Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems

Stefan Büttcher and Charles L. A. Clarke
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{sbuettch, claclark}@plg.uwaterloo.ca

UW Technical Report CS-2005-31

ABSTRACT

We examine issues in the design of fully dynamic information retrieval systems with support for instantaneous document insertions and deletions. We present one such system and discuss some of the major design decisions. These decisions affect both the indexing and the query processing efficiency of our system and thus represent genuine trade-offs between indexing and query processing performance.

Two aspects of the retrieval system – fast, incremental updates and garbage collection for delayed document deletions – are discussed in detail, with a focus on the respective trade-offs. Depending on the relative number of queries and update operations, different strategies lead to optimal overall performance. Special attention is given to a particular case of dynamic search systems – desktop and file system search. As one of the main results of this paper, we demonstrate how security mechanisms necessary for multi-user support can be extended to realize efficient document deletions.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Textual databases; H.3.4 [**Systems and Software**]: Performance evaluation

General Terms

Experimentation, Performance

Keywords

Information Retrieval, Dynamic Indexing, File System Search, Indexing Performance, Query Performance

1. INTRODUCTION

Indexing performance and query processing performance are two closely related aspects of information retrieval. However, when optimization techniques for either indexing performance or query processing performance are discussed, the two aspects are usually considered separate, unconnected entities, and optimization techniques for either aspect are examined independently. This does not reflect reality, as many query optimization techniques necessitate additional work at indexing time:

- frequency-ordered lists [8] can be used to speed up query processing but require additional work at indexing time because the postings have to be sorted by within-document frequency;
- word stemming [9], which is usually applied at indexing time, can also be performed at query time, allowing for either faster indexing or faster query processing.

While the subtleties of these trade-offs do not play a great role in traditional retrieval systems that work on static collections for which a large number of queries has to be processed (because these systems obviously require maximum performance at query time), they are very important in dynamic information retrieval systems for which the underlying text collection is continuously changing and the number of queries to be processed may vary greatly.

In this paper, we discuss several opportunities for indexing performance vs. query processing performance trade-offs. Because this is done in the context of a dynamic information retrieval system, *indexing* actually refers to *index maintenance* and includes two types of index update operations: document insertions and document deletions.

In the next section, we give an overview of related work. Section 3 gives an introduction to our indexing system, the Wumpus[1] file system search engine. It describes the general layout of the system, the security mechanisms necessary for use in multi-user environments, and the internal index structure. Each of the following two sections discusses one particular component of the search system involving an essential indexing vs. query time trade-off:

- sub-index merging strategies, which are necessary for incremental updates (section 4);
- index garbage collection, which is used to support delayed index updates after document deletions and is closely related to the security subsystem (section 5).

[1] http://www.wumpus-search.org/

2. RELATED WORK

Methods to support dynamically changing text collections can be divided into two categories: Support for document insertions and support for document deletions.

Techniques to support document insertions into an existing index have been studied by many researchers over the last decade. Most of them follow the same basic scheme. They maintain both an on-disk and an in-memory index. Postings for new documents are accumulated in main memory until it is exhausted, and then the data in memory are somehow combined with the on-disk index.

Tomasic et al. [12] present an in-place update scheme for inverted files, based on a distinction between short lists and long lists. They also discuss how different allocation strategies for the long lists affects index maintenance and query processing performance. Lester et al. [7] give an evaluation of three different methods to combine the in-memory information with the on-disk data. Kabra et al. [6] present a hybrid IR/DB system with delayed update operations through in-memory buffers. All of these solutions have in common that the entire on-disk index has to be read (or written) every time main memory is exhausted, which causes performance problems for large collections. We show how the number of disk operations can be significantly reduced, at minimal cost for query performance.

In contrast to the case of document insertions, a thorough evaluation of techniques for document deletions is not available. Chiueh and Huang [2] present a lazy invalidation approach that keeps an in-memory list of all deleted documents and performs a postprocessing step for every query, taking the contents of that list into account. The approach to document deletions presented in this paper is similar to theirs, but more general, and is not done as a postprocessing step, but integrated into the actual query processing.

None of this related work provides a general discussion of how different index maintenance strategies affect query processing performance and how this implies opportunities for indexing versus query processing performance trade-offs.

3. THE WUMPUS SEARCH SYSTEM

The retrieval system described in this paper is the Wumpus file system search engine. Wumpus is similar to other file system search engines, such as Google Desktop Search[2], Apple Spotlight[3], or Beagle[4]. Unlike most desktop search systems (except Spotlight), it is a true multi-user search system; only a single index is used for all files in the file system, and security restrictions are applied at query time in order to guarantee that the query results are consistent with all file permissions.

In addition to mere file system search cabilities, Wumpus supports several standard information retrieval methods, such as Okapi BM25 [10] and MultiText QAP [4]. The search system is based on the GCL query language and the implementation framework proposed by Clarke et al. [3].

