Efficient and Effective Solutions for Search Engines

Xiang-Fei Jia
Department of Computer Science
University of Otago
Dunedin, New Zealand
fei@cs.otago.ac.nz

ABSTRACT

IR efficiency is normally addressed in terms of accumulator initialisation, disk I/O, decompression, ranking and sorting. In this paper, The bottlenecks of search engines are investigated and several solutions for efficiency optimisation are proposed. Particularly, the proposed *heapk* pruning algorithm is very efficient and effective. The future directions of this research work are also discussed.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – Search process; H.3.4 [**Information Storage and Retrieval**]: Systems and Software – performance evaluation (efficiency and effectiveness)

General Terms

Algorithms, Experimentation, Performance

Keywords

Efficiency, Pruning, Impact-ordering

1. INTRODUCTION

Effectiveness and efficiency are two of the main issues in Information Retrieval (IR). Effectiveness has been the main focus of research. In recent years, efficiency has started to draw more attention under the trend of larger document collection sizes.

IR efficiency is normally addressed in terms of accumulator initialisation, disk I/O, decompression, ranking and sorting. A large portion of the performance of search engines is dominated by (1) slow disk read of dictionary terms and the corresponding postings lists, (2) CPU-intensive decompression of postings lists, (3) complex similarity ranking functions and (4) sorting a large number of possible candidate documents [28]. In this paper, IR efficiency issues are addressed and the optimisation approaches are discussed for the search engine used in our research lab.

2. RELATED WORK

Disk I/O involves reading query terms from a dictionary (a vocabulary of all terms in the collection) and the corresponding postings lists for the terms. The dictionary has a small size and can be loaded into memory at start-up. However, due to their large size, postings are usually compressed and stored on disk. A number of compression algorithms have been developed and compared [27, 4]. Another way of reducing disk I/O is caching, either at application level or system level [6, 14]. Since the advent of 64-bit machines with vast amounts of memory, it has become feasible to load both the dictionary and the compressed postings into main memory, thus eliminating all disk I/O. Reading both dictionary and postings lists into memory is the approach taken in our search engine.

The processing (decompression and similarity ranking) of postings and subsequent sorting of accumulators can be computationally expensive, especially when queries contain frequent terms. Processing of these frequent terms not only takes time, but also has little impact on the final ranking results. Postings pruning is a method to eliminate unnecessary processing of postings and provide partial scores for top k documents. Postings pruning can be done at either index time or query time. Pruning at index time reduces the physical size of the index file [10, 19, 8]. However it is a lossy compression; Pruned postings are not kept for access at query time.

Pruning at query time does not modify the index file, but prunes postings at run-time during query evaluation. It allows the approaches of different criterion at query time to keep track of the top k documents. A number of pruning methods have been developed and proved to be efficient [9, 11, 18, 16, 29, 21, 1, 28, 12]. In this paper, pruning at query time is discussed.

Buckley and Lewit [9] developed an upper-bound pruning algorithm which keeps track of the top k and $k+1$ ranked documents and stops query evaluation when it is impossible for the $k+1^{th}$ document to impact the top k documents. Harman and Candela [11] used pairs of $\langle$document id, term weight$\rangle$ for postings instead of traditional $\langle$document id, term frequency$\rangle$. Term weights are pre-calculated at index time and accumulated at search time to form the similarity scores. Postings with low IDFs were statically pruned. Persin [20, 21] developed a pruning algorithm which takes into consideration both the K_{ins} global parameter (term importance across the whole collection, e.g. DF) and the K_{add} local parameter (the number of occurrences of a term in each document, e.g. TF).The K_{ins} threshold determines whether a new document should be either treated as a candidate or ignored. The K_{add} threshold determines if it is worth to update a candidate document which is already in the top k. Moffat et al [18, 16] introduced *QUIT* and *CONTINUE* pruning methods. The *QUIT* method stops processing postings when the number of non-zero accumulators reaches to a pre-defined threshold, while the *CONTINUE* methods continues to refine the order of the already-inserted accumulators. Anh et al [1] developed the any-time stopping pruning algorithm which uses an array of lists (indexed by quantised impact values) to keep track of the current top candidates. Theobald et al [25] presented a pruning method for XML retrieval which maintains the top k candidate documents using a probabilistic score predication.

