Available online at www.sciencedirect.com
SCIENCE DIRECT®

Information Processing and Management 42 (2006) 916–933

INFORMATION PROCESSING & MANAGEMENT

www.elsevier.com/locate/infoproman

Efficient online index maintenance for contiguous inverted lists ☆

Nicholas Lester [a,*], Justin Zobel [a], Hugh Williams [b]

[a] *School of Computer Science and Information Technology, RMIT University, Victoria 3001, Australia*
[b] *MSN Search, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, United States*

Received 14 April 2005; accepted 26 September 2005
Available online 8 November 2005

Abstract

Search engines and other text retrieval systems use high-performance inverted indexes to provide efficient text query evaluation. Algorithms for fast query evaluation and index construction are well-known, but relatively little has been published concerning update. In this paper, we experimentally evaluate the two main alternative strategies for index maintenance in the presence of insertions, with the constraint that inverted lists remain contiguous on disk for fast query evaluation. The in-place and re-merge strategies are benchmarked against the baseline of a complete re-build. Our experiments with large volumes of web data show that re-merge is the fastest approach if large buffers are available, but that even a simple implementation of in-place update is suitable when the rate of insertion is low or memory buffer size is limited. We also show that with careful design of aspects of implementation such as free-space management, in-place update can be improved by around an order of magnitude over a naïve implementation.
© 2005 Published by Elsevier Ltd.

Keywords: Text indexing; Search engines; Index construction; Index update

1. Introduction

Search engines are one of the most widely used computer technologies. They are employed on scales ranging from the help systems embedded in desktop software up to search in digital libraries, and ranging from search on local intranets of a few thousand web pages up to search on collections of billions of pages spanning the entire Web. Even in the largest such collections, typical queries can be resolved in under a second on standard hardware, by use of inverted indexes designed specifically for text search.

Inverted indexes are used in all practical text retrieval systems and have two main components (Witten, Moffat, & Bell, 1999). First is a vocabulary of searchable terms; in most text search engines, terms are words.

☆ This paper incorporates and extends material from "In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems", N. Lester, J. Zobel, and H.E. Williams, Australasian Computer Science Conference, Dunedin, New Zealand, 2004.
* Corresponding author.
E-mail addresses: nml@cs.rmit.edu.au (N. Lester), jz@cs.rmit.edu.au (J. Zobel), hughw@microsoft.com (H. Williams).

0306-4573/$ - see front matter © 2005 Published by Elsevier Ltd.
doi:10.1016/j.ipm.2005.09.005

Second, for each term, is a postings list of locations of that term in the collection. The postings list may contain only coarse-grain information—such as the numbers of documents in the collection—or fine-grain detail, such as document numbers, paragraph numbers, and word positions. In practice, the indexes of typical web search engines store document identifiers and word positions.

The efficiency of query evaluation using inverted indexes has improved dramatically over the last 15 years (Scholer, Williams, Yiannis, & Zobel, 2002; Witten et al., 1999; Zobel, Moffat, & Ramamohanarao, 1998), through innovations such as application of compression and use of on-the-fly pruning. These advances have been complemented by new methods for building indexes (Heinz & Zobel, 2003; Witten et al., 1999) that on typical 2001 hardware allow creation of text indexes at a rate of around 8 GB of text per hour, that is, a gigabyte every 8 min. In our experiments, on 2004 desktop hardware, similar methods index around a gigabyte every 2 min—close to a thousand full-text documents per second—for data volumes ranging from 10 GB to over 400 GB.

These indexing speeds are for the task of bulk load only, that is, for creation of a complete index on a collection of documents. In contrast, the problem of efficient maintenance of inverted indexes has had relatively little attention. Indeed, even some commercial systems have no update facility: instead, the index is rebuilt each time new documents are available. Such a *re-build* approach would be prohibitively expensive if invoked at each new document, but if new documents are buffered then overall costs may well be acceptable. A re-build strategy can be effective for small databases with infrequent updates; an example is an corporate intranet, which may well be less than a gigabyte in size and is only re-indexed when the intranet is crawled, an operation that takes far longer than indexing. Thus, re-build provides a reasonable baseline against which to measure update strategies.

Often, however—even for small applications—the problem of update is significant. In some cases, documents arrive at a high rate, and even within the context of a single desktop machine a naïve update strategy may be unacceptable; for example, a search facility in which update costs were the major consumer of CPU and disk cycles would not be of value. Indexes for text databases present difficult problems for update, as efficient query processing requires each list to be stored contiguously on disk. In principle, insertion of a single document involves adding a few bytes to the postings list of every term in the document, but a typical document contains hundreds of distinct terms. Each postings list can contain many millions of entries, a significant size even when compressed, and there are typically millions of postings lists. Update therefore is potentially costly in disk accesses and complex in space management.

In this paper we evaluate the two main strategies for maintaining text indexes, *in-place update* and *re-merge update*, focusing on addition of new documents. Only strategies that store inverted lists contiguously on disk are considered in this paper, since query performance would typically be adversely affected when single lists are stored in non-adjacent locations on disk. To our knowledge, there has been no previous experimental evaluation or comparison of these or any other update strategies for text indexes, nor has there been exploration of the practicalities of implementing these approaches.

The re-merge maintenance strategy makes use of algorithms for efficient index construction. In index construction, a collection is indexed by dividing it into blocks, constructing an index for each block, and then merging. To implement update, it is straightforward to construct an index for a block or buffer of new documents, then re-merge it with the existing index.

The second maintenance strategy is to amend the index in-place, list by list, to include information about a new document. However, as a typical document contains hundreds of distinct terms, such an update involves hundreds of disk accesses, a cost that is only likely to be tolerable if the rate of update is very low indeed. This cost can be ameliorated by buffering new documents; as they will share many terms, the per-document cost of update is reduced. In a practical in-place implementation, new documents are indexed in a temporary in-memory buffer. When the buffer is full, it is sequentially integrated into the main index, by accessing each list in turn and adding to it any new postings that have been accumulated. In contrast to re-merge, it is necessary to manage disk fragmentation, but re-merge is asymptotically more costly as the whole index must be copied at each update.