In the remainder of this section, we give a short introduction to the specific requirements of file system search, explain how our system deals with the individual aspects of file system search, and describe the basic experimental setup as well as our performance evaluation methodology.

[2]http://desktop.google.com/
[3]http://www.apple.com/macosx/features/spotlight/
[4]http://www.gnome.org/projects/beagle/

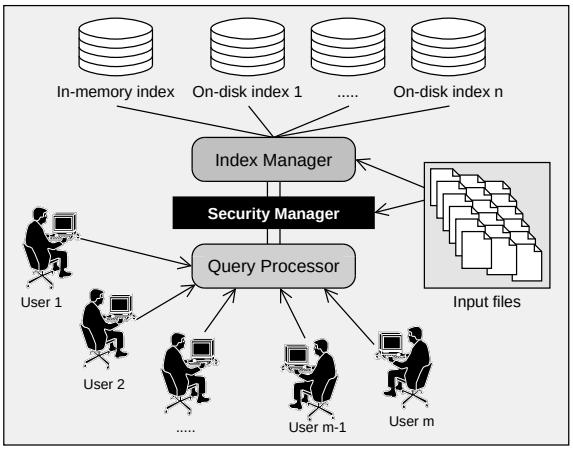


Figure 1: General Layout of the Wumpus search system. Wumpus maintains an in-memory index for recent updates and several on-disk indices for persistent storage. User-specific security restrictions are applied to all data that are transferred from the index manager to the query processor.

3.1 File System Search

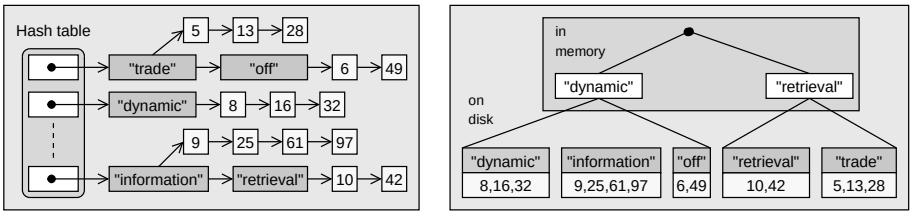
File system search is different from the traditional information retrieval task. The search engine not only has to deal with a large heterogeneous document collection, but a file system is also a truly dynamic environment: files are constantly created, modified, and deleted. The expected number of index update operations is much greater than the number of queries to be processed. Using Wumpus, one of the authors counted more than 4,000 index update operations (document insertions and deletions) on his laptop computer during a typical work day.

Furthermore, when an e-mail arrives, or a new file is created, the user expects the search system to reflect this change immediately. Delays greater than a few seconds are not acceptable. This, together with the great number of update operations that have to be performed, suggest that indexing performance plays a much greater role than query processing performance in this particular domain. Wumpus supports fast instantaneous updates (i.e., changes to the file system are reflected by the search system within fractions of a second). The indexing techniques used to achieve this are presented in section 3.2.

In addition to being a dynamic environment, file system search is a multi-user application. In order to avoid wasting disk space due to indexing the same file many times, a single index has to be used for all users in the system. Special care has then to be taken so as to guarantee file system security. Wumpus' built-in security mechanisms guarantee that query results only depend on the content of files that are readable by the user who submitted the query. The security subsystem, which is also used to support document deletions, is briefly discussed in section 3.5.

3.2 Indexing (Document Insertions)

Wumpus uses inverted files [15] as its main index structure. All posting lists contain fully positional information (i.e., the exact locations of all occurrences of a term). Positional indexing is, for instance, necessary for phrase queries and term proximity ranking [5], both of which are supported



(a) In-memory index with hash table and linked lists.

(b) On-disk index with 2-level search tree.

Figure 2: Basic structure of in-memory index and on-disk index. Vocabulary terms in memory are arranged in a hash table. Terms on disk are sorted in lexicographical order, allowing for efficient prefix search operations.

by Wumpus. However, the results presented in this paper, can easily be applied to document-centered search systems that do not maintain a fully positional index.

The actual indexing is a one-pass process. It can be described as *in-memory inversion* with ad-hoc *text-based partitioning* (cf. Witten et al. [13, pp. 245-253]). When a new document is added to the index, tokens are read from the input file, and an inverted index is built in memory until the memory consumption reaches a predefined threshold. At that point, an on-disk inverted file is created from the postings in memory, the in-memory index is deleted, and the system continues to read further tokens from the input file. Wumpus is able to maintain multiple on-disk indices at the same time. Different strategies for merging these sub-indices are discussed in section 4.

In the in-memory index, a hash table with move-to-front heuristics [14] is used to efficiently search for existing terms and add new terms to the vocabulary. All postings are stored in compressed form (byte-aligned gap compression [11]), using linked lists to allow for flexibility and efficient insertion of new postings. This is shown in Figure 2(a).

