Fast and Effective Focused Retrieval

Andrew Trotman[1], Xiang-Fei Jia[1], and Shlomo Geva[2]

[1] Computer Science, University of Otago, Dunedin, New Zealand
[2] Queensland University of Technology, Brisbane, Australia

Abstract. Building an efficient and an effective search engine is a very challenging task. In this paper, we present the efficiency and effectiveness of our search engine at the INEX 2009 Efficiency and Ad Hoc Tracks. We have developed a simple and effective pruning method for fast query evaluation, and used a two-step process for Ad Hoc retrieval. The overall results from both tracks show that our search engine performs very competitively in terms of both efficiency and effectiveness.

1 Introduction

There are two main performance issues in Information Retrieval (IR); effectiveness and efficiency. In the past, the research was mainly focused on effectiveness. Only until recent years, efficiency is getting more research focus under the trend of larger document collection sizes. In this paper, we present our approaches towards efficient and effective IR and show our submitted results at the INEX 2009 Efficiency and Ad Hoc Tracks. We have developed a simple and effective pruning method for fast query evaluation, and used a two-step process for Ad Hoc retrieval. The overall results from both tracks show that our search engine performs very competitively in terms of both efficiency and effectiveness.

In Section 2, IR efficiency issues are discussed. Section 3 explains how we achieve fast indexing and searching for large document collections. Experiments and results are shown in Section 4 and 5. Section 6 discusses our runs in the Ad Hoc Track. The last section provides the conclusion and future work.

2 Background

Inverted files [1,2] are the most widely used index structures in IR. The index has two parts: a dictionary of unique terms extracted from a document collection and a list of postings (a pair of <document number, term frequency>) for each of the dictionary terms.

When considering efficiency issues, IR search engines are very interesting because search engines are neither purely I/O-intensive nor solely CPU-intensive. To serve a query, I/O is needed in order to read dictionary terms as well as postings lists from disk. Then postings lists are processed using a ranking function and intermediate results are stored in accumulators. At the end, the accumulators are sorted and the top results are returned. There are two obvious questions; (1) How do we reduce the I/O required for reading dictionary terms and posting lists, and (2) how do we minimise the processing and sorting.

© Springer-Verlag Berlin Heidelberg 2010

When considering effectiveness of Focused Retrieval, it is necessary to consider whether to index documents, elements or passages. This leads to the question of how effectiveness is affected by these index types — we have experimented using document index and post processing to focus.

2.1 Disk I/O

The dictionary has a small size and can be loaded into memory at start-up. Due to their large size, postings must be compressed and stored on disk. Various compression algorithms have been developed, including Variable Byte, Elias gamma, Elias delta, Golomb and Binary Interpolative. Trotman [3] concludes that Variable Byte coding provides the best balance between the compression ratio and the CPU cost for decompression. Anh & Moffat [4,5] construct word-aligned binary codes, which are effective at compression and fast at decompression. We are experimenting with these compression algorithms.

Caching can also be used to reduce disk I/O. There are two levels of caching; system-level and application-level. At the system-level, operating systems provide general purpose I/O caching algorithms. For example, the Linux kernel provides several I/O caching algorithms [6]. At the application-level, caching is more effective since the application can deploy specialised caching algorithms [7]. We are experimenting with caching approaches.

For IR search engines, there are two ways of caching at the application-level. The first solution is to cache query results, which not only reduces disk I/O but also avoids re-evaluation of queries. However, queries tend to have low frequency of repetition [8]. The second is to cache raw postings lists. The challenge is to implement a efficient replacement algorithm in order to keep the postings in memory. We are also experimenting with caching algorithms.

Since the advent of 64-bit machine with vast amount of memory, is has become feasible to load both the dictionary and the compressed postings of a whole-document inverted file into main memory, thus eliminating all disk I/O. For Focused Retrieval a post process of the documents can be a second step. If the documents also fit into memory, then no I/O is needed for Focused Retrieval. This is the approach we are taking, however our experiments in this paper were performed without caching.

2.2 Query Pruning

The processing of postings and subsequent sorting of the accumulators can be computationally expensive, especially when queries contain frequent terms. Frequent terms appear in many documents in a collection and have low similarity scores due to having a low Inverse Document Frequency (IDF). Processing the postings for these terms not only takes time, but also has little impact on the final ranking results.

