a given number in an inverted list.

All other articles that we have read about the subject cites this book so we could not find any help there. We solved the problem simply by using $\lceil \log b \rceil$ all the time which is not optimal but not too bad since at least half of the numbers already use $\lceil \log b \rceil$ digits.

A minor detail concerning the Bernoulli compression is that one has to store the number of documents containing a specific word, along with this word's inverted list, since this is used in decompressing the Local Bernoulli encoded list. This number is simply stored as a 32-bit integer at the start of each list and this is of course a waste of space, when a word is in very few documents. It should however not be a major problem when the number of documents increase, since most words will be in many documents.

5 The Index Server

The purpose of the index server is to map search word queries into a list of document IDs indicating the documents matching the query. The server listens for requests from the web interface front end and answers these using the lexicon and inverted file.

5.1 The protocol

To allow independent development of the index server and the web interface front end, an exact communication protocol between these two has been defined. This protocol is described on the following. This description also serves as a specification of the features of the index server.

The server listens for TCP connections on a specific port. The client opens a connection and first sends 32 0-terminated strings in ISO-8859-1 encoding. These strings must match the data indexed by the server, which specifically means that they must be normalized as described in section 3.

All 32-bit integers are transfered in little endian.

The client first sends 32 0-terminated strings in ISO-8859-1 encoding. These strings will be processed as described in section 3 to match the data indexed by the server.

The words are followed by a boolean expression in the following grammar (prefix notation), where all values are single bytes:

$$
\begin{array}{rcll}
q & ::= & n & \text{(where n is between 0 and 31, indicating a search word)} \\
 & | & 65\ q\ q & \text{(and-operation between two queries)} \\
 & | & 66\ q\ q & \text{(or-operation between two queries)} \\
 & | & 67\ q & \text{(negation of a query)}
\end{array}
$$

The boolean expression is, in principle, evaluated for every document in the collection. Word references (n in the grammar) are true if the word occurs in the given document and false otherwise. If the expression evaluates to true, the document is matched by the query. The expression must not evaluate to true for a document that contains none of the search words, since this would return practically all documents.

The server will respond with a 32-bit integer indicating the approximate number of documents matching the query. The actual number can be larger or smaller. If the number returned is negative, it indicates an error. Seven error messages are defined:

-1: Illegal character in one or more search words.
-2: Illegal format of one or more search words.
-3: Illegal byte value in query.
-4: Empty word referenced in query.
-5: Query is true if all word occurences are false.
-6: Internal I/O error.
-7: Unspecified internal error.

If no error occurs, the connection is still open, and the client can begin requesting documents.

A document request is a 32-bit integer indicating how many documents are requested by the client.

The reply from the server will start with a 32-bit integer indicating how many documents will be returned. This will never be more than the requested number. It might be less, indicating that the documents returned are the last matching ones. A negative value signals an error.

For each matching document, the following is returned, in this order:

- A 32-bit document ID.
- A 32-bit PageRank.
- A 32-bit mask indicating which of the search word occurs in the document. The first search word is indicated by the least significant bit.

Documents are returned in order of decreasing PageRank.

Any number of document requests can be sent. The connection stays open until it is explicitly requested closed by the client. This is done by requesting -1 documents.

5.2 The internals

The index server implements the above protocol by merging the inverted lists corresponding to the given search words. Since the document IDs in the inverted lists are sorted by decreasing PageRank, it is sufficient to merge the lists from the beginning until the requested number of document IDs have been extracted. On the other hand, the PageRanks are not stored along with the document IDs, so the server cannot return the actual PageRank for a document. The current implementation returns the number of documents minus the document ID, which is redundant but respects PageRank ordering.

The approximation of the total number of matching documents has not been implemented. The server simply returns zero for this value.

The lexicon is kept completely in internal memory. It is implemented using a trie, mapping search words into start positions of the inverted list for the word inside the inverted file. A trie node contains only the child pointers actually used, and only the nodes that correspond to a word contain an inverted list position. A bit mask indicates which data are present. The pointers and position data are internally distinguishable, allowing leaf nodes to be embedded into their parents.