We propose and evaluate several methods for improving the efficiency of in-place update, based on observations of the principal costs. The first is to keep short postings lists in the vocabulary structure (Cutting & Pederson, 1990). The standard arrangement is to keep the complete vocabulary in a disk-based dynamic structure such as a B^+-tree, constituting a few percent of the total size of the index. As most terms are rare—and their lists short, often only a few bytes—their lists can be brought into the B^+-tree without significantly increasing the cost of managing it, but greatly reducing the number of accesses to lists required during update. The second method is to leave a small block of free-space at the end of each list. As most updates only add a little to each list, this simple strategy greatly reduces the amount of list copying required during update. Other, more elementary steps are also of value, such as recording the position of the last valid byte in each list; new entries can then be written in-place without a costly read of the existing list.

In all of the update approaches—re-build, in-place, and re-merge—performance depends on buffer size. Using large collections of web documents, we explore how these methods compare for different buffer sizes across a small and large collection. These experiments, using a version of our open-source ZETTAIR search engine, show that the in-place method becomes increasingly attractive as collection size grows. We had expected to observe that re-build was competitive for large buffer sizes. Interestingly, this expectation was not confirmed, with re-merge being substantially more efficient. Our results also show that the basic approach to incremental update is remarkably slow even with a large buffer; for a large collection our best speed is about 0.01 s per document, compared to roughly 0.001 s per document for batch index construction using the same implementation. However, our refinements to in-place update increase speed by a factor of, in the best case, over 5 on a 10 GB database and over 2 on a 426 GB database. With these improvements, in-place update can be competitive with re-merge.

2. Indexes for text retrieval

Inverted files are the only effective structure for supporting full text search (Witten et al., 1999; Zobel et al., 1998). An inverted index is a structure that maps from a query term—typically a word—to a *postings list* that identifies the documents that contain that term. For efficiency at search time, each postings list is stored contiguously[1]; typical query terms occur in 0.1–1% of the indexed documents, and thus list retrieval, if fragmented, would be an unacceptable overhead.

The set of distinct terms occurring in the collection is its *vocabulary* and each term has a postings list. Each postings list in a typical implementation contains the number and locations of term occurrences in the document, for each document in which the term occurs. More compact alternatives are to omit the locations, or even to omit the number of occurrences, recording only the document identifiers. However, term locations can be used for accurate ranking heuristics and for resolution of advanced query types such as phrase queries (Williams, Zobel, & Bahle, 2004).

Entries in postings lists are typically ordered by document number, where numbers are ordinally assigned to documents based on the order in which they are indexed by the construction algorithm. This is known as *document ordering* and is commonly used in text retrieval systems because it is straightforward to maintain, and additionally yields compression benefits as discussed below. Ordering postings list entries by metrics other than document number can achieve significant efficiency gains during ranked query evaluation (Anh & Moffat, 2002; Garcia, Williams, & Cannane, 2004; Persin, Zobel, & Sacks-Davis, 1996), but the easiest way to construct such variant indexes is by post-processing a document-ordered index.

Another key to efficient evaluation of text queries is index compression. Well-known integer compression techniques (Elias, 1975; Golomb, 1966; Williams & Zobel, 1999) can be cheaply applied to postings lists to significantly reduce their size. Integer compression has been shown to reduce query evaluation cost by orders of magnitude for indexes stored both on disk and in main memory (Scholer et al., 2002).

To realise a high benefit from integer compression, a variety of techniques can be used to reduce the magnitude of the numbers stored in postings lists. For example, document ordering allows differences to be taken between consecutive numbers, and then the differences can be encoded; this reduces both the magnitude and distribution of values. This technique, known as taking *d-gaps*, can also be applied to within-document term occurrence information in the postings list, for further compression gains. Golomb-coding of d-gaps,

[1] For example, we have observed (in experiments not reported here) that, on a large collection, splitting every list into two components that are stored in separate areas on disk increases query evaluation time by about 50%.

as-suming terms are distributed randomly amongst documents, yields optimal bitwise codes (Witten et al., 1999); alternatively, byte- or word-oriented codes allow much faster decompression (Anh & Moffat, 2002, 2005; Scholer et al., 2002).

Compression can reduce the total index size by a factor of three to six, and decompression costs are more than offset by the reduced disk transfer times. However, both integer compression and taking d-gaps constrain decoding of the postings lists to be sequential in the absence of additional information. This can impose significant overheads in situations where large portions of the postings list are not needed in query evaluation.

In a typical text retrieval system, evaluation of a ranked query—which are around 80% of queries (Spink, Wolfram, Jansen, & Saracevic, 2001)—involves reading the postings lists associated with the terms in the query. The lists are then processed from least- to most-common term (Kaszkiel, Zobel, & Sacks-Davis, 1999). For each document that is referenced in each postings list, a score for that document is increased by the result of a similarity computation such as the cosine (Witten et al., 1999) or Okapi BM-25 (Robertson, Walker, Hancock-Beaulieu, Gull, & Lau, 1992) measures. The similarity function considers factors including the length of the document, the number of documents containing the term, and the number of times the term occurred in the document.

Other kinds of query include Boolean, which involve reading only the document numbers, and phrase, which involve reading the document numbers, frequencies, and locations. In these kinds of query, as for ranked queries, the whole of each postings list must be processed.

Techniques for decreasing the decoding costs imposed by index compression have been proposed. Skipping (Moffat & Zobel, 1996) involves encoding information into the postings lists that allows portions of the postings list to be passed over without cost during decoding. Skipping was found to increase the speed at which conjunctive queries, such as Boolean queries, can be processed. Non-conjunctive queries can also benefit from this approach, by processing postings lists conjunctively after selecting a set of candidate results disjunctively (Anh & Moffat, 1998). However, with increases in processor speeds the gains due to skipping have been argued to be no longer significant (Williams et al., 2004).

2.1. Index construction

Inverted indexes are key to fast query evaluation, but construction of the inverted index is a resource-intensive task. On 2004 hardware and using techniques described twelve years ago (Harman & Candela, 1990), we estimate that the inversion process would require around half a day per gigabyte. The latest techniques in index construction have dramatically reduced this time, to (in 2004) around 2 min/GB on the same hardware.