The purpose of query pruning is to eliminating any unnecessary evaluation while still maintaining good precision. In order to best approximate original results, query pruning requires that (1) every term is assigned a weight [9,10],

(2) query terms are sorted in decreasing order of their weights (such as IDF), (3) the postings are sorted in decreasing order of their weights (such as TF). Partial similarity scores are obtained when some stop condition is met. Either partial or the whole postings list of a query term might be pruned.

Harman & Candeka [11] experimented with a static pruning algorithm in which complete similarity scores are calculated by processing all query terms and postings of the terms. But only a limited number of accumulators, those above a given threshold, are sorted and returned. A dynamic pruning algorithm developed by Buckley and Lewit [12] keeps track of the top $k+1$ partial similarity scores in the set of accumulators, and stops the query evaluation when it is impossible to alter the top-k documents. The algorithm tries to approximate the upper-bound of the top k candidates.

Moffat & Zobel [13] developed two pruning algorithms; the *quit* algorithm is similar to the top-k algorithm and stops processing query terms when a none-zero number of accumulators exceeds a constant value. While the *continue* algorithm continues to process query terms when the stopping condition is met, but only updates documents already in the set of accumulators.

Persin et al. [14,15] argue that a single stopping condition is not efficient enough to maintain fair partial similarity scores. They introduced both a global and a local threshold. The global threshold determines if a new document should be inserted into the set of accumulators, while the local threshold checks if existing accumulators should be updated. The global threshold is similar to the *quit* algorithm, while the combination of the global and local thresholds is like the *continue* algorithm. However, there are two differences; (1) the quit algorithm keeps adding new documents into the set of accumulators until reaching a stopping condition, while the global threshold algorithm adds a new document into the set of accumulators only if the partial similarity score of the document is above the predefined global threshold. (2) The local threshold algorithm only updates the set of accumulators when a partial similarity score is above the local threshold, while the continue algorithm has no condition to update the accumulators.

Anh et al. [16] introduced impact ordering, in which the postings for a term are ordered according to their overall contribution to the similarity scores. They state that Persin et al. [14,15] defined term-weighting as a form of TF-IDF (the global threshold is the IDF and the local threshold is the TF), while Anh et al. used normalised TF-IDF. The term impact is defined as $w_{d,t}/W_d$ where $w_{d,t}$ is the document term weight and W_d is the length of the document vector.

In this paper, we present a simple but effective static pruning method, which is similar to the *continue* algorithm.

3 Efficiency

3.1 Indexer

Memory management is a challenge for fast indexing. Efficient management of memory can substantially reduce indexing time. Our search engine has a memory management layer above the operating system. The layer pre-allocates large chunks of memory. When the search engine requires memory, the requests are served from the pre-allocated pool, instead of calling system memory allocation functions. The sacrifice is that some portion of pre-allocated memory might be wasted. The memory layer is used both in indexing and in query evaluation. As we show in our results, only a very small portion of memory is actually wasted.

The indexer uses hashing with a collision binary tree for maintaining terms. We tried several hashing functions including Hsieh's super fast hashing function. By default, the indexer uses a very simple hashing function, which only hashes the first four characters of a term and its length by referencing a pre-defined look-up table. A simple hashing function has less computational cost, but causes more collisions. Collisions are handled by a simple unbalanced binary tree. We will examine the advantages of various hashing and chaining algorithms in future work.

Postings lists can vary substantially in length. The indexer uses various sizes of memory blocks chained together. The initial block size is 8 bytes and the resize factor is 1.5 for the subsequent blocks.

In order to reduce the size of the inverted file, we always use 1 byte to store term frequencies. This limits term frequencies to a maximum value of 255. Truncating term frequencies could have an impact on long documents. But we assume long documents are rare in a collection and terms with high frequencies in a document are more likely to be common words.

As shown in Figure 1, the index file has four levels of structure. Instead of using the pair of <document number, term frequency> for postings, we group documents with the same term frequency together and store the term frequency at the beginning of each group. By grouping and impacting order documents according to term frequency, during query evaluation we can easily process documents with potential high impacts first and prune the less important documents at the end of the postings list. The difference of document ids in each group are then stored in increasing order and each group ends with a zero. Postings are compressed with Variable Byte coding.