When new data is added to the in-memory index, postings are immediately assigned to vocabulary terms using the hash table. After a posting has been added to a posting list, it can be accessed by the query processor without any further delay. Thus, instantaneous updates, which are needed for real-time file system search, are supported by this data structure.

In the on-disk index, as in memory, all postings are stored in compressed form. Terms are sorted in lexicographical order, which minimizes the number of disk seeks necessary for prefix searches and query-time stemming (a special kind of prefix search). Terms and their postings are kept together, which again decreases the number of disk seeks at query time. For every on-disk index that is part of the indexing system's main index, some meta-information is kept in memory. Conceptually, this meta-information can be thought of as a B-tree of height 2 that is used to reduce the search space when postings have to be fetched from the on-disk index (shown in Figure 2(b)).

3.3 Multiple Sub-Indices

Wumpus is able to maintain multiple on-disk indices at the same time. An important property of these sub-indices is that they form an *ordered partitioning* of the index address space, i.e. the postings in sub-index $\mathcal{I}_k$ come always before those in sub-index $\mathcal{I}_{k+1}$. This property is preserved by all index maintenance operations (sub-index merging and garbage collection). It has two important implications:

- during query processing, the posting list for a given term can be created by concatenating the sub-lists retrieved from the individual sub-indices;
- during a sub-index merge process, postings do not have to be decompressed, but may simply be copied in their compressed form, avoiding expensive within-list merging and thus allowing for very fast sub-index merge operations.

3.4 Query Processing

Wumpus' query processor is based the GCL query language and its shortest-substring paradigm [3]. GCL supports a variety of operators that can be used to impose structural constraints on query results. In the context of this paper, we only need two of them:

- $A \cdots B$ generates all intervals $[x, y]$ (called *index extents*), such that A occurs at index address x and B occurs at index address y.
- $E_1 \lhd E_2$ generates all index extents that satisfy the GCL expression E_1 and are *contained* in an extent that satisfies E_2.

The exact sequence of operations performed by the system when a query is being processed, of course, depends on the scoring function used to rank the documents in the collection. In general, query processing consists of four steps:

1. Create a list of all index extents that are to be scored. Traditionally, this is the list of documents in the collection, but Wumpus supports arbitrary GCL expressions. For the text collections used in our experiments, document extents are generated by the GCL expression `<doc>` $\cdots$ `</doc>`.
2. Fetch all postings for all query terms from the on-disk indices and the in-memory index.
3. Compute the score for every extent from Step 1, using the information found in the posting lists collected in Step 2.
4. Report the top n extents, along with their scores, to the user.

Since all on-disk indices and the in-memory index form an ordered partitioning of the index address space, the posting list for a given term can be created by concatenating the lists found in the individual sub-indices. The number of disk seeks necessary to fetch all postings for one term is therefore linear in the number of sub-indices, which is why at some point it is advisable to merge all sub-indices in order to increase query processing performance.

3.5 Security Restrictions and Document Deletions

As pointed out in section 3.1, special security mechanisms are needed for a file system search engine if all users in the system share one global index. Wumpus' security manager keeps track of file system meta-information, such as directory structure, file ownership, and file permissions. Whenever a user U submits a query, these meta-data are used to generate a list V_U of index extents that correspond to all files which may be searched by that user. While the query is being processed, every time a posting P list is fetched from the index, V_U is used to create a new list

$$P_U \equiv (P \lhd V_U),$$

which consists of all parts of P that refer to files that may be searched by U. The query processor never accesses P directly, but always through P_U. This guarantees that all query results are consistent with the user's view of the file system and that all security permissions are respected. In our implementation, P_U allows lazy evaluation so that the security restrictions only have to be applied to those parts of the original list P that are needed to process the query.

This security mechanism can be used to delay applying update operations to the index. When a file is deleted, the postings for that file are not immediately removed from the index. Instead, only the index extent corresponding to the file is removed from the security manager's database. This is equivalent to marking the file as "cannot be read by anybody". The next time a query is processed, V_U does not contain the extent in question any more, and the query processor behaves as if the postings in the interval had been physically removed from the index. Since the index is fully positional, partial document deletions, such as file truncations, are also supported.

Our invalidation scheme has certain advantages over the scheme presented by Chiueh and Huang [2]. Their postprocessing approach uses outdated (and therefore slightly wrong) collection statistics to compute query term weights, and it is unable to restrict the number of result candidates during query processing: Even if the user is only interested in the top 20 documents, scores for all documents examined have to be kept because arbitrarily many documents may be removed from the result set in the postprocessing step. Both problems are solved by integrating the invalidation into the actual query processing.

Although the security subsystem can be used to delay hard index updates, postings that belong to deleted files have to be removed from the index at some point. Garbage collection strategies are discussed in section 5.