The most efficient method for index construction is a refinement of sort-based inversion (Heinz & Zobel, 2003). Sort-based inversion operates by recording a posting (consisting of a term, ordinal document number, and position information) in temporary disk space for each term occurrence in the collection. Once the postings for the entire collection have been accumulated in temporary disk space, they are sorted—typically using an external merge-sort algorithm—to group postings for the same term into postings lists (Harman & Candela, 1990). The postings lists then constitute an inverted index of the collection. Sort-based inversion has the advantages that it only requires one pass over the collection and can operate in a limited amount of memory, as full vocabulary accumulation is not required.

Simple implementations of sort-based inversion are impractically slow, but the strategy of creating temporary indexes in memory, writing them as runs, then merging the runs to yield the final index is highly efficient. In this variant, an in-memory buffer is used to accumulate a temporary index, consisting of a simple vocabulary structure—such as a hash table or trie (Heinz, Zobel, & Williams, 2002)—and lists for each term in the structure. When memory is full, this temporary index is flushed to disk as a compact *run*. Once all the documents have been processed, the runs are merged to give the final index. With careful disk management, this merge can reuse the space allocated to the runs, so that the disk overhead is small.

An alternative to sort-based inversion is in-memory inversion (Witten et al., 1999), which proceeds by building a matrix of terms in the collection in a first pass, and then filling in document and term occurrences in a second pass. If statistics about term occurrences are gathered during the first pass, the exact amount of memory required to invert the collection can be allocated, using disk if necessary. Term occurrence information is written into the allocated space in a second pass over the collection. Allocation of space to hold postings from disk allows in-memory inversion to scale to very large collection sizes. However, in-memory inversion has the disadvantages that it requires two passes over the collection and the vocabulary must be accumulated over the entire collection.

Another alternative to construction is a hybrid sorting approach (Moffat & Bell, 1995) in which the vocabulary is kept in memory while blocks of sorted postings are written to disk. However, compared to the pure sort-based approach, more memory and indexing time is required (Heinz & Zobel, 2003). For the largest collection we use in our experiments, the size of the vocabulary prohibits the use of such approaches.

The techniques described here have been implemented in the ZETTAIR text search engine, written by the Search Engine Group at RMIT.[2] ZETTAIR was used for all experiments described in this paper.

3. Index update strategies

For text retrieval systems, the principles of index maintenance or update are straightforward. When a document is added to the collection, the index terms are extracted; a typical document contains several hundred distinct terms that must be indexed. (It is well established that all terms, with the exception of a few common terms such as "the" and "of", must be indexed to provide effective retrieval (Baeza-Yates & Ribeiro-Neto, 1999; Witten et al., 1999). For phrase matching to be accurate, all terms must be indexed (Williams et al., 2004).) For each of these terms, it is necessary to add to the list information about the new document and thus increase its length by a few bytes, then store the modified list back on disk.

This simple approach to update, naïvely implemented, carries unacceptable costs. For example, on the 426 GB TREC GOV2 collection (Clarke, Craswell, & Soboroff, 2004), the postings list for "the"—the commonest of the indexed terms—is around 1 GB. In contrast, the average list is about half a kilobyte. To complete the update, the system must fetch and modify a vast quantity of data, find contiguous free-space on disk for modified postings lists, and garbage-collect as the index becomes fragmented. Therefore, the only practical solution is to amortize the costs over a series of updates.

The problem of index maintenance for text data has not been broadly investigated: there is relatively little published research on how to efficiently modify an index as new documents are accumulated or existing documents are deleted or changed. Cutting and Pederson (1990) describe the in-memory buffering scheme that we use in this paper, and propose keeping short lists in the vocabulary structure. Barbará, Mehrotra, and Vallabhaneni (1996) describe an indexing engine that they use as an email repository. They utilize an in-place strategy, which we explain later, but focus on concurrent queries and updates. The in-place strategy is further explored in Tomasic, Garcia-Molina, and Shoens (1994), who experiment with space allocation strategies to optimize either maintenance speed or querying speed. A more complex over-allocation scheme, based on the rate of inverted list growth, is proposed by Shieh and Chung (2005). Tomasic et al. (1994) and Shoens et al. (1994) suggest grouping short lists in fixed-size buckets, to reduce the problem of managing space for numerous small inverted lists; this approach is likely to be a useful incremental improvement to our approaches, and we plan to investigate it further. A variation of our re-merge strategy is proposed (Clarke & Cormack, 1995; Clarke, Cormack, & Burkowski, 1994). Most of this prior work was undertaken prior to the major innovations in text representation and index construction described in the previous section, and thus much of its validity is, at best, unclear.

There are several possible approaches to cost amortization for index maintenance, as considered in detail later. A shared element of all these approaches is that new documents are indexed in a temporary structure in memory, to amortize costs (Barbará et al., 1996; Clarke et al., 1994). This index is similar to the structure used to build a run during bulk index construction. This index provides immediate query-time access to new documents, and can simplify subsequent update processes.

Update techniques from other areas cannot be readily adapted to text retrieval systems. For example, there is a wide body of literature on maintenance of data structures such as B-trees, and, in the database field, specific research on space management for large objects such as image data (Biliris, 1992a, 1992b; Carey, DeWitt, Richardson, & Shekita, 1986, 1989; Lehman & Lindsay, 1989). However, these results are difficult to apply to

[2] Available at http://www.seg.rmit.edu.au/zettair.

text indexes, which present very different technical problems to those of indexes for conventional databases. On the one hand, the number of terms per document and the great length of postings lists make the task of updating a text retrieval system much more costly than is typically the case for conventional database systems. On the other hand, as query-to-document matching is an approximate process—and updates do not necessarily have to be instantaneous as there is no equivalent in a text system to the concept of integrity constraint—there are opportunities for novel solutions that would not be considered for a conventional database system.

Three update algorithms are compared in the experiments presented in this paper. All three accumulate postings in main memory as documents are added to the collection. These postings can be used to resolve queries, making new documents available immediately. Once main memory is filled with accumulated postings, the index is updated according to one of the three strategies. We describe these strategies next.