The dictionary of terms is split into two parts. The first level stores the first four bytes of a term string, the length of the term string and the position to locate the second level structure. Terms with the same prefix (the first four bytes) are stored in a term block in the second level. The term block stores the statistics for the terms, including collection frequency, document frequency, offset to locate the postings list, the length of the postings list stored on disk, the uncompressed length of the postings list, and the position to locate the term suffix which is stored at the end of the term block.

At the very end of the index file, the small footer stores the location of the first level dictionary and other values for the management of the index.

3.2 Query Evaluation

At start-up, only the the first-level dictionary is loaded into memory. To process a query term, two disk reads have to be issued; The first reads the second-level

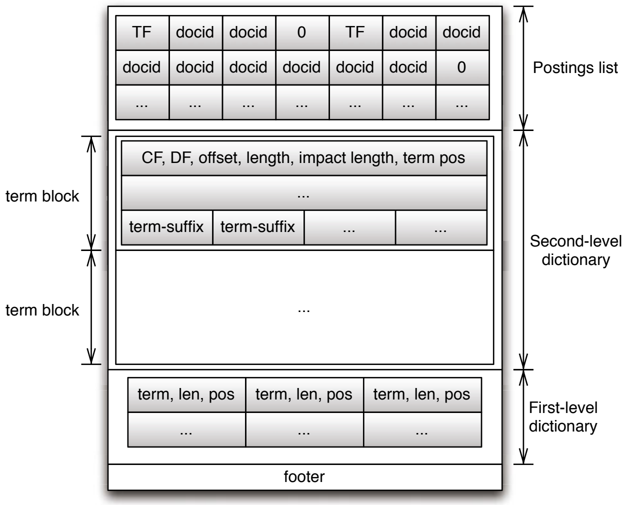

Fig. 1. The index structures

dictionary. Then the offset in that structure is used to locate postings, since we do not use caching in these experiments. The current implementation has no disk I/O caching. We simply deploy the general purpose caching provided by the underlying operating system.

An array is used to store the accumulators. We used fixed point arithmetic on the accumulators because it is faster than the floating point.

We have implemented a special version of quick sort algorithm [17] for fast sorting of the accumulators. One of the features of the algorithm is partial sorting; It will return the top-k documents by partitioning and then only sorting the top partition. Pruning accumulators using partial sorting is similar to that of Harman & Candeka [11]. A command line option (lower-k) to our search engine is used to specify how many top documents to return.

We have also developed a method for static pruning of postings. A command line option (upper-K) is used to specify a value, which is the number of document ids (in the postings list of a term) to be processed. The upper-K value is only a hint. The search engine will always finish processing all postings with the same TF at the K^{th} postings. The combined use of both the lower-k and upper-K methods is similar to the *continue* algorithm.

When upper-K is specified at the command line, the whole postings list of a term is decompressed, even though only partial postings will be processed (this is left for future work). Moffat and Anh [13,18] have developed methods for partial decompression for Variable Byte compressed lists. However, these methods do not come without a cost; Extra housekeeping data must be inserted into the postings lists, thus increasing the size of the index. Further more, there is also a computational cost in keeping track of the housekeeping data.

A modified BM25 is used for ranking. This variant does not result in negative IDF values and is defined thus:

$$RSV_d = \sum_{t \in q} log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1 + 1)\, tf_{td}}{k_1\left((1-b) + b \times \left(\frac{L_d}{L_{avg}}\right)\right) + tf_{td}}$$

Here, N is the total number of documents, and df_t and tf_{td} are the number of documents containing the term t and the frequency of the term in document d, and L_d and L_{avg} are the length of document d and the average length of all documents. The empirical parameters k_1 and b have been set to 0.9 and 0.4 respectively by training on the previous INEX Wikipedia collection.

4 Experiments

We conducted our experiments on a system with dual quad-core Intel Xeon E5410 2.3 GHz, DDR2 PC5300 8 GB main memory, Seagate 7200 RPM 500 GB hard drive, and running Linux with kernel version 2.6.30.