3.6 Experimental Setup

For our experiments, we used two text collections, having different size and different collection characteristics:

- TREC disks 4 and 5 (without Congressional Record), referred to as TREC4+5-CR; our query set were 100 topics from the TREC 2003 Robust track.
- the 2004 TREC Genomics collection (subset of the PubMed Medline corpus), referred to as TREC-Genomics; as queries, we used the 50 ad hoc topics from the 2004 Genomics track.

For both collections, general corpus statistics and basic performance figures are given by Table 1. The performance values represent systems configured for different levels of trade-off between indexing and query processing performance.

Table 1: Collection statistics and performance figures for different trade-off levels. Indexing a static text collection: (1) with final sub-index merging, (2) without sub-index merging, (A) stemming at indexing time, (B) stemming at query time.

		TREC4+5-CR	TREC-Genomics
	Collection size	1,904 MB	14,332 MB
	# documents	528,155	4,591,008
	# tokens	$3.23 \cdot 10^8$	$2.05 \cdot 10^9$
	# distinct terms	$1.16 \cdot 10^6$	$7.87 \cdot 10^6$
(1A)	Index size	550 MB	3.613 MB
	Indexing time	3m08s	21m42s
	Qry. performance	4.6 queries/s	0.56 queries/s
(1B)	Index size	573 MB	3,696 MB
	Indexing time	2m35s	19m36s
	Qry. performance	3.6 queries/s	0.44 queries/s
(2A)	Index size	569 MB	4,004 MB
	Indexing time	2m34s	16m09s
	Qry. performance	3.8 queries/s	0.33 queries/s
(2B)	Index size	595 MB	4,117 MB
	Indexing time	1m59s	13m56s
	Qry. performance	2.7 queries/s	0.26 queries/s

In order to be able discuss indexing time vs. query time trade-offs with respect to the amount of work the indexing subsystem and the query processor have to do, all experiments discussed in the following sections were conducted with varying degrees of system dynamics. The dynamics of a search system can be expressed by the number of update operations (document insertions or deletions) per search query, denoted as $\mathcal{D}_Q^U$. A system with $\mathcal{D}_Q^U = 0$ processes queries for a static text collection. $\mathcal{D}_Q^U = \infty$, on the other hand, describes a system which only performs update operations and never processes any queries. A dynamic retrieval system has a $\mathcal{D}_Q^U$ value somewhere between these two extremes. For file system search, due to the high frequency of file changes on a typical computer system, $\mathcal{D}_Q^U \in [10^2, 10^5]$ can be expected. In addition to $\mathcal{D}_Q^U$, we also use $\mathcal{D}_U^Q$, the number of queries per update operation.

In order to decrease the number of disk seeks necessary to read the input files and thus be able to conduct all experiments in a timely manner, documents were grouped into files containing 100 documents each (45,911 files for TREC-Genomics). This uniformly decreases the running time for all experiments and thus does not affect our conclusions.

All experiments were conducted on a PC based on an AMD Athlon64 3500+ processor with 2 GB main memory and a 7200-rpm SATA hard drive. The operating system was GNU/Linux.

4. SUB-INDEX MERGING

Many retrieval systems build an index for a given text collection by partitioning the collection into smaller parts, whose size is determined by the amount of available memory. Postings are accumulated in memory until main memory is exhausted. At that point, all in-memory postings are sorted and written to disk, forming a new sub-index. When the entire collection has been indexed, all sub-indices that have

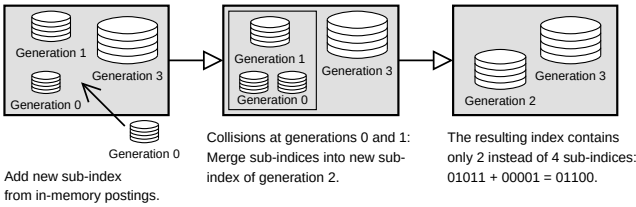


Figure 3: *Logarithmic Merge* by example. A new sub-index is added from in-memory postings. Sub-indices of the same generation n are merged into into a new sub-index of generation $n+1$.

been created during the indexing process are merged into one big inverted file that represents the whole collection.

While this technique is perfectly reasonable for static collections, it cannot be used for dynamic collections, because the index is continuously updated and the point at which the whole collection has been indexed is never reached. However, the indices should be merged at some point, since a large number of sub-indices significantly decreases query processing performance due to disk seek latency. Therefore, we have to decide when it is acceptable to have multiple sub-indices and when it is sensible to merge them. We present three different merge strategies and evaluate their performance under different conditions.