3.1. Re-build

The simplest approach to maintenance is to build a new index once a sufficient number of new documents has been added to the collection. This re-build algorithm discards the current index after constructing an entirely new one. The new index is built by processing the stored collection, including any new documents added since the last build. To service queries during the re-building process, a copy of the old index and the accumulated in-memory postings must be kept. After the re-building process is finished, the in-memory postings and old index are discarded and the new index substituted in their place. Explicitly, this process is as follows:

1. Postings are accumulated in main memory as documents are added to the collection.
2. Once main memory is exhausted, a new index is built from the entire collection.
3. The old index and in-memory postings are discarded, replaced by the new index.

The re-building algorithm constructs a new index from stored sources each time that update is required. This necessitates that the entire collection be stored and re-processed during indexing, and existing postings are ignored.

This approach has obvious disadvantages, but these do not mean that it is unacceptable in practice. Consider, for example, a typical 1 GB university web site. A re-build might take 2 min: this is a small cost given the time needed to crawl the site and the fact that there is no particular urgency to make updates immediately available. Given the complexities of managing updates, the use of re-build may well be acceptable in this scenario and, indeed, it is the strategy adopted by some commercial enterprise-level search engines.

A separate copy of the index must be maintained to resolve queries during the maintenance process; alternatively, while we do not consider such approaches in detail, the search service may be made unavailable while the index is constructed. In addition, as in the other approaches, postings must still be accumulated in-memory to defer index maintenance and these must be kept until the re-build is complete. This requirement has impact on the index construction process, since less memory is available to construct runs.

3.2. Re-merge

The re-merge algorithm updates the on-disk index by linearly merging the file of on-disk postings with the postings in main memory, writing the result to a new disk location. This requires one complete scan of the existing index. The on-disk postings and the in-memory postings are both processed in ascending lexicographical term order, allowing the use of a simple merge algorithm to combine them. After the merge is finished, the new index is substituted for the old.

In detail, this algorithm is as follows:

1. Postings are accumulated in main memory as documents are added to the collection.
2. Once main memory is exhausted, for each in-memory postings list and on-disk postings list:
   (a) If the term for the in-memory postings list is less than the term for the on-disk postings list, write the on-disk postings list to the new index and advance to the next in-memory list.

(b) Otherwise, if the in-memory posting term is equal to the on-disk postings term, write the on-disk postings list followed by the in-memory postings list to the new index. Advance to the next in-memory and on-disk postings lists.
(c) Otherwise, write the on-disk postings list to the new index and advance to the next on-disk postings list.
3. The old index and in-memory postings are discarded, replaced by the new index.

The re-merge algorithm processes the entire index, merging in new postings that have been accumulated in memory. This algorithm allows the index to be read efficiently, by processing it sequentially, but forces the entire index to be processed for each update. We have found—as we report later in detail—that the merge takes tens of minutes on an index for 100 GB of text.

To avoid the system being unavailable during the merge, a copy of the index can be kept in a separate file, and the new index switched in once the merge is complete. A drawback, therefore, is that the new index is written to a new location and there are therefore two copies of the index; however, the index can be split and processed in chunks to reduce this redundancy. In contrast to the in-place scheme that we describe next, this scheme has the benefit that lists are stored adjacently, that is, there is no disk fragmentation. A further benefit is that the update process is fairly straightforward and for moderate-sized collections and with a large buffer for the temporary index we expect that the scheme (when measured per document) would be reasonably fast.

3.3. In-place

The in-place update strategy is to amend postings lists in an existing index as new documents are added to the collection. In practice, as for the other strategies, postings are buffered in main memory. Then, once main memory is exhausted, the postings lists on disk are individually merged in-place with entries from the temporary index. Updating of postings lists in-place requires consideration of the problems associated with space management of postings lists, but the cost of update can be significantly reduced by reducing the number of times that individual postings lists have to be written to disk.

In-place updating follows two basic principles. First, list updates should be in lexicographical term order, maximising the benefit of caching in the vocabulary access structure, which is also lexicographically sorted. (In unreported experiments, we tested alternative sort orders; none were better than lexicographical ordering, with all but reverse lexicographical sorting being disastrous for the efficiency of the in-place strategy.) Second, free-space for relocation of postings lists that are new or have been extended should be managed using a list of free locations on the disk, or *free map*. These must be stored sorted in order of disk location, to be able to efficiently determine whether additional free-space is available after an existing allocation. Sorting by disk location also allows efficient consolidation of fragmented records into a single location, which would otherwise be costly.

The second principle above is necessary but costly. We found that maintenance of the free map is a significant cost in the in-place strategy, as the number of distinct free locations exceeds a million entries in some of our experiments. As well as providing efficient, scalable access by disk location, the free map must also support efficient access to entries of the correct size for new allocations. In our initial implementation of the free map (Lester, Zobel, & Williams, 2004), free locations were stored in a sorted array, which proved to a be a significant bottleneck during in-place updates. The experiments in this paper use a more efficient free map, which stores each free location in multiple lists. We describe this new approach next.

The first list contains all free locations and is sorted by disk location. To provide efficient random access to the list, a random 20% of the list nodes are inserted in a red-black tree, which can be traversed to find list nodes near a specified location. The free list structure also maintains an array of 32 size-based lists, which contain each free location within size range $[2^i, 2^{i+1} - 1)$, where i is the index of the list in the array. Using this array, the free map can satisfy requests for allocations of a given size without examining large numbers of free locations that are too small to be used. Allocation of new space of a given size involves examination of the size-based list of index $\lfloor \log_2 size \rfloor$. Entries are allocated on a first-fit basis from this list. We refer to this algorithm as *close-fit*. If no entries in the list satisfy the request, the allocation algorithm considers the next largest size-based list, until it has either found a suitable location or is forced to allocate new disk space.

Using this free map structure, the in-place algorithm first attempts to append new postings to the current list in its current location. If that fails, the postings list is relocated to a new disk location with sufficient space to hold the modified list. The complete algorithm is as follows:

1. Postings are accumulated in main memory as documents are added to the collection.
2. Once main memory is exhausted, for each in-memory postings list:
   (a) Determine how much free-space follows the corresponding on-disk postings list.
   (b) If there is sufficient free-space, append the in-memory postings list to the existing list by writing into this space.
   (c) Otherwise,
   - Determine a new disk location with sufficient space to hold the on-disk and in-memory postings lists, using a close-fit algorithm.
   - Read the on-disk postings list from its previous location and write it to the new location.
   - Append the in-memory postings list to the new location.