The collection used in the INEX 2009 Efficiency Track is the INEX 2009 Wikipedia collection [19]. The collection was indexed using the default parameters as discussed in Section 3. No words were stopped and stemming was not used. The indexing took about 1 hour and 16 minutes. The memory layer allocated a total memory of 5.3 GB with a utilisation of 97%. Only 160 MB of memory was allocated but never used. Table 1 shows a summary of the document collection.

The INEX 2009 Efficiency Track used two types of topics, with both types having 115 queries. Type A Topics are short queries and the same as the INEX 2009 Ad Hoc topics. Type B Topics are expansions of topics in Type A and intended as long queries. Both topics allow *focused*, *thorough* and *article* query evaluations. Our search engine does not natively support focused retrieval yet, but we instead use a post-process. We only evaluated the topics for *article* Content-Only. We used the BM25 ranking model as discussed in previous section. The k_1 and b values were 0.9 and 0.4 respectively.

We experimented only sorting the top-k documents using the lower-k parameter with k = 15, 150 and 1500 as required by the Efficiency Track. Query terms do not have to be sorted in descending order of term frequency since our pruning method does not prune query terms. We also experimented pruning of postings using the

Table 1. Summary of INEX 2009 Wikipedia Collection

Collection Size	50.7 GB
Documents	2666190
Average Document Length	881 words
Unique Words	11393924
Total Worlds	2348343176
Postings Size	1.2 GB
Dictionary Size	369 MB

upper-K parameter. For each iteration of the lower-k, we specified the upper-K of 1, 15, 150, 1500, 15000, 150000, 1500000. In total we submitted 21 runs.

The disk cache was flushed before each run. No caching mechanism was deployed except that provided by the Linux operating system.

5 Results

This section talks about the evaluation and performance of our 21 submitted runs, obtained from the official Efficiency Track.

Table 2 shows a summary of the runs evaluated on the Type A topics. The first column shows the run-id. The interpolated Precision (iP) reflects the evaluations of top-k documents at points of 0%, 1%, 5% and 10%. The overall performance is shown as Mean Average interpolated Precision (MAiP). The average run time, consisting of the CPU and I/O, is the total time taken for the runs. The last two columns show the lower-k and upper-K parameters. In terms of MAiP, the best runs are Eff-21, Eff-20 and Eff-19 with a value of 0.3, 0.3 and 0.29 respectively.

Figure 2(a) shows the Precision-Recall graph of our 21 runs for Type A topics. Except the Eff-1, Eff-8 and Eff-15 runs, all other runs achieved a very good early precision. Bad performance of the three runs was caused by pruning too many postings (a too small value for upper-K) regardless the number of top-k documents retrieved.

Table 2. A summary of the runs for Type A topics

run-id	iP[0.00]	iP[0.01]	iP[0.05]	iP[0.01]	MAiP	Total time	CPU	I/O	Lower-k	Upper-K
Eff-01	0.22	0.22	0.18	0.14	0.06	77.8	20.4	57.3	15	1
Eff-02	0.35	0.35	0.32	0.29	0.13	77.1	20.4	56.7	15	15
Eff-03	0.48	0.48	0.43	0.36	0.15	77.2	20.0	57.2	15	150
Eff-04	0.55	0.54	0.5	0.44	0.18	78.1	21.0	57.1	15	1500
Eff-05	0.6	0.58	0.53	0.47	0.21	84.5	26.9	57.6	15	15000
Eff-06	0.6	0.59	0.53	0.48	0.21	101.3	43.0	58.2	15	150000
Eff-07	0.6	0.59	0.53	0.48	0.2	122.2	64.9	57.3	15	1500000
Eff-08	0.22	0.22	0.18	0.14	0.06	77.7	20.4	57.3	150	1
Eff-09	0.36	0.36	0.33	0.31	0.14	76.9	19.9	57.0	150	15
Eff-10	0.48	0.48	0.44	0.38	0.19	77.4	20.3	57.2	150	150
Eff-11	0.55	0.54	0.51	0.47	0.23	78.3	21.3	57.0	150	1500
Eff-12	0.6	0.59	0.55	0.51	0.27	83.8	26.9	56.9	150	15000
Eff-13	0.6	0.59	0.55	0.52	0.28	100.0	42.7	57.3	150	150000
Eff-14	0.6	0.59	0.55	0.52	0.28	122.2	64.9	57.3	150	1500000
Eff-15	0.22	0.22	0.18	0.14	0.06	76.9	20.3	56.6	1500	1
Eff-16	0.36	0.36	0.33	0.31	0.14	77.1	20.2	56.9	1500	15
Eff-17	0.48	0.48	0.44	0.38	0.19	77.4	20.1	57.3	1500	150
Eff-18	0.55	0.54	0.51	0.47	0.24	78.5	20.9	57.6	1500	1500
Eff-19	0.6	0.59	0.55	0.51	0.29	83.6	26.8	56.9	1500	15000
Eff-20	0.6	0.59	0.55	0.52	0.3	100.3	42.7	57.6	1500	150000
Eff-21	0.6	0.59	0.55	0.52	0.3	121.7	64.3	57.4	1500	1500000