In my early work [28], the *topk* pruning algorithm is presented. It partially sorts the static array of accumulators using a special version of quick sort [7], and also statically prunes postings. Instead of explicit sorting the accumulators, an improved version of the *topk* was then developed [12]. It keeps track of the current top documents and the minimum partial similarity scores among the top documents. The improved *topk* algorithm was then extended to the *heapk* pruning algorithm [12, 13] which uses a minimum heap to keep track of the top k documents.

Traditionally, postings are stored in pairs of $\langle$document number, term frequency$\rangle$ pairs. However, postings should be impact ordered so that most important postings can be processed first and the less important ones can be pruned using pruning methods [20, 21, 1]. One approach is to store postings in order of term frequency, and documents with the same term frequency are grouped together [20, 21]. Each group stores the term frequency at the beginning of the group followed by the compressed differences of the document numbers. The format of a postings list for a term is a list of the groups in descending order of term frequencies. Another approach is to pre-compute similarity values and use these pre-computed impact values to group documents instead of term frequencies [1]. Pre-computed impact values are positive real numbers. In order to better compress these numbers, they are quantised into whole numbers [18, 1]. Three forms of quantisation method have been proposed (*Left.Geom*, *Uniform.Geom*, *Right.Geom*) and each of the methods can better preserve certain range of the original numbers [1]. In our search engine, we use pre-computed BM25 impact values to group documents and the differences of document numbers in each group are compressed using Variable Byte Coding by default. We choose to use the *Uniform.Geom* quantisation method for transformation of the impact values, because the *Uniform.Geom* quantisation method preserves the original distribution of the numbers, thus no decoding is required at query time. Each impact value is quantised into an 8-bit whole number.

Since only partial postings are processed in query pruning, there is no need to decompress the whole postings lists. Skipping [16] and blocking [17] allow pseudo-random access into encoded postings lists and only decompress the needed parts. Further research work [3, 2] represent postings in a fixed number of bits, thus allowing full random access. Our search engine partially decompress postings list based on the worst case of the static pruning, we do not implement skipping. We can simply halt decompression after a fixed number of postings have been decompressed.

A number of accumulators, usually as a static array, need to be created and initialised for *term-at-a-time* processing [9, 11]. The accumulators hold the intermediate accumulated results for each document. For large collections, a large number of accumulators has to be used. Initialisation of large number of accumulators can take time. This is the criticism of the *term-at-a-time* approach. Alternatively, the *document-at-a-time* approach ranks one document at a time, thus does not need to hold intermediate results [30, 26]. However, the *document-at-a-time* approach requires random scan of postings lists, which takes time [24].

3. MY RESEARCH

This research is about (1) identifying the bottlenecks in the search engine, (2) devising with efficient and effective solutions to minimise or eliminate the bottlenecks and (3) adopting the solutions for distributed IR.

As I have shown elsewhere [28], the bottlenecks are (1) slow disk read of dictionary terms and the corresponding postings lists, (2) complex similarity ranking function and (3) sorting a large number of possible candidate documents. Disk I/O can be completely eliminated by simply storing the index in memory. Similarity scores are pre-computed at index time and stored in the index. During query evaluation, the scores can be simply added together. Postings pruning is an efficient and effective method to reduce the number of possible candidate document, thus reduce the time to sort.

3.1 Disk I/O

Although on a cluster of 64-bit servers it is often possible to store the entire index in memory, it is not reasonable to assume all uses will have this kind of hardware. Such is the case when the search engine is run on a laptop or on a mobile phone. In this case the index is stored on disk.

In order to speed up disk access, general-purpose operating systems usually provide buffer caching, prefetching and scheduling optimisation algorithms. However, the I/O algorithms are general-purpose, as these operating systems have to serve various kinds of applications. For special-purpose applications, it is better for application to bypass the general ones and deploy their own I/O optimisation algorithms.