   (d) Discard the in-memory postings list and advance to the next.

This algorithm requires that it is possible to append to a postings list without first decoding it. Doing so involves separately storing state information that describes the end of the existing list: the last number encoded, the number of bits consumed in the last byte, and so on. For addition of new documents in document-ordered lists, such appending is straightforward; under other organizations of postings lists—such as frequency-ordered (Persin et al., 1996), impact-ordered (Anh & Moffat, 2002), or access-ordered (Garcia et al., 2004)—the entire existing list must be decoded. Our previous experiments have shown that implementations that must decode postings lists prior to updating them are significantly less efficient than implementations that store information describing the state of the end of each list (Lester et al., 2004).

An advantage of in-place update is that it is asymptotically cheaper than the other approaches. As the volume of indexed information grows, the proportion of the index that must be processed during in-place update decreases. That is, as data volume increases or buffer size is reduced, the in-place strategy theoretically becomes the most efficient update mechanism. We explore later whether this holds in practice.

3.4. Improving in-place update

We have explored two refinements to in-place update that have the potential to dramatically reduce costs, in addition to the implementation options identified above.

3.4.1. Short postings lists

The first refinement is to stored short postings lists within the vocabulary. This technique was first outlined by Cutting and Pederson (1990). Our implementation uses a B$^+$-tree (Comer, 1979) with a four kilobyte page size as the vocabulary structure, and can store data entries of up to one quarter of a page with each leaf node. Typically, this entry space is used to record the string of characters comprising a term, statistics about the term, and information such as the disk location of the postings list.

By using this space to store small postings lists directly within the vocabulary B$^+$-tree, the majority of disk accesses during update can plausibly be avoided, as most postings lists are very short. Indeed, a large fraction of lists are only a few bytes long—barely greater than the 6 bytes needed to give the disk location of the list—and thus storing them separately is not effective. We are not aware of a previous empirical investigation of this technique.

In our experiments, we limit the maximum size of postings lists stored in the vocabulary B$^+$-tree. The vocabulary B$^+$-tree caches all non-leaf nodes and the last-accessed leaf node in main memory, ensuring that a B$^+$-tree entry can be accessed in approximately one disk access per term look-up, while still using a practical amount of main memory.

Since postings lists have a maximum size for in-memory storage, our in-place algorithm must decide on the location of postings lists prior to attempting to update them. For example, if an existing postings list grows too large to be stored within the B$^+$-tree, it must be migrated to a new location on disk prior to being updated.

New postings lists (for terms that have not previously occurred) are either inserted into the vocabulary or placed on disk, depending on how large the buffered postings list is when first considered.

Note that the impact on query time is minimal; in principle, a disk access may be saved, but only a tiny minority of query terms have sufficiently short lists.

3.4.2. Over-allocation

A second refinement to the in-place strategy is to reduce the number of entries within the free map. We have observed that, while the in-place strategy produces a very large number of free disk locations, the vast majority are tiny—perhaps only a few bytes in size—and cannot be used to store postings lists. However, if we allow the space allocator to selectively over-allocate a small number of additional bytes with each space request, potentially a large number of these small locations can be removed from the free map.

A negative consequence of this strategy is a possible increase in the fragmentation of disk space used for postings lists, as this space is no longer available for subsequent allocation. Note that this optimization only applies to postings lists stored on disk; in-vocabulary postings lists are stored contiguously under the B^+-tree space organization. The conceptual in-place algorithm is unchanged by this refinement.

Tomasic et al. (1994) explore several over-allocation strategies in the context of in-place update, including over-allocating a constant amount of space for each list. This is similar to our strategy, but their intention is to minimise the number of list relocations. In contrast, the over-allocation strategy explored here is intended to minimise the number of entries in the free map, since we have found that management of this data structure is a significant performance bottleneck. In practice, the distinction is that Tomasic's strategy always overallocates extra space at the end of each list, whereas our strategy over-allocates only to the extent that an entry will be removed from the free map, up to a given maximum. Predictive over-allocation (Shieh & Chung, 2005; Tomasic et al., 1994) may provide some additional performance benefit, but we do not explore it here, as larger gains appear to be available from other optimizations.

4. Experimental design

We ran our experiments on Pentium IV 2.80 GHz machines with hyper-threading disabled. Each machine has 2 GB of main memory on a 800 MHz front-side bus. Each experiment used only a single machine. In all experiments, the machines in use were under light load, with no other significant tasks accessing disk or memory. As discussed previously, all experiments used the ZETTAIR search engine. Considerable effort was undertaken to ensure that the index construction and index maintenance strategies are as comparable as possible, with the result that they share large sections of code.

Two collections were used in the experiments, the TREC WT10g collection and the TREC GOV2 collection.[3] TREC is a large-scale international collaboration intended primarily for comparison of text retrieval methods (Harman, 1995), and provides large volumes of text data to participants, allowing direct comparison of research results. The WT10g collection contains around 1.7 million documents from a 1997 web crawl (Hawking, Craswell, & Thistlewaite, 1999); it was used primarily as an experimental collection at TREC in 2000 and 2001. The 426 GB GOV2 collection consists of 25 million documents from a 2004 web crawl of the `.gov` domain (Clarke et al., 2004).

In the first set of experiments, an initial *WT10g-9* index of 9 GB of WT10g data (1,478,628 documents) was updated using the remaining 1 GB of data (218,489 documents). The WT10g-9 index is approximately 2 GB, including word positions. Our initial experiment was to time each of the strategies when updating the WT10g index, and to compare them with our index construction implementation when constructing the entire WT10g collection. Buffer sizes were varied from about 3000 documents (typically 15 megabytes or so) to 200,000 documents (typically about a gigabyte). The temporary in-memory indexes are around a fifth of this size.

Other experiments updated the WT10g-9 index to quantify the performance benefits of both maintaining in-vocabulary postings lists and of allowing the free map to overallocate space to postings lists. To properly measure the effect of in-vocabulary lists, an initial index that also stored all postings of less than the size

[3] Information on GOV2 can be found at es.csiro.au/TRECWeb/gov2-summary.htm.