The relationship between the MAiP measures and the lower-k and upper-K parameters is plotted in Figure 3(a) using data from Table 2. When upper-K has values of 150 and 1500, MAiP measures are much better than the upper-K 15. In terms of lower-k, MAiP measures approach constant at a value of 15000.

To have a better picture of the total time cost, we plotted the time costs of all runs in Figure 4(a) using data from Table 2. Regardless of the values used for lower-k and upper-K, the same number of postings were retrieved from disk, thus causing all runs to have the same amount of disk I/O. The figure also shows that the CPU usage is high when upper-K has a value greater than 1500.

We used the same measures for Type B topics. Table 3 shows the summary of averaged measures for Type B topics. The best runs are Eff-20, Eff-21, Eff-13 with an MAiP measure of 0.18, 0.17 and 0.17 respectively. An interesting observation is that the best run (Eff-20) does not has the highest upper-k value. Processing fewer postings not only saves time, but also improves precision. The best MAiP in Type A is 0.3 while only 0.18 in Type B. We are investigating why.

Figure 2(b) shows the Precision-Recall graph for Type B topics. The Eff-1, Eff-8 and Eff-15 runs also achieved low precision at the early stage. All other runs received good early precision. Figure 3(b) shows the MAiP measures using various lower-k and upper-K values. It shows a similar pattern to that of Figure 3(a). However, good performance is seen when upper-K has a value of 150, rather than the 15000 for Type A topics.

Table 3. A summary of the runs for Type B topics

run-id	iP[0.00]	iP[0.01]	iP[0.05]	iP[0.10]	MAiP	Total time	CPU	I/O	Lower-k	Upper-K
Eff-01	0.24	0.24	0.17	0.14	0.05	380.22	31.97	348.25	15	1
Eff-02	0.34	0.33	0.32	0.29	0.1	367.53	32.07	335.46	15	15
Eff-03	0.35	0.34	0.33	0.29	0.12	367.44	33.41	334.03	15	150
Eff-04	0.38	0.38	0.34	0.32	0.12	373.95	41.72	332.23	15	1500
Eff-05	0.38	0.37	0.33	0.31	0.11	418.02	89.73	328.29	15	15000
Eff-06	0.39	0.39	0.34	0.3	0.11	511.56	184.9	326.66	15	150000
Eff-07	0.39	0.38	0.33	0.3	0.11	542.98	216.97	326.02	15	1500000
Eff-08	0.24	0.24	0.18	0.15	0.05	367.21	32.08	335.13	150	1
Eff-09	0.34	0.34	0.33	0.3	0.13	367.51	32.14	335.37	150	15
Eff-10	0.36	0.35	0.34	0.32	0.16	370.12	33.43	336.69	150	150
Eff-11	0.39	0.39	0.35	0.34	0.16	387.61	41.98	345.63	150	1500
Eff-12	0.39	0.38	0.35	0.34	0.16	419.43	90.03	329.39	150	15000
Eff-13	0.4	0.4	0.36	0.33	0.17	512.54	185.07	327.47	150	150000
Eff-14	0.4	0.4	0.36	0.33	0.16	543.53	216.59	326.94	150	1500000
Eff-15	0.24	0.24	0.18	0.15	0.05	368.33	31.84	336.49	1500	1
Eff-16	0.34	0.34	0.33	0.3	0.14	369.46	32.33	337.13	1500	15
Eff-17	0.36	0.35	0.34	0.32	0.17	378.73	33.23	345.5	1500	150
Eff-18	0.39	0.39	0.35	0.34	0.17	378.19	41.77	336.42	1500	1500
Eff-19	0.39	0.38	0.35	0.34	0.17	421.83	90.11	331.72	1500	15000
Eff-20	0.4	0.4	0.36	0.33	0.18	533.32	184.88	348.44	1500	150000
Eff-21	0.4	0.4	0.36	0.33	0.17	551.8	217.52	334.28	1500	1500000