I have introduced and tested [14] application-specific I/O optimisations for search engines. One way of implementing a cache is to deploy a buffer replacement policy, like Least Recent Used (LRU) or Least Frequent Use (LFU), which defines how efficiently a finite amount of cache memory can be used for large amounts of data on disk. This is so called dynamic caching. Another approach is to define which postings lists are the most important and then let them stay in cache memory without eviction. This is so called static caching. Both dynamic and static caching have pros and cons. I deployed a new caching algorithm which combines both dynamic and static caching. The static part caches high document frequency words since these words have long postings lists and so take considerable time to read from disk. A simply LRU cache policy is used for dynamic caching. The cache policy is defined to cache postings of terms which were seen more than once in the query stream. Prefetching and scheduling are straightforward.

The disk access pattern of a search engine can be predicted from the query terms. Postings lists for the next term can be prefetched while the current term is being processed. If we consider that postings lists are sorted in alphabetic order of the dictionary terms, we can define a new scheduler which sorts disk I/O requests in the order of the dictionary terms, which equates to linear order in the index file. The sorting can be either local or global, where local means sorting terms in a single query and global means sorting terms in several queries executed concurrently.

Four different disk I/O methods were tested. In READ the operating system disk read() function was called. In O_DIRECT the Linux O_DIRECT flag was set for all file I/O (which turns off operating system caching). READ OPTIMISED used read() and my optimisations, and O_DIRECT OPTIMISED used O_DIRECT with my optimisations. The optimised versions include the scheduler, postings list prefetching, and caching. Testing was on .GOV with the Million Query Track queries. The specification of the hardware and complete results are presented in [14]; but in summary, O_DIRECT OPTIMISED is 73% faster than O_DIRECT and 28% faster than READ. The results also show an 11% improvement of O_DIRECT OPTIMISED over READ OPTIMISED.

3.2 Pruning

By putting the index into memory the I/O bottleneck disappears. In this case the processing of the postings list becomes the bottleneck. To alleviate this others have proposed top-k searching. In [28] I developed a simple top-k algorithm called *topk*. It uses a special version of quick sort 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. A command line option (*lower_k*) to our search engine is used to specify how many top documents to return. Another command line option (*upper_k*) is used for static pruning of postings. It specifies a value, which is the number of (impact ordered) postings to be processed.

Based on the *topk*, I further developed an improved version of the *topk* [13, 12]. Instead of explicit sorting of all accumulators, during query evaluation it keeps track of the current top documents and the minimum partial similarity score among the top documents. The improved *topk* uses an array of pointers to keep track of top documents. Two operations are required to maintain the top documents, i.e. *update* and *insert*. If a document is in the top documents and gets updated to a new score, the improved *topk* simply does nothing. If a document is not in the top k and gets updated to a new score which is larger than the minimum score, the document needs to be inserted into the top. The insert operation is accomplished by two linear scans of the array of pointers; (1) the first scan locates the document which has the minimum score and swaps the minimum document with the newly updated document, (2) the second finds the current minimum similarity score.

The performance of the improved *topk* grows exponentially when there are large numbers of documents and postings to be processed. To resolve this problem, I developed the *heapk* pruning algorithm [13]. It uses a minimum heap to keep track of the top documents. Instead of using the minimum similarity score, the *heapk* uses a bit string to define if a document is among the top k. The heap structure is only built once which is when the number of top slots are fully filled. Two special functions (*min_update()* and *min_insert()*) are implemented for efficiency optimisation. Every time one of the top candidate documents gets updated, the *min_update*() function is called. It first linearly scans the *heapk* array to locate the right pointer and then partially traverse down the subtree of the pointer for proper update of the minimum heap. The linear scan is required because minimum heap is not a binary search tree. Every time a new document is going to be inserted into the minimum heap, the *min_insert*() function is called. It first replace the document with the smallest score and then partially traverse down the tree for proper update of the minimum heap.