Table 1
Summary of update time (seconds) per document for each strategy, using two buffer sizes on each of the two collections

Buffer size (docs)	Collection			
	WT10g		GOV2	
	10,000	100,000	10,000	100,000
Construction	0.0009	0.0009	0.0018	0.0020
Re-build	0.1508	0.0218	4.5487*	0.7139*
Re-merge	0.0297	0.0048	0.5839	0.0846
In-place	0.1819	0.0303	0.7933	0.2307
Improved in-place	0.0946	0.0057	0.5858	0.1096

Figures marked by an asterisk are estimates based on construction times for that collection.

parameter in the vocabulary was constructed. The difference in construction time between the index with all postings lists on disk and indexes with some postings in the vocabulary was negligible.

A second set of experiments used an initial index of the entire 426 GB of GOV2 data (25,205,181 documents) and updated it using the same 1 GB of WT10g data (218,489 documents) as was used in the first set of experiments. The initial index size was approximately 45 GB, including full word positions. Index construction for this collection—which the re-build algorithm must do at least once—takes over 13 h. Estimated running times for the re-build algorithm, based on bulk load construction costs and presented in Table 1, indicate that re-build is not competitive with the in-place and re-merge methods on this collection. Only the in-place and re-merge strategies were tested in this experiment.

These experiments were chosen to explore the characteristics of the strategies under different conditions. In particular, we explored the behaviour of the approaches when the index fits into main memory, as is the case for the WT10g-9 index, and contrasted this with the behaviour when the index is many multiples of main-memory size.

5. Results

Fig. 1 shows overall results of our experiments on WT10g, presenting comparison of the three unoptimized update strategies and of bulk index construction. It also contains the best results obtained by optimizing the in-place strategy with in-vocabulary lists and overallocation (these results are discussed in detail later). The x-axis shows the number of documents buffered in main memory prior to index update; for example, the left-most point is where one hundred documents are added to the main-memory index before the on-disk index and B^+-tree is updated. As in all graphs where the number of documents buffered is shown,

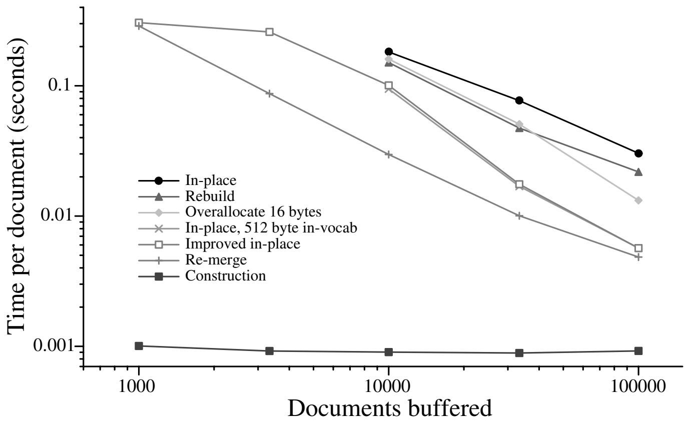

Fig. 1. Running times per input document for the update strategies on the WT10g collection, with a range of buffer sizes. Index construction per input document is also shown for construction of the entire collection. Note that the "in-place, 512 byte in-vocab" curve is obscured by "improved in-place".

the right-most data points correspond, from right to left, to 3, 7, and 22 physical updates. The y-axis shows the average time taken to add a new document to the on-disk and B^+-tree index, averaged over the entire update process.

Not surprisingly, the results show that re-building is inefficient for all but unrealistic scenarios. It is therefore clear that the only reason that re-build is a preferred strategy in some search engines is that there is an implementation cost in developing an alternative.

The basic in-place scheme (the uppermost line) is less efficient than all other schemes, even re-build. This is in contrast to our previous results (Lester et al., 2004), where the in-place scheme appeared to scale more efficiently. The transition to an on-disk vocabulary structure from the hash table used in our previous work has enabled far greater scale in the collections that are used; but the in-place scheme now has to seek to and update each vocabulary item on disk in addition to updating the postings list associated with it, making it less efficient relative to the re-merge and re-build schemes, which are able to take advantage of efficient bulk-loading algorithms in order to create an updated vocabulary. Another performance issue with the in-place strategy is that, although inverted lists are initially stored on disk in lexographical order, matching the arrangement of vocabulary entries, this order is not preserved during successive updates. After many physical updates the terms are in random order, with the exception—as discussed later—that the largest term tends to be the last in the index. This shuffling of list disk locations reduces the benefit of processing updates in lexographical order.

However, as can be seen later with regard to Fig. 8, with larger collections the in-place scheme is increasingly competitive for a given buffer size. That is, as collection size grows, the efficiency of re-merge degrades more rapidly than does the efficiency of in-place update. The change in efficiency is not dramatic; re-merge and in-place update are about equal for a 1000 document buffer on WT10g and for a 7000 document buffer on GOV2.

5.1. In-vocabulary lists

The line labelled ‘‘in-vocab’’ in Fig. 1 shows the efficiency of the in-place scheme when short postings lists of up to 512 bytes (encoded using the vbyte (Scholer et al., 2002) compression scheme) are stored within the vocabulary. Values ranging from 8 to 512, in multiples of 2, were tested. These detailed results are shown in Fig. 2, which shows that keeping increasingly long postings lists within the vocabulary is of benefit for maintenance speed, though a limit is approached. Even keeping the short postings lists within the vocabulary provides a dramatic speed-up during update.

Keeping short lists within the vocabulary has the potential to degrade the space utilization within the vocabulary. In order to verify that space was not being used poorly, the utilization of the index was recorded, as presented in Fig. 3. Our reporting of utilization considers all postings lists to be utilized space, as well as

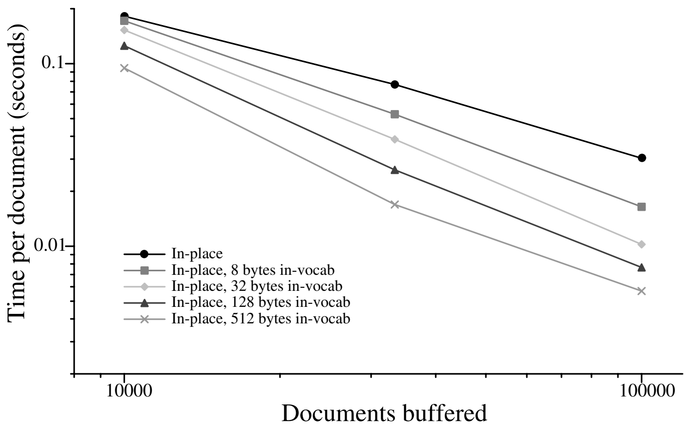

Fig. 2. Running times per input document for the in-place update strategy on the WT10g collection, varying the maximum size of postings lists stored within the vocabulary, with a range of buffer sizes.