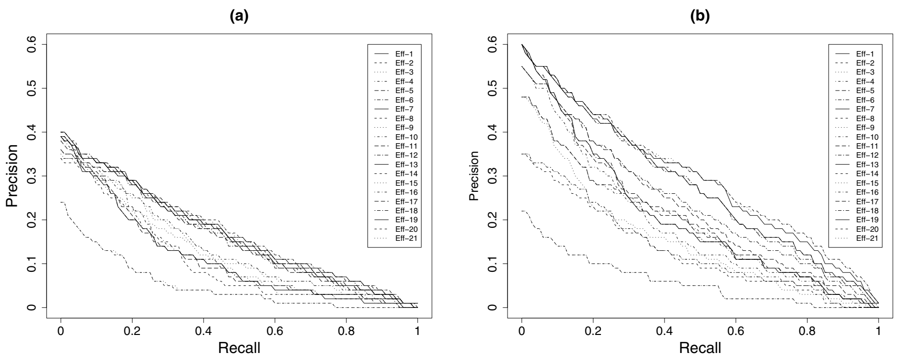

Fig. 2. Precision-Recall plot for (a) Type A and (b) Type B topics

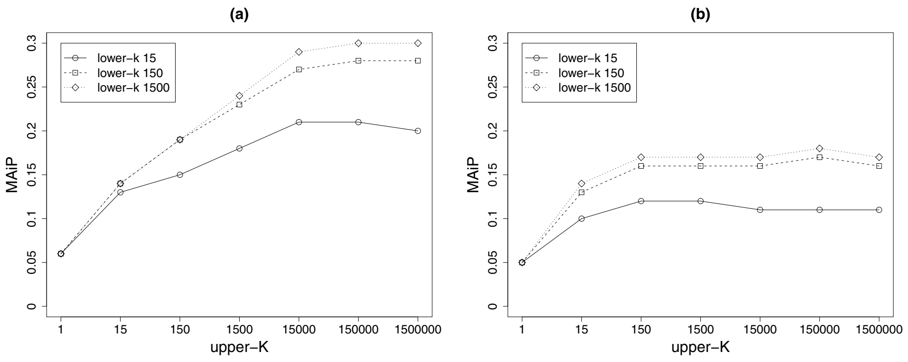

Fig. 3. MAiP measures for (a) Type A and (b) Type B topics

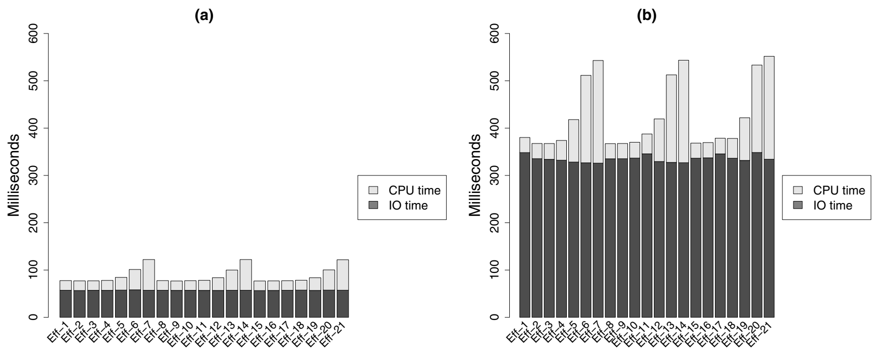

Fig. 4. Total runtime for (a) Type A and (b) Type B topics

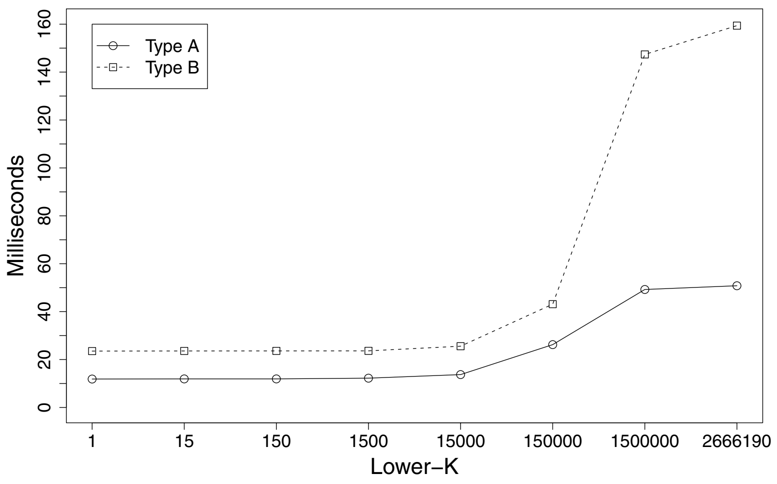

Fig. 5. Times taken for sorting accumulators

The time cost for Type B queries is plotted in Figure 4(b). All runs used the same amount of time for I/O, and have different CPU cost due to various values used for the lower-k and upper-K parameters. The lower-k again has no effect on the CPU cost, and values of 1500 or above for upper-K causes more CPU usage. It took a much longer time for I/O, due to more terms, when compared with I/O cost in Type A.

As shown in both Figure 4(a) and 4(b), the runtime is dominated by the I/O. This leads use to consider that storing the whole index in memory is important.

The submitted runs used small values for the lower-k parameter. In order to see the impact of lower-k, we evaluated both topic sets using only lower-k with a value of 1, 15, 150, 1500, 15000, 150000, 1500000 and 2666190. The number 2666190 is the total number of documents in the collection. As shown in Figure 5, the time taken for sorting the accumulators increases when lower-k has a value above 15000. The sorting times increase from 13.72 ms and 25.57 for Type A and B topics (when lower-k is 15000) to 50.81 ms and 159.39 ms (when lower-k is 2666190) respectively.

6 Ad Hoc

We also used our search engine in the ad hoc track. The whole-document results were extracted and submitted as the REFERENCE run. We then took the reference run and ran a post-process to focus the top 50 results. Our rationale is that document retrieval should rank document from mostly about a topic is mostly not about a topic. If this is the case then focusing should be a fast and relatively simple post process.

6.1 Query Evaluation