Four sets of experiments were conducted. The first set was the base set, which did not use any optimisation. The other sets were for the *topk*, the improved *topk* and *heapk* algorithms. The test collection was the INEX 2009 Wikipedia collection [22] and the test queries were the 107 topics in INEX Ad Hoc 2010 [5]. Only title was used for each topic. The collection was indexed twice, one for the base set, and one for the *topk*, the improved *topk* and the *heapk*. For the base and *topk* sets, term frequencies were used as impact values. For the improved *topk* and *heapk*, pre-computed BM25 similarity scores were used as impact values. For both indexes, S-Striping stemming was used. The experiments were conducted as part of the INEX Efficiency Track in 2009 and the efficiency task of the ad hoc track in 2010.

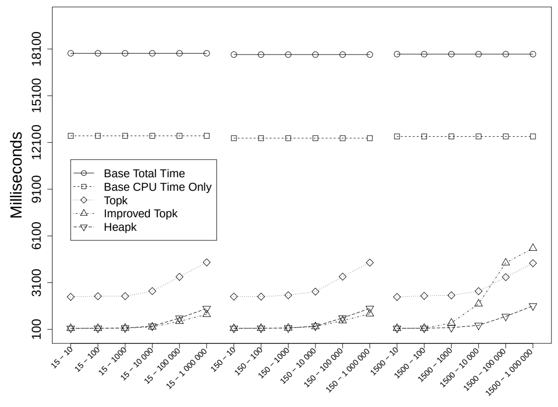

Figure 1: Efficiency comparison.

For the sets of experiments on the *topk*, improved *topk* and *heapk*, the whole index was loaded into memory, thus no I/O was involved at query evaluation time. For the base set, only the first-level dictionary was loaded into memory at start-up (The index has a two-level structure [28]).

Figure 1 shows the total evaluation times for the four sets. The performance of the base set is plotted twice, one with the total time and one with the CPU time only. As the figure shows, the base set is CPU intensive (70% of the total time).

The performance differences between the base CPU time only and the other three sets are the times taken for (1) decompression of the postings lists, (2) evaluation of the similarity scores and (3) sorting of the accumulators. First, partial decompression was used in *topk*, improved *topk* and *heapk* while the base set did not. Second, evaluation of similarity scores were converted to addition of impact values in *topk*, improved *topk* and *heapk*. Third, only the required top candidate documents were sorted in *topk*, improved *topk* and *heapk* instead of sorting all accumulators.

Figure 2 shows the MAiP measures for the four sets. For the base set, the values of upper-K has no effect since post-

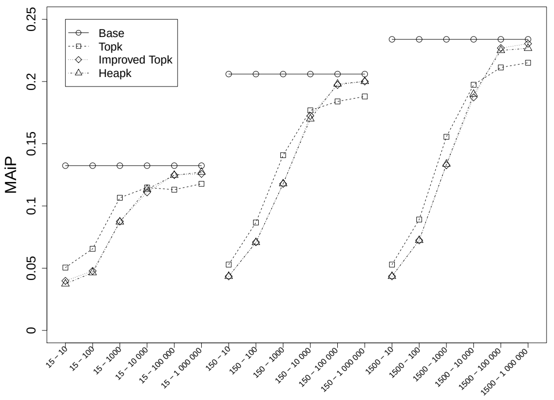

Figure 2: MAiP measures.

ings were always processed to completion. The MAiP measures are about the same for the improved *topk* and *heapk*. The subtle differences are when documents have the same similarity scores and the order of these documents can be different between the improved *topk* and *heapk*. The MAiP measures for the *topk* are different as it used term frequencies directly for similarity ranking.

3.3 Accumulator Initialisation

Since search engines are normally bottlenecked by other factors, the effect of accumulator initialisation has almost been ignored. However, after applying these optimisations, our search engine was bottlenecked by accumulator initialisation. In [13], I proposed an efficient accumulator initialisation method. It uses two static arrays. One array is used to hold all accumulators (one for each document) and the other to hold a number of flags. Every flag is associated with a particular subset of the accumulators, indicating the initialisation status for that set of accumulators. Essentially, we turn the one dimensional array of accumulators into a logical two dimensional table. The number of the flags is the same as the height of the table.