No special measures have been taken to lay out the trie nodes in a cache efficient manner. A simple depth-first order is used. This allows the trie to be constructed from a sorted list of word/position pairs in linear time with negligible memory overhead. See section 7.3 for the actual values for our system.

When the index server receives a request, it first looks up the positions of the inverted lists for all the search words. For each of these, a document ID stream is created. Such a stream reads the inverted list in blocks and extracts the document IDs from the compressed data. A multi-way merge is then performed on these streams. This gives, for each document ID occuring in the union of the merged lists, a mask indicating which search words occur in the given document. This mask is matched against the boolean expression, and if it matches, the document ID and mask are output according to the protocol.

The inverted lists are always merged linearly. There are no special optimizations for merging very long lists with very short ones. This greatly simplifies both the algorithm and the representation, and as shown is section 7.3, the performance is fully satisfactory.

6 Position data

6.1 Overview

To implement phrase searches in our document collection the query group needed a way to obtain position data from the documents. That is, given a document ID and a word, to get a list of the positions of the word in the document. For instance if a document reads "to be or not to be" then the position list of the word "be" is $[2, 6]$.

To support the need for position data, we have implemented a position handler and a position server. The position handler receives parsed documents from the parser module, builds a position list for each word in each document, compresses these lists and stores them in a database. The position server receives requests from the web interface front end via a TCP connection, looks up the correct position list in the database, decompresses and returns it. So first the handler builds the database and then the server uses it.

The main design considerations when implementing the position handler and server are described in the following sections.

6.2 Temporary data structures

When the parsed documents are received from the parser module they need to be transformed from a string of words to a collection of position lists. Two types of data structures were considered for this purpose: A hash table and a balanced binary search tree.

In both cases, we scan through the document and for each word we meet, we look it up in the data structure and append the current position to its list. When the we reach the end of the document all of the words and position lists are appended to a buffer string, which is later dumped to the database.

Of course, the hash table supports a much faster lookup compared to the binary search tree, and the time it takes to scan through the structures at the end are pretty much the same. So the hash table seems the logical choice. But if we use the search tree, the resulting buffer string for each document is sorted with respect to the words (in lexicographic order), whereas the hash table version of a document is unsorted. And it seems reasonable to assume that inserting sorted data into a database is faster than inserting unsorted ditto.

So the question was: Is the sorting worth the price of the slow lookup? In our case the answer was no. The hash table version outperformed the search tree version by about 5% when tested on actual data.

6.3 Compression algorithm

A position list is a list of integers in increasing order. It is usually quite short as most words occur only a few times in each document. Our database contains millions and millions of these lists, so compressing them is essential to reducing our space consumption. Four different algorithms (and a hybrid of the four) were considered for this task: Fixed-width deltas, Variable-width positions, Exact Bernoulli and Sloppy Bernoulli.

6.3.1 Fixed-width deltas

This method first transforms the position list to a delta-list. That means that only the size of the gaps between the positions are stored. So if the position list is $[p_1, p_2, ..., p_k]$ then the delta-list is $[p_1, \delta_2, ..., \delta_k]$ where $\delta_i = p_i - p_{i-1}$. Then each δ is written in binary with the necessary 0-padding added to make each bitstring as long as the bitstring of the largest δ. This maximum length is then stored in a separate table entry and takes up one byte for each list.

6.3.2 Variable-width positions

This method does not use delta-lists. It utilizes the fact that the positions are listed in increasing order. Before we begin, the bit-length l_{min} of the smallest position is stored separately. Then each position is written in binary using only as many bits as necessary. First we write all the positions that use l_{min} bits, then a 0, then all the positions that use $l_{min} + 1$ bits, then a zero, and so on. With a little thought, this can later be decompressed because the leftmost bit of the bitstring of each position is always a 1.

6.3.3 Two kinds of Bernoulli

The Exact Bernoulli method uses the somewhat complicated Local Bernoulli encoding which works on delta-lists and is described in [1]. One of its properties is that it requires an approximation, b, of the average gap length (takes up 4 bytes) to be stored for each list.