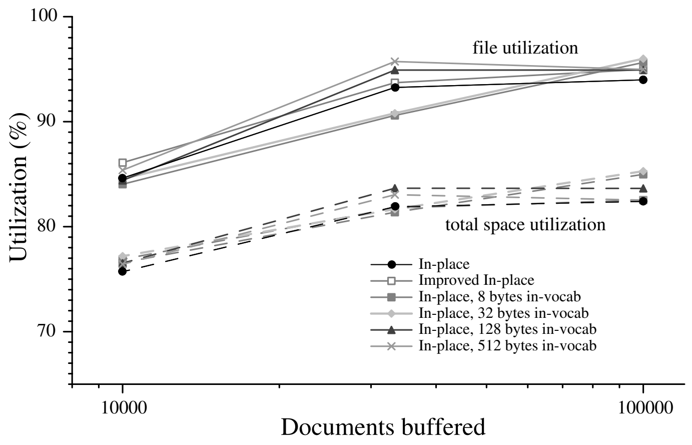

Fig. 3. Index fragmentation for the in-place update strategy on the WT10g collection, varying the maximum size of postings lists stored within the vocabulary, with a range of buffer sizes.

statistical information in the vocabulary. Information in the vocabulary was determined by summing the space occupied by, for each term in the vocabulary, the term itself and any statistics. All integers were coded using vbyte compression, and the term was represented by a length count instead of a delimiting scheme. Information regarding the location of postings lists, as well as the B^+-tree internal index and other overheads, were excluded in this calculation. The graph in Fig. 3 shows two different measures of utilization in the index for different sizes of postings lists stored within the vocabulary, including the reference in-place strategy. The top, solid lines show the fragmentation on disk in the space used for the postings lists alone, which is directly comparable to our previous results and to other work. The bottom, dashed line for each strategy show the total utilization of all files, including the vocabulary, which gives a true indication of the overall space utilization within the index.

According to this measure, the vocabulary B^+-tree had fairly poor utilization—of slightly more than 50%—but represented only about 10% of the total space in the index. The vocabulary does not gain a significant amount of extra information in any of the experiments, as far too small a volume of postings lists are stored within it to make much difference. As a baseline, the utilization by this measure for the re-merge strategy is 93–94%. All of the loss is due to overhead in the B^+-tree.

While we had initially feared that in-vocabulary lists might cause excessive fragmentation, the results show that in most cases this optimization actually improves utilization, making it an unconditional improvement to the in-place strategy for the metrics shown here.

5.2. Overallocation

The second improvement made to the in-place scheme was to allow overallocation of space in postings lists. This variant is intended to reduce the number of entries in the free map, particularly of small size, and may save some copying costs as postings lists grow. The results of this approach are shown in the line labelled "overallocate" in Fig. 1, and in detail in Fig. 4, where it is clear that this optimization improves the efficiency of the in-place strategy. (In these results, all postings lists are stored on disk; none are in the vocabulary.) These results indicate that, despite considerable effort spent carefully coding the free map, it remained a significant bottle-neck, and reduction in its size can yield considerable gains. As for the optimization of keeping lists in the vocabulary, the gains from overallocation diminish as the buffer size is reduced, and the cost of additional reads and writes dominates.

Fig. 5 shows the space utilization of the index for different overallocation values, in the same format as in Fig. 3. Increasing amounts of overallocation cause utilization to fall, indicating that some of the speed gains obtained using this method are as a result of over-allocation, albeit based on convenience to the implementation rather than likelihood of further expansion. The large speed gains for overallocation with a mere 4 bytes

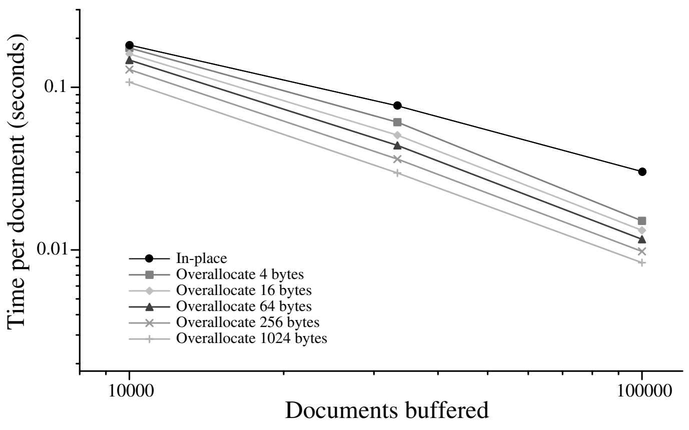

Fig. 4. Running times per input document for the in-place update strategy on the WT10g collection, varying the amount of space that can be overallocated, for a range of buffer sizes.

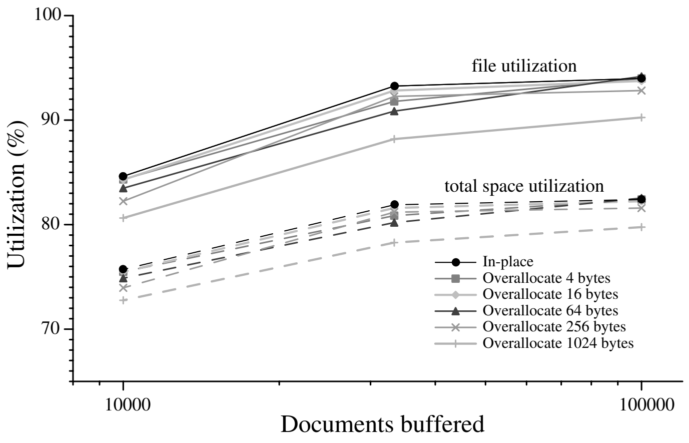

Fig. 5. Index fragmentation for the in-place update strategy on the WT10g collection, varying the amount of space that the freemap is allowed to appended to allocations, with a range of buffer sizes.

suggest that the principal gains are in reduction in the size of the free map, rather than saving of costs during copying.