Strategy 1: Immediate Merge

The first merge strategy has been proposed by Lester et al. [7]. They analyze three different techniques to deal with dynamically growing text collections. Out of these techniques, a strategy called *Re-Merge* exhibits the best performance. The indexing system maintains one on-disk and one in-memory index. As soon as main memory is full, Re-Merge sorts the in-memory postings and merges them with the existing on-disk index, creating a new on-disk index that contains all postings gathered so far. We call it Immediate Merge because in-memory postings are immediately merged with the on-disk index when they are written to disk.

The advantage of this strategy is that, in order to fetch a posting list, in most cases only a single disk seek is necessary, since there is only one sub-index from which postings have to be retrieved. The disadvantage is that for every merge operation the entire index has to be scanned. Therefore, the number of disk operations necessary to create the index is

$$D_{index}(C, M) \in \Theta\left(\frac{C^2}{M}\right),$$

where C is the size of the text collection and M is the amount of available main memory. This is devastating for very large text collections and for systems with little available main memory (halving the available main memory doubles the number of disk operations).

Strategy 2: No Merge

The second strategy does not perform any merge operations. When memory is full, postings are sorted and written to disk, creating a new on-disk sub-index. On-disk indices are never merged. When the posting list for a given term has to be retrieved from the index, sub-lists are fetched from all sub-indices. Since the sub-indices are ordered, the term's posting list can be created by concatenating the sub-lists.

The advantage of No Merge is its high indexing performance – each sub-index is accessed exactly once: when it is created from the in-memory postings. This implies

$$D_{index}(C, M) \in \Theta(C).$$

The disadvantage is that, in order to fetch a term's posting list,

$$D_{fetch}(C, M) \in \Omega\left(\frac{C}{M}\right)$$

disk operations (and in particular $\Omega(\frac{C}{M})$ disk seeks) are necessary, causing a low query processing performance for collections that are much larger than the available main memory. This behavior is shown in Figure 4.

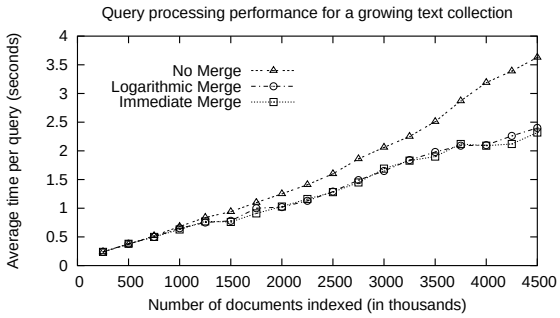


Figure 4: Development of query performance for a growing index. Logarithmic Merge and Immediate Merge stay very close to each other. No Merge gets worse as the number of sub-indices increases.

Strategy 3: Logarithmic Merge

The two strategies described so far (Immediate Merge and No Merge) represent the two extremes. The third strategy is a compromise: A newly created on-disk sub-index is sometimes merged with an existing one, but not always.

In order to determine when to merge two sub-indices, we introduce a new concept: index *generation*. An on-disk index that was created directly from in-memory postings is of generation 0. An index that was created by merging several other indices is of generation $g+1$, where g is the highest generation of any of the indices involved in the merging process. If, after a new on-disk index has been created, there are two indices of the same generation, they are merged. This is repeated until there are no more such collisions.

We call this strategy Logarithmic Merge after the *logarithmic method* by Bentley and Saxe [1]. It is similar to the incremental indexing scheme used by Lucene[5].

When Logarithmic Merge is used, the vector

$$\overrightarrow{a} = (a_0, a_1, \ldots),\ a_i := \begin{cases} 0\text{: no sub-index of generation } i \\ 1\text{: otherwise} \end{cases}$$

behaves like a binary number that is increased by 1 every time a new on-disk index is created.

It follows immediately that at a given point in time the number of sub-indices is bounded by

$$O\left(\log\left(\frac{C}{M}\right)\right),$$

where C and M are again the size of the collection and the available main memory, respectively. This is much better

[5]http://lucene.apache.org/

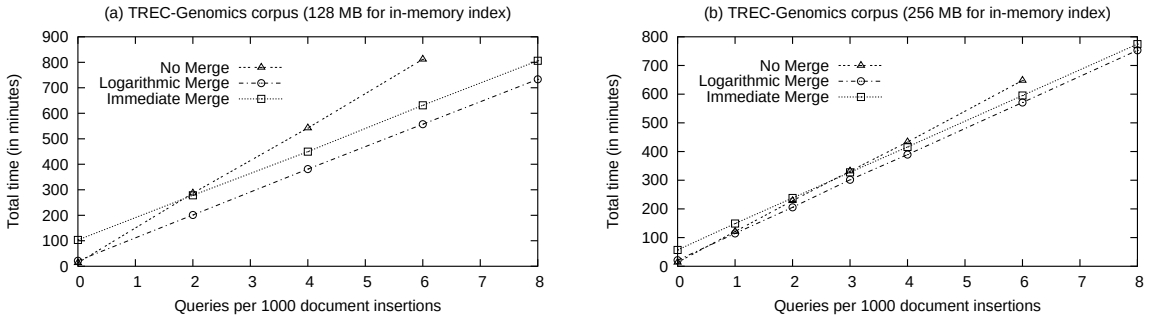


Figure 5: Comparison of sub-index merging strategies for growing text collections: (a) TREC-Genomics with 128 MB main memory, (b) TREC-Genomics with 256 MB main memory. Every data point represents one experiment (indexing the whole collection and processing relevance queries in parallel). The time given is the total time for indexing the collection and processing all queries. $\mathcal{D}_U^Q$ is varied between 0 and 10^{-2}.