Initially, the flags are initialised to zero, indicating all accumulators having zero values. When updating an accumulator with a new value, the flag associated with that row of the accumulator is set to 1. For updating operation, the logical row of a accumulator can be obtained by a division operation of the index of the accumulator. Two possible cases can happen. If the flag is 0, it is set to 1, the associated accumulators are initialised and the new value is added to the accumulator. If the flag is 1, the new value can be simply aggregated to the accumulator.

In order to compare the static array approach with the logical two dimensional table, the experiments were conducted using the INEX 2009 Wikipedia collection [22] and the 115 Type-A (short) queries for the INEX 2009 Efficiency Track [23]. The collection was indexed with no words stopped and stemming was not used.

The results are shown in Figure 3. The width for the logical two dimensional table was chosen to be 256 accumulators. The only performance differences between these two approaches are the times taken for the accumulator initialisation, and the added overhead to locate the accumulator in ranking function. The logical two dimensional table took about zero time for the accumulator initialisation since it was very fast to initialise a small number of flags. However, the logical two dimensional table added small amount of overhead for ranking due to the extra operations required to keep track of the flag status. When comparing the total evaluation times, the logical two dimensional table outperformed the static array in all runs.

In [13] I include a formal proof as to the optimal width of the row - it is dependent on the number of terms and the lengths of the postings lists. In further work I will compare estimates of this optimal value to the current estimate we use (a whole power of two) and develop a model of the error between the two.

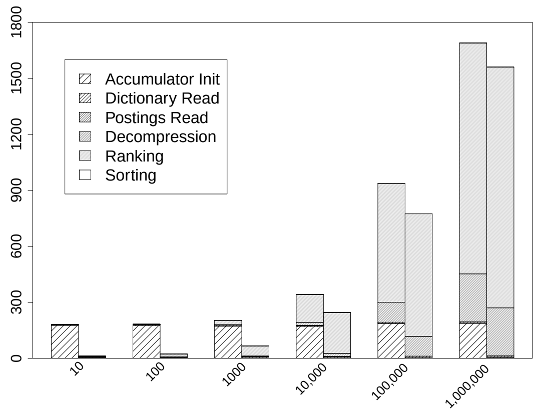

Figure 3: Comparison of difference approaches for accumulator initialisation. The first bar in each group shows the results for the static array structure while the second shows the results for the logical two dimensional table.

3.4 Dictionary Compression

I am currently working on an efficient solution for dictionary compression [28, 12]. In our search engine, the dictionary of terms is split into two parts. Terms with the same prefix are grouped together in a term block. The common prefix (only the first four characters) is stored in the first level of the dictionary and the remaining are stored in the term block in the second level (it is a 2-level B-tree). The number of terms in the block is stored at the beginning of the block. The term block also 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.

The performance of this dictionary compression algorithm has not yet been compared to those of others. To do so it will be compared to *dictionary-as-a-string* and *blocked-storage* [15], front coding [31].

4. DISCUSSION

This research investigates the bottlenecks of the search engine and proposed efficient and effective solutions to minimise or eliminate the bottlenecks; (1) Disk I/O is completely eliminated by simply loading the index file into memory. (2) Similarly scores are pre-computed at index time. (3)

The proposed *heapk* pruning algorithm is very efficient and effective. (4) Partial decompression of postings lists is deployed. (5) efficient solution for accumulator initialisation is proposed.

When the *heapk* pruning algorithm is used, the search engine can be between 70% and 94% more efficient than when no optimisation is used. When the postings lists are static pruned at 10,000 postings or fewer, the search engine is bottlenecked by the accumulator initialisation. I introduced an efficient accumulator initialisation algorithm to address this [13]. Essentially, the search engine has been optimised to be able to handle one query in sub-millisecond on average on the INEX Wikipedia collection.