The Sloppy Bernoulli method uses the same Local Bernoulli encoding as above, except that the b is rounded off to the nearest power of 2. And then only the bit-length of this rounded b is stored for each list. It gives a less accurate Bernoulli function, which leads to worse list compression. But on the other hand it always saves 3 bytes on the b compared to the Exact Bernoulli. This is sometimes worthwhile – especially on short position lists.

6.3.4 Hybrid

The hybrid simply tries all the four compression methods above, and picks the shortest result. Two separate bits are used to indicate to the decompressor which method is chosen.

6.3.5 Comparison

The different compression methods were used on various real-life sets of documents from the World Wide Web. As it turned out, all of these datasets did yield roughly the same results. The results presented here are from a set containing around 5 million different words.

2 seperate experiments were conducted. In each experiment three kinds of compression were performed on every position list. For each position list the lengths of the compressed lists were compared and the winning compression technique awarded one point. If two or more techniques tied for the shortest list they both received a point. The final outcome can be seen in figure 2.

It is obvious that the Exact Bernoulli approach is useless for our purpose. Apparently, almost all position lists are very short, so using the 3 extra bytes is too high a price to pay for increased precision.

The Variable-width Positions algorithm and the Sloppy Bernoulli perform almost equally well, but are beaten by the Fixed-width Deltas algorithm. Both Bernoulli algorithms are quite slow to compute, and the Variable-width is only very slightly faster than the Fixed-width, so the latter was the choice of compression in our final system. Naturally, the hybrid outperformed the other algorithms but it was so slow to compute that it was not a suitable choice.

Notice that the Fixed-width Deltas algorithm achieves close to 5 million points in both experiments. This means that one of the other algorithms would have been a better choice for only a very small group of lists (typically very long ones).

6.4 Database table design

The design of the tables in the database is very simple: There is one huge table which contains all the position data. The columns of the table are: document ID, word, size and list. The ID is an integer, the word is a character string and the list column holds the binary object that is the compressed list.

The size column is used for different purposes in each compression technique, for instance in the Variable-width Positions algorithm it stores the minimum bit-length l_{min}. This column is 4 bytes when Exact Bernoulli is used and 1 byte otherwise.

The table is indexed by $(docID, word)$ which gives a very fast position list lookup, even on an enormous table.

With the table designed the way it is, there is some redundancy in the storage of the document IDs. Every document ID is stored as many times as there are words in the document. This could have been avoided by making one table for each document. But that would mean creating millions of tables that each contribute some space overhead to the total space consumption. It would also mean that the database would consist of millions of files in the same directory, which is generally a bad idea speed-wise.