Table 2: Exact times for the experimental results shown in Figure 5(b). Extrapolated point of equality for Log. Merge and Immediate Merge: $\mathcal{D}_U^Q \approx 0.024$.

$\mathcal{D}_U^Q$	No Merge	Log. Merge	Imm. Merge
0.000	13m40s	21m08s	57m01s
0.002	228m45s	205m17s	238m05s
0.004	433m29s	389m22s	416m05s
0.006	647m41s	571m08s	595m11s
0.008		752m54s	774m26s
0.024		$\approx$ 2207m	$\approx$ 2208m

than for the No Merge strategy and implies increased query processing performance. The number of disk operations necessary to build the index is

$$D_{index}(C, M) \in \Theta\left(C \cdot \log\left(\frac{C}{M}\right)\right),$$

which is much smaller than for Immediate Merge.

The number of disk operations can be reduced by a constant factor by anticipating future merge operations and starting a multi-way merge process whenever merging two indices would cause a new collision and thus a new merge process. This process is shown in Figure 3.

Results

The experiments run to evaluate the different merging strategies under different query/update conditions involve a monotonically growing document collection. For the TREC-Genomics collection, we ran two series of experiments, with 128 and 256 MB for the in-memory index, respectively. Experiments for TREC4+5-CR are not shown here, because the collection is too small to allow any conclusions.

Starting from an empty index, the system concurrently adds documents to the index and processes relevance queries until the whole collection has been indexed. The number of relevance queries per update operation (document insertion) is varied in order to examine the effect that each of the merge strategies has on both indexing performance and query processing performance. The total time needed by the system to finish the job, i.e. index the entire collection and process all queries, is measured. Results for both series of experiments are shown in Figure 5. In addition, the exact numbers for the 256 MB series are given by Table 2.

As expected, No Merge is the best strategy under a very light query load ($\mathcal{D}_U^Q < 5 \cdot 10^{-4}$). As the number of queries per update operation increases, No Merge becomes worse and worse, due to the high number of sub-indices that have to be accessed every time a query is processed.

Logarithmic Merge shows excellent indexing performance; it is only 35% slower than No Merge for $\mathcal{D}_U^Q = 0$ and 256 MB of memory. As $\mathcal{D}_U^Q$ increases, Logarithmic Merge soon overtakes No Merge and becomes the best strategy. In all our experiments, overall performance for Logarithmic Merge was better than for Immediate Merge. However, by extrapolating from the experimental results, we can predict that Immediate Merge becomes the optimal strategy for $\mathcal{D}_U^Q > 0.024$ (24 queries per 1,000 update operations). But even then, Logarithmic Merge stays very close. Its asymptotical performance ($\mathcal{D}_U^Q \to \infty$) is less than 3% lower than that of Immediate Merge on our test collection.

It also has to be pointed out that Logarithmic Merge is very robust against decreasing the available memory. While going from 256 MB down to 128 MB significantly decreases No Merge's query performance and doubles indexing time with Immediate Merge, both indexing and query performance remain almost constant when Logarithmic Merge is used as the sub-index merging strategy.

Discussion

We have shown that, depending on how dynamic the underlying text collection is, different sub-index merging strategies have to be chosen in order to achieve optimal overall performance for the search system. The No Merge strategy, which offers the best indexing performance, should only be used if the number of queries to be processed is extremely small. For most applications, Logarithmic Merge is the best choice, since it scales very well and can therefore be used to index very large text collections, where Immediate Merge causes problems because of its $\Theta(C^2)$ indexing complexity.

5. GARBAGE COLLECTION

In the previous section, we have discussed sub-index merging strategies for continuously growing document collections. In a truly dynamic environment, however, the collection does not grow monotonically. Update operations may be either document insertions or deletions.