Three sets of runs were submitted: Those starting BM25 were based on the reference run and generated from the topic title field (CO) using the BM25 search engine. Those starting ANTbigram were an experiment into phrase searching (and are not discussed here).

For structure searching (CO+S/CAS) we indexed all those tags in a document as special terms. If the path /A/B/C were present in the document then we indexed the document as containing tags A, B, and C. Searching for these tags did not take into consideration the path, only the presence of the tag; that is, /C/B/A would match /A/B/C. Ranking of tags was done with BM25 because the special terms were treated as ordinary terms during ranking. We call this technique Bag-Of-Tags.

Runs containing BOT in their name were generated from the CAS title using the Bag-Of-Tags approach. All search terms and tag names from the paths were included and the queries were ranked using BM25.

The post processing step was not done by the search engine — we leave that for future work. Several different techniques were used:

1. No Focusing (ARTICLE)
2. Deepest enclosing ancestor of all search terms (ANCESTOR)
3. Enclosing element range between first and last occurrence of a search term (RANGE/BEP)
4. All non-overlapping elements containing a search term
5. All overlapping elements containing a search term (THOROUGH)

Runs were submitted to the BIC task (1 & 2), RIC (1-4), Focused (1-4) and thorough (1-5) tasks.

6.2 Results

In the BIC task our run BM25bepBIC placed first. It used BM25 to rank documents and then placed the Best Entry Point at the start of the first element that contained the first occurrence of any search term. Our second best run placed third (RMIT placed second) and it used the ancestor approach.

In the RIC task our runs placed first through to ninth. Our best run was BM25RangeRIC which simply trimmed all those elements from the start and end of the document that did not contain any occurrences of the search terms. The next most effective run was BM25AncestorRIC which chose the lowest common ancestor of the range (and consequently more non-relevant material). Of note, the REFERENCE run placed third – that is, whole document retrieval was very effective.