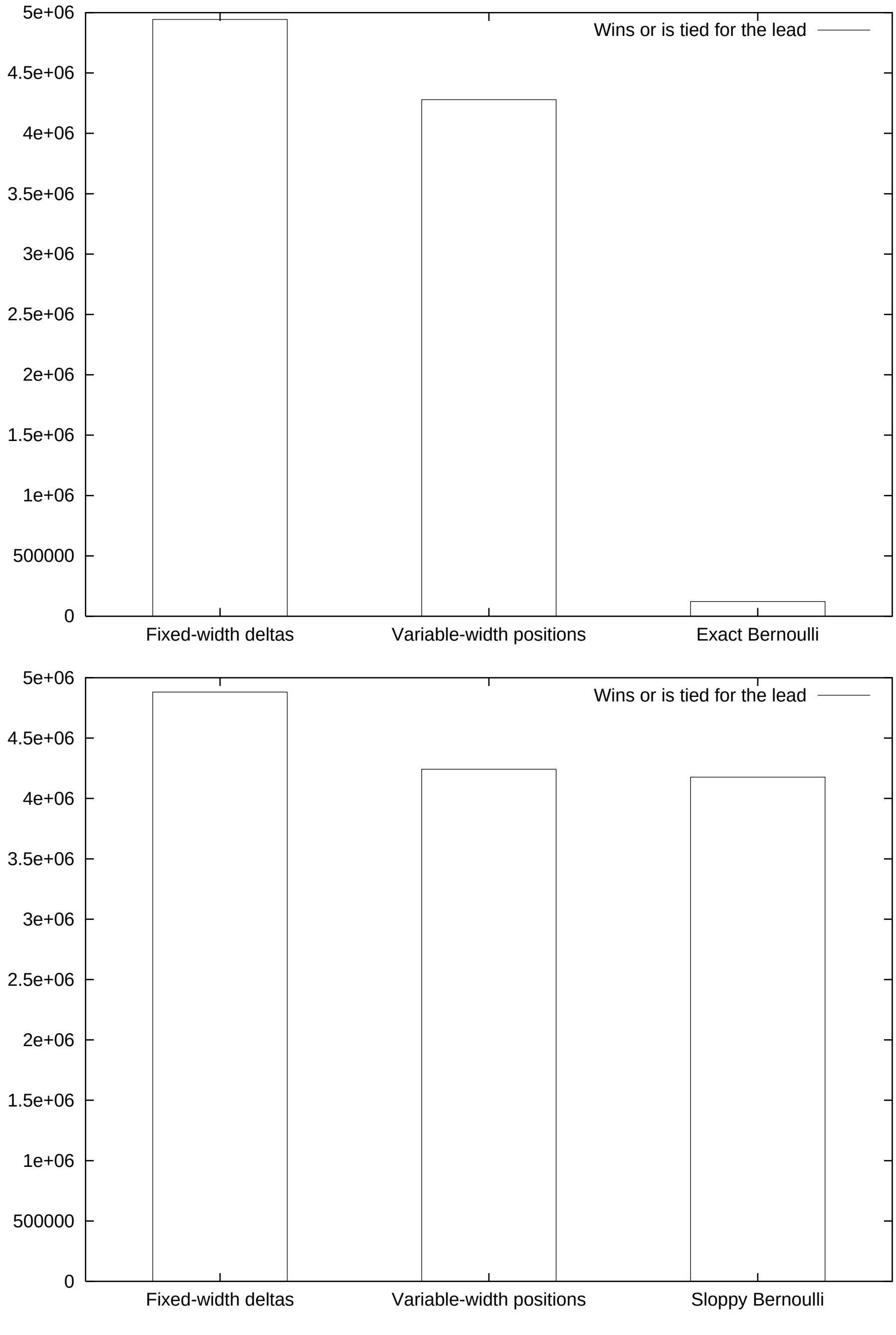

Figure 2: Comparison of compression algorithms. The taller the box, the better.

Actually, to ensure scalability we do not have only one huge table in the final system. We create as many tables of the maximum table size (4 gigabytes) as we need. One such table holds about one million documents, so even large datasets do not suffer the gripes of too many tables that are mentioned above.

6.5 Communication protocol

The communication between the position server and the query engine is quite simple. A TCP connection is established, the web front end first sends all the words contained in the phrases of the user query. Then the following procedure is repeated as many times as needed: The web front end sends a list of document IDs obtained from the index server. The position server then responds with a position list for each pair $(searchword, document)$.

In the final implementation, the length of a document ID list is fixed semi-arbitrarily at 20. A different fixed length – or even a dynamic length – may improve performance on phrase searches.

7 The performance of the final system

Now that we have discussed the implementaion of the system. We turn to the important aspect of practical performance. How fast is it on real data? All experiments were conducted on `snaps`.

7.1 The parser

We measured the performance of the document parser on the different crawls that were available. On all the preliminary datasets that had sizes of up to 722.000 documents we were able to parse around 300.000 documents an hour. But on the final crawl containing 1.700.000 documents the speed was only around 130.000 an hour. So it seems that the parser has scalability problems, yet the origin of the performance drop is not clear.

7.2 Lexicon and inverted file

So how did the compression perform? During the project we have tried our implentation on 3 larger datasets. The results can be seen in the following figures.

The figure shows for each of the three runs, the size of the text file (obtained from the parsing), the size of the inverted file, and the size of the corresponding lexicon. The runs had 305633, 702031 and 1651722 documents, respectively. The size of the inverted file seems to be a constant ratio of the size of the text file, whereas the lex files size ratio compared to the inverted file seems to be decreasing (this is also somewhat expected since there are only a limited number of words in regular use in natural languages.