Postings that are added during an insert operation can

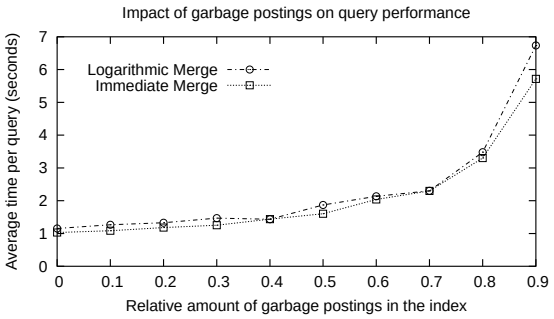


Figure 6: Experimental results for a sub-collection of TREC-Genomics, containing 2.3 million active and a varying number of deleted documents. Query processing performance drops as the number of garbage postings is increased. Logarithmic Merge stays very close to Immediate Merge.

simply be appended to existing posting lists (assuming that newly added documents reside at the very end of the index address space). Therefore, it is possible to create a new on-disk sub-index from the in-memory postings without having to access the other on-disk sub-indices – a great reduction in the number of disk accesses. This is not true for document deletions, as they can potentially affect the entire address space. Hence, their impact is not limited to a single sub-index (a small part of the index address space), which makes deletions more difficult to deal with than insertions.

Wumpus' security mechanism helps deal with document deletions. Every user can only access the part of a posting list that lies within files that may be read by that user, since the respective file permissions are automatically applied every time a query is processed (cf. section 3). This mechanism can be used to support document deletions: When a document is deleted from the collection, the file associated with that document is marked as "cannot be read by anybody". After that, all postings lying within the file are ignored by the query processor.

However, filtering posting lists during query processing can only be one part of the solution. At some point, postings that belong to deleted documents have to be removed, because the time it takes to process a query is essentially linear in the total number of postings in the index (as shown in Figure 6). We need a garbage collection strategy.

Threshold-based Garbage Collection

A relatively simple strategy is to keep track of the relative number of postings in the index that belong to documents that have been deleted:

$$r = \frac{\#\text{deleted postings}}{\#\text{postings}}.$$

As soon as this number exceeds a predefined threshold (i.e., $r > \rho$, for some $\rho \in (0, 1]$), the garbage collector is started, and all superfluous postings are removed from the index.

This is the basic garbage collection strategy used by Wumpus. Different values for ρ affect indexing performance and query processing performance in different ways: If the garbage collector is run after every single document deletion ($\rho \approx 0$), this guarantees maximum query performance – at the cost of very bad indexing performance. If, on the other hand, the garbage collector is never run ($\rho = 1$), indexing performance is maximal, but the query processor spends a great amount of time fetching and decompressing postings that are irrelevant to the query because they belong to deleted documents.

Since during garbage collection all sub-indexes have to be read completely anyway, Wumpus automatically merges all sub-indexes into one big index every time the garbage collector is run.

On-the-Fly Garbage Collection

The threshold-based garbage collection strategy descibed above has the disadvantage that it is completely independent of the sub-index merging strategy employed and therefore causes additional disk access operations that could have been avoided if the garbage collector had been integrated into the sub-index merging process. We call this integration *on-the-fly garbage collection*: Every time two or more sub-indices are merged into a bigger sub-index, the garbage collector can be used to filter the postings that are read from the input indices and only write those postings to the output index that belong to documents that have not been deleted. On-the-fly garbage collection is employed in addition to the simple threshold-based approach.

Since garbage collection requires the decompression of posting lists, it causes a notable reduction of sub-index merging performance (ordinary sub-index merging does not require the decompression of postings – see section 3.3). It is therefore not desirable to run it for *every* merge operation. Instead, we define a new threshold value ρ'; the garbage collector is integrated into a merge process if

$$\frac{\sum_{i=1}^{k} \#\text{deleted postings in index } \mathcal{I}_i}{\sum_{i=1}^{k} \#\text{postings in index } \mathcal{I}_i} > \rho',$$

where $\mathcal{I}_1 \ldots \mathcal{I}_k$ are the input indices of the merge operation. Reasonable choices for ρ' are in the interval $(0, \rho]$.

Experimental Results

We conducted several experiments to compare different garbage collection threshold values ρ and study how the integration of the garbage collector into sub-index merging can increase overall system performance. For each text collection (TREC4+5-CR and TREC-Genomics), one series of experiments was conducted. For all experiments, we limited the size of the in-memory index to 256 MB and used Logarithmic Merge as the sub-index merging strategy.

Starting from an index that contains 50% of the documents in the text collection (≈2.3 million documents for TREC-Genomics and ≈260,000 documents for TREC4+5-CR), documents are added and removed in a random fashion, with equal probabilities for insertions and deletions. In contrast to the experiments described in the previous section, we had the system process a fixed number of queries (5,000) and varied $\mathcal{D}_Q^U$, the number of update operations between two queries, between 0 and 10,000.