In the Focused task all our Bag-Of-Tags (CAS) runs placed better than our CO runs. Our best run placed ninth and used ranges, the ancestor run placed tenth and the article run placed eleventh.

In the thorough task our best run, BM25thorough, placed sixth with the Bag-of-Tags placing seventh. We have not concentrated on the focused track.

7 Conclusion and Future Work

In this paper, we introduced our search engine, discussed our design and implementation. We also demonstrated the initial evaluation on the INEX 2009 Efficiency and Ad Hoc Tracks. Our best runs for Type A topics have an MAiP measure of 0.3 and runtime of 100 milliseconds, an MAiP measure of 0.18 and runtime of 533 milliseconds for Type B topics. Compared with the overall results from the Efficiency Track, we believe that our results are very competitive.

Our ad hoc experiments have shown that the approach of finding relevant documents and then post-processing is an effective way of building a Focused Retrieval search engine for the in-Context tasks (where we placed first). They also show that ignoring the structural hints present in a query is reasonable.

Our Focused and Thorough results were not as good as our in-Context runs (we placed respectively fifth and third institutionally). Our experiments here suggest that the Bag-of-Tags approach is effective with our BOT runs performing better than ignoring structural hints in the Focused task and comparably to ignoring the hints in the Focused task. In the Focused task we found that ranges are more effective than common ancestor and (because they are better excluders of non-relevant material). In future work we will be concentrating on increasing our performance in these two tasks.

Of particular interest to us, our runs did not perform best when measured as whole-document retrieval. Our focusing were, however, effective. LIG, RMIT University, and University of Amsterdam bettered our REFERENCE run and we are particularly interested in their approaches and how they might be applied to whole document retrieval (so that we may better our own runs).

In the future, we will continue to work on pruning for more efficient query evaluation. We are also interested in other techniques for improving efficiency without loss of effectiveness, including compression, caching and multi-threading on multi-core architectures.

References

1. Zobel, J., Moffat, A., Ramamohanarao, K.: Inverted files versus signature files for text indexing. ACM Trans. Database Syst. 23(4), 453–490 (1998)
2. Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Comput. Surv. 38(2), 6 (2006)
3. Trotman, A.: Compressing inverted files. Inf. Retr. 6(1), 5–19 (2003)
4. Anh, V.N., Moffat, A.: Inverted index compression using word-aligned binary codes. Inf. Retr. 8(1), 151–166 (2005)
5. Anh, V.N., Moffat, A.: Improved word-aligned binary compression for text indexing. IEEE Transactions on Knowledge and Data Engineering 18(6), 857–861 (2006)
6. Bovet, D.P., Cesati, M.: Understanding the linux kernel, 3rd edn. (November 2005)
7. Jia, X., Trotman, A., O'Keefe, R., Huang, Z.: Application-specific disk I/O optimisation for a search engine. In: PDCAT '08: Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, Washington, DC, USA, pp. 399–404. IEEE Computer Society, Los Alamitos (2008)

8. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 183–190. ACM, New York (2007)
9. Salton, G., Buckley, C.: Term-weighting approaches in automatic text retrieval, pp. 513–523 (1988)
10. Lee, D.L., Chuang, H., Seamons, K.: Document ranking and the vector-space model. IEEE Softw. 14(2), 67–75 (1997)
11. Harman, D., Candela, G.: Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. Journal of the American Society for Information Science 41, 581–589 (1990)
12. Buckley, C., Lewit, A.F.: Optimization of inverted vector searches, pp. 97–110 (1985)
13. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst. 14(4), 349–379 (1996)
14. Persin, M.: Document filtering for fast ranking, pp. 339–348 (1994)
15. Persin, M., Zobel, J., Sacks-Davis, R.: Filtered document retrieval with frequency-sorted indexes. J. Am. Soc. Inf. Sci. 47(10), 749–764 (1996)
16. Anh, V.N., de Kretser, O., Moffat, A.: Vector-space ranking with effective early termination, pp. 35–42 (2001)
17. Bentley, J.L., Mcilroy, M.D.: Engineering a sort function (1993)
18. Anh, V.N., Moffat, A.: Compressed inverted files with reduced decoding overheads, pp. 290–297 (1998)
19. Schenkel, R., Suchanek, F., Kasneci, G.: YAWN: A semantically annotated wikipedia xml corpus (March 2007)