The final run, that is the one currently used in the MIA search engine, had approximately 1.7 million documents and the parsed file a size of approximately 3.4 GB. Our sorted runs had a size of 6 MB, this was mainly to ensure that less than 1024 files were open at the same time. The time used for constructing the inverted file and the flat lexicon was approximately 210 minutes. 165 minutes were used in producing the sorted runs, the remaining 45 minutes were spent merging these together. The final inverted file had a size of 310 MB, approximately $\frac{1}{11}$ of the parsed file.

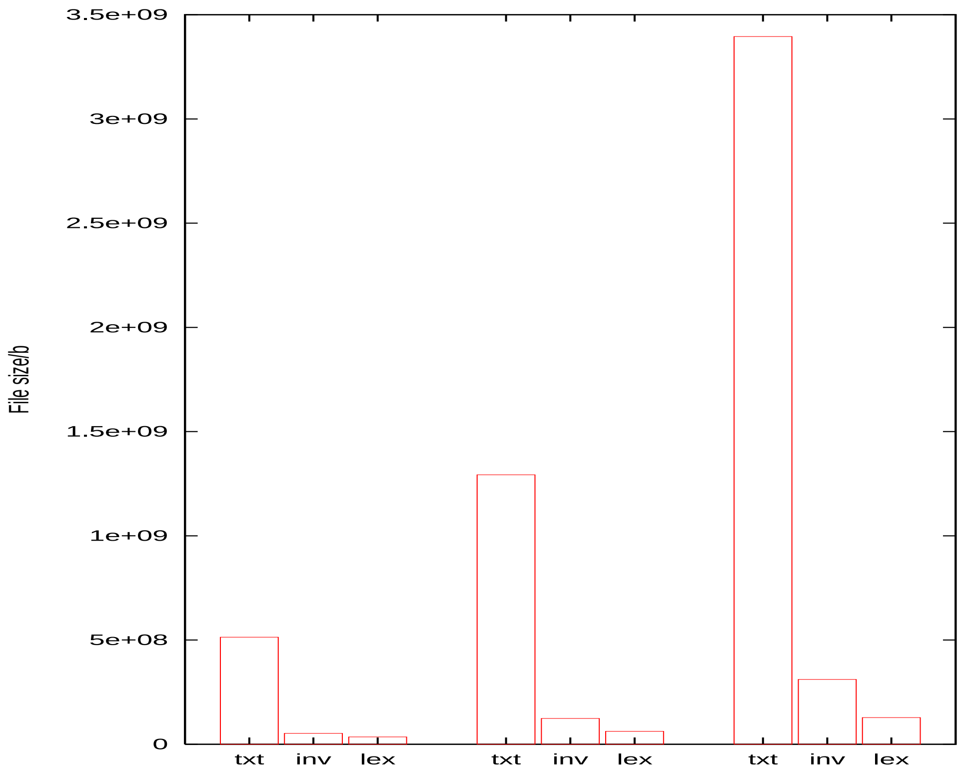

Figure 3: File sizes

The final lexicon file contained around 5 million words and had a size of about 130 MB. Of those 5 million only a small fraction are "real" words. The rest are numbers and arbitrary strings of digits and characters.

7.3 The Index Server

With the lexicon completely in internal memory, some concerns about the index server could be its memory usage and the time used to read the lexicon on startup. With the final lexicon of 5 million words, it uses 253MB of memory for the trie representation of the lexicon, and the loading time is less than 6 minutes. With the server machine having 1GB of memory, and for an index server that only needs to be restarted when a new lexicon and inverted file has been generated, this is certainly no problem.

The time performance of the index server is also fully satisfactory. Typical queries requesting 20 documents are answered in a few milliseconds. A worst-case test (merging many long lists with one short one) was made, requesting the first 20 documents containing all of the words "og", "at", "en", "et", "men" and "røvsnaps". Answering this query took 3430ms. Only 8 documents were returned, implying that the whole of the inverted lists for all the words were traversed.