A number of other pruning algorithms have also been investigated and implemented into the search engine, including the threshold pruning algorithm [9], *QUIT* and *CONTINUE* [18, 16], the filtered pruning algorithm [20, 21] and the any-time stopping algorithm [1]. In the future a comparison of these algorithms will be conducted. What effectiveness measurement should be used in order to make a fair comparison (P@n, MAP, MAiP or others)?

The *heapk* is a static pruning algorithm. One potential improvement of the algorithm is to make it dynamic. The first questions is what is wrong with static pruning? If a static pruning algorithm can be efficient and effective, why not just use a static pruning algorithm? Dynamic pruning may just add more runtime overhead.

Multi-core CPU architectures are getting popular. A research question on using such architectures is how to optimise search engines to use all available cores. One approach is to run one instance of a search engine which has multiple threads. The problem with this approach is locking, synchronisation and the complexity of managing threads in different states. Another approach is to run one instance of a search engine on each core. This approach is similarly to traditional distributed IR, but on a single machine.

The most important question is: in what future directions can this research be taken?

5. REFERENCES

[1] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. 2001.

[2] V. N. Anh and A. Moffat. Compressed inverted files with reduced decoding overheads. 1998.

[3] V. N. Anh and A. Moffat. Random access compressed inverted files. *Proc. 9th Australasian Database Conf.*, 20(2):1–12, Feb. 1998.

[4] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[5] P. Arvola, S. Geva, J. Kamps, R. Schenkel, A. Trotman, and J. Vainio. Overview of the inex 2010 ad hoc track. In *Pre-Proceedings of the INEX'10*, 2010.

[6] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR '07*, pages 183–190, 2007.

[7] J. L. Bentley and M. D. Mcilroy. Engineering a sort function, 1993.

[8] R. Blanco and A. Barreiro. Probabilistic static pruning of inverted files. *ACM Trans. Inf. Syst.*, 28(1):1–33, 2010.

[9] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. pages 97–110, 1985.

[10] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. pages 43–50, 2001.

[11] D. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *J AM SOC INFORM SCI*, 41:581–589, 1990.

[12] X.-F. Jia, D. Alexander, V. Wood, and A. Trotman. University of otago at inex 2010. In *INEX '10: Pre-Proceedings of the INEX*. ACM, 2010.

[13] X.-F. Jia, A. Trotman, and R. O'keefe. Efficient accumulator initialisation. In *ADCS '10*, 2010.

[14] X.-F. Jia, A. Trotman, R. O'Keefe, and Z. Huang. Application-specific disk I/O optimisation for a search engine. In *PDCAT '08*, pages 399–404, 2008.

[15] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[16] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[17] A. Moffat, J. Zobel, and S. T. Klein. Improved inverted file processing for large text databases. 1995.

[18] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 1994.

[19] E. S. d. Moura, C. F. d. Santos, B. D. s. d. Araujo, A. S. d. Silva, P. Calado, and M. A. Nascimento. Locality-based pruning methods for web search. *ACM Trans. Inf. Syst.*, 26(2):1–28, 2008.

[20] M. Persin. Document filtering for fast ranking. 1994.

[21] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[22] R. Schenkel, F. Suchanek, and G. Kasneci. YAWN: A semantically annotated wikipedia xml corpus. 2007.

[23] R. Schenkel and M. Theobald. Overview of the inex 2009 efficiency track. volume 6203 of *Lecture Notes in Computer Science*, pages 200–212. 2010.

[24] M. Theobald. *Efficient and Versatile Top-k Query Processing for Text, Structured, and Semistructured Data*. PhD thesis, April 2006.

[25] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. pages 242–249, 2005.

[26] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. pages 242–249, 2005.

[27] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[28] A. Trotman, X.-F. Jia, and S. Geva. Fast and effective focused retrieval. volume 6203 of *Lecture Notes in Computer Science*, pages 229–241. 2010.

[29] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. pages 987–990, 2007.

[30] H. Turtle and J. Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6):831 – 850, 1995.

[31] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann, 1999.