The results visualized in Figure 7 show that for most update loads $\rho = 0.50$ is a safe choice, leading to an acceptable query performance. Only on TREC-Genomics, it is outperformed by $\rho = 0.25$ for $0 \leq \mathcal{D}_Q^U \leq 5000$. This means that the additional work caused by eager garbage collection is usually not rewarded by superior query performance, even for low update loads. For both collection, the integration of on-the-fly garbage collection into the merge process improves the system's performance only marginally, indicating

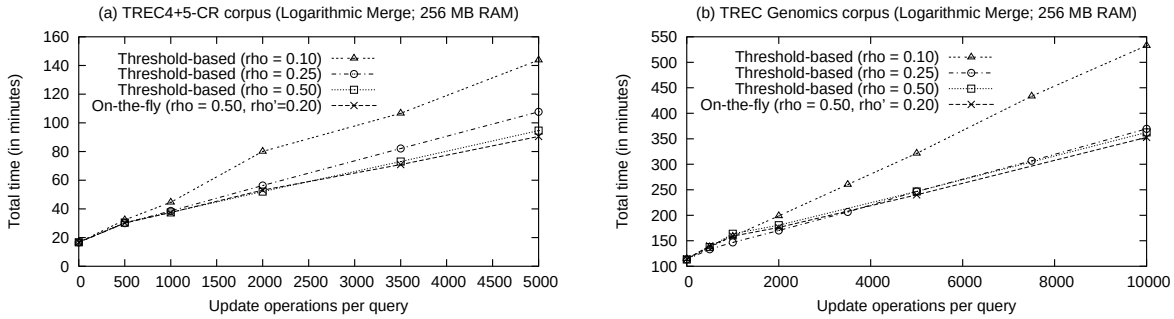


Figure 7: Garbage collection strategies for fully dynamic text collections. (a) TREC4+5-CR. (b) TREC-Genomics. The number of update operations per query is varied between 0 and 10,000. The total time needed to process 5,000 queries and concurrently perform the update operations is given.

that the relatively simple threshold-based method is in fact not as bad as we had thought and is suited very well for most scenarios.

6. CONCLUSION AND FUTURE WORK

We have presented a fully dynamic information retrieval system, supporting document insertions and deletions. Our contribution is two-fold: We have pointed out different ways to realize index update operations and examined their respective effect on both indexing and query processing performance, leading to a discussion of the associated indexing vs. query processing performance trade-offs. We have shown that for a broad range of relative query and update loads, including those typical for desktop and file system search, a combination of logarithmic sub-index merging and threshold-based garbage collection with a garbage threshold around 50% is an appropriate way to address index updates. Only under extreme conditions (very large or very small number of updates per query), other configurations exhibit better performance.

As one of the main results of this paper, we have demonstrated how security mechanisms necessary for multi-user support can easily be extended to support efficient document deletions with delayed index garbage collection.

Using $\mathcal{D}_Q^U$, the number of index updates per query, to describe how dynamic a text collection is, is a rather ad-hoc approach. We chose it because of its immediate perspicuity. Other measures, such as the relative number of documents or the relative number of postings changed between two consecutive queries, might give a better characterization of the system. An experimental evaluation using a large number of different corpora will be necessary to find a good measure for the degree of a retrieval system's dynamics.

7. REFERENCES

[1] J. L. Bentley and J. B. Saxe. Decomposable Searching Problems I: Static-to-Dynamic Transformations. *Journal of Algorithms*, 1(4):301–358, 1980.

[2] T. Chiueh and L. Huang. Efficient Real-Time Index Updates in Text Retrieval Systems. Technical report, Stony Brook, New York, USA, August 1998.

[3] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal*, 38(1):43–56, 1995.

[4] C. L. A. Clarke, G. V. Cormack, and T. R. Lynam. Exploiting Redundancy in Question Answering. In *Proceedings of the 24th Annual ACM SIGIR Conference*, pages 358–276, November 2001.

[5] C. L. A. Clarke, G. V. Cormack, and E. A. Tudhope. Relevance Ranking for One to Three Term Queries. *Inf. Proc. and Management*, 36(2):291–311, 2000.

[6] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIC Engine: A Hybrid IRDB System. Technical report, Madison, Wisconsin, USA, 2002.

[7] N. Lester, J. Zobel, and H. E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *CRPIT '26: Proceedings of the 27th Conference on Australasian Computer Science*, pages 15–23, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[8] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered Document Retrieval with Frequency-Sorted Indexes. *Journal of the American Society for Information Sciences*, 47(10):749–764, 1996.

[9] M. F. Porter. An Algorithm for Suffix Stripping. *Readings in Inf. Retr.*, 14(3):130–137, 1980.

[10] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference*, November 1994.

[11] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th Annual ACM SIGIR Conference*, pages 222–229, 2002.

[12] A. Tomasic, H. García-Molina, and K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD Conference*, pages 289–300, New York, NY, USA, 1994. ACM Press.

[13] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[14] J. Zobel, S. Heinz, and H. E. Williams. In-Memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letters*, 80(6), 2001.

[15] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted Files versus Signature Files for Text Indexing. *ACM Trans. on Database Systems*, 23(4):453–490, 1998.