7.4 Posistion data

The construction of the position database ran at about 350.000 documents per hour. This was about the same as the performance on the preliminary datasets, so no surprises there.

The space used for position data is about 150% of the size of the parsed wordlist. This is a bit more than we would like, but it can be reduced at the cost of the extra running time of the hybrid compression. Also, a final compression of the entire database would bring down the figure dramatically.

8 Final words

We have produced a working indexing engine that performs the necessary number crunching at a satisfying pace, and almost all parts of the engine appear to be completely scalable. We regard this as a succesful completion of our part of the web search engine project.

A Source code

The source files for the parser are:
```
/users/karsten/prog/webalg/DocInfo.java
/users/karsten/prog/webalg/DocQueue.java
/users/karsten/prog/webalg/DocReader.java
/users/karsten/prog/webalg/ParseCrawlerFile.java
/users/karsten/prog/webalg/Parsing.java
```

The source files of the position handler and server are:
```
/users/karsten/prog/webalg/PositionServer.java
/users/karsten/prog/webalg/PositionStorage.java
/users/karsten/prog/webalg/PositionStorageIntf.java
/users/karsten/prog/webalg/StopWordFinder.java
/users/karsten/prog/webalg/WordlistToDB.java
/users/karsten/prog/webalg/KillServer.java
/users/karsten/prog/webalg/scripts/createIndex1.sh
/users/karsten/prog/webalg/scripts/createIndex2.sh
/users/karsten/prog/webalg/scripts/createTable1.sh
/users/karsten/prog/webalg/scripts/createTable2.sh
/users/karsten/prog/webalg/scripts/dumpFileToDB1.sh
/users/karsten/prog/webalg/scripts/dumpFileToDB2.sh
```

The source files for the inverted file are:
```
/users/u991836/programmering/AfWIaS/InvertedFileBuilder.java
/users/u991836/programmering/AfWIaS/InvertedFile.java
/users/u991836/programmering/AfWIaS/BernoulliCompressor.java
/users/u991836/programmering/AfWIaS/BernoulliWriter.java
/users/u991836/programmering/AfWIaS/Posting.java
```

```
/users/u991836/programmering/AfWIaS/PostingReader.java
/users/u991836/programmering/AfWIaS/PostingWriter.java
```

The source files for the index server are:

```
/users/aske/kurser/AfWIaS/afwias/Alphabet.java
/users/aske/kurser/AfWIaS/afwias/BernoulliReader.java
/users/aske/kurser/AfWIaS/afwias/BitReader.java
/users/aske/kurser/AfWIaS/afwias/BoolAnd.java
/users/aske/kurser/AfWIaS/afwias/BoolExp.java
/users/aske/kurser/AfWIaS/afwias/BoolExpReader.java
/users/aske/kurser/AfWIaS/afwias/BoolNot.java
/users/aske/kurser/AfWIaS/afwias/BoolOr.java
/users/aske/kurser/AfWIaS/afwias/BoolWord.java
/users/aske/kurser/AfWIaS/afwias/DocIdReader.java
/users/aske/kurser/AfWIaS/afwias/DocIdReaderFactory.java
/users/aske/kurser/AfWIaS/afwias/IndexConnection.java
/users/aske/kurser/AfWIaS/afwias/IndexServer.java
/users/aske/kurser/AfWIaS/afwias/InitError.java
/users/aske/kurser/AfWIaS/afwias/InvertedListIterator.java
/users/aske/kurser/AfWIaS/afwias/InvertedListIteratorFactory.java
/users/aske/kurser/AfWIaS/afwias/Log.java
/users/aske/kurser/AfWIaS/afwias/Merger.java
/users/aske/kurser/AfWIaS/afwias/Parsing.java
/users/aske/kurser/AfWIaS/afwias/ProtocolException.java
/users/aske/kurser/AfWIaS/afwias/Trie.java
```

References

[1] Managing Gigabytes: Compressing and Indexing Documents and Images by Ian H. Witten, Alistair Moffat, and Timothy C. Bell, published by Morgan Kaufmann Publishing, San Francisco, ISBN 1-55860-570-3.