The low utilization evident at the left-hand sides of Figs. 3 and 5 suggests that space management is a significant problem in the in-place strategy. In order to explore utilization of the index during maintenance, the utilization on disk was sampled after every update of the on-disk postings, using the unoptimized in-place strategy, for the WT10g collection. Buffer sizes of 1000, 3333, 10,000, and 33,333 documents were plotted, with the results shown in Fig. 6. These results indicate that, after an initial period where the utilization rapidly falls after updating the initially-packed organization given by bulk load, the utilization remains relatively stable, suggesting that the degree of utilization in the index is related to the size of the updates applied to it, not to the number of updates. This is confirmed by the fragmentation results for the GOV2 collection, shown in Fig. 7.

The oscillation that can be observed in the utilization in Fig. 6 is due to the movement of large postings lists. Postings lists for frequently occurring words such as "the" have to be updated for almost every document added to the index. This frequent growth causes the postings list to be moved toward the back of the index, where previously unused space can be allocated to hold them. Utilization then falls because a large space in the index is left in the previous position of the list. Once at the back of the index, the large postings list can grow without having to be moved and smaller postings lists can be placed in its previous position. This causes

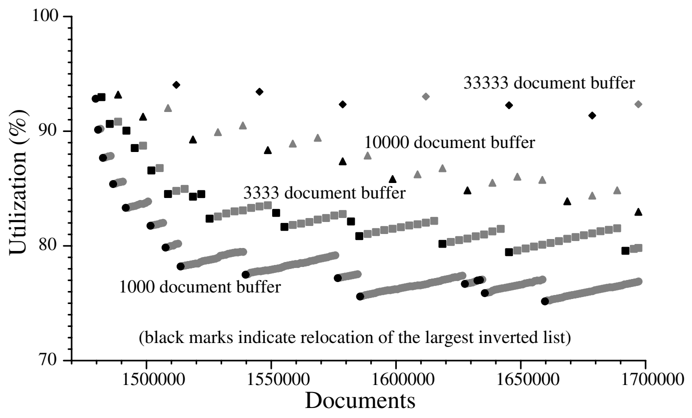

Fig. 6. Index fragmentation of the in-place strategy on the WT10g collection during the maintenance process.

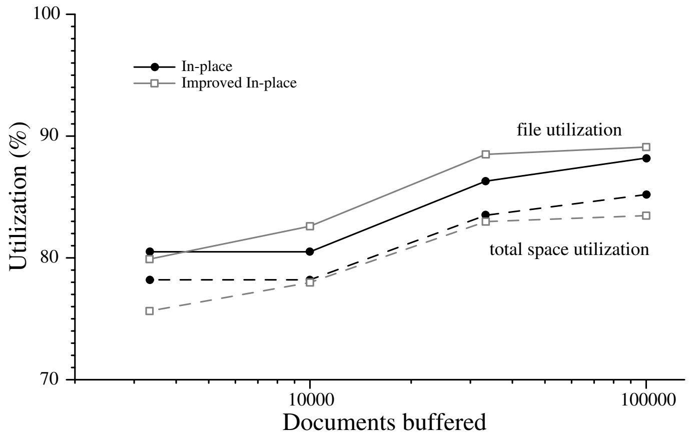

Fig. 7. Index fragmentation of the in-place strategy on the GOV2 collection during the maintenance process.

utilization to rise, and can continue until another large postings list needs to be relocated to the end of the index, starting the process again.

5.3. Combination of methods

The two enhancements to in-place update—in-vocabulary lists and overallocation—are compared in Fig. 1. First, for most buffer sizes in-vocabulary lists provide much greater gain than does overallocation. Second, combining the two provides no benefit in addition to using in-vocabulary lists without overallocation. Taking these results in conjunction with our analysis of fragmentation costs, with lists in the vocabulary we conclude that the potential benefits of overallocation are small.

5.3.1. GOV2

The size of the GOV2 collection prohibits exhaustive experiments of the kind reported earlier, as each run for a given buffer size or other parameter setting takes 20–200 times longer to complete. We therefore used GOV2 to confirm the results obtained on WT10g.

These results for GOV2 are shown in Fig. 8. The three lines are for re-merge; the basic implementation of in-place update; and in-place update with in-vocabulary postings lists of up to half a kilobyte and overallocation of 16 bytes per list. (As noted earlier, re-build is not feasible on this collection.) For the smallest buffer

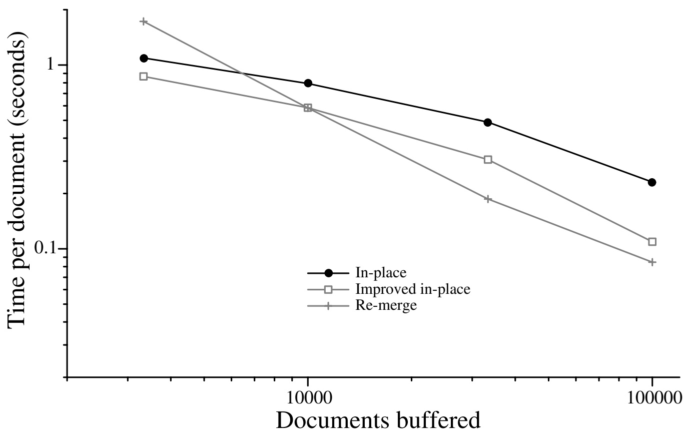

Fig. 8. Running times per input document for the update strategies on the GOV2 collection, with a range of buffer sizes. The "improved in-place" line has in-vocabulary postings lists of up to 512 bytes and overallocation of 16 bytes.

sizes, in-place update is superior to re-merge. As buffer size is increased, the basic in-place scheme becomes relatively less efficient. However, the improved in-place scheme remains competitive.

GOV2 is about 40 times the size of WT10g, but the data is sparser; the difference in index sizes is about a factor of 20. Each update on GOV2 takes about 20 times longer per document than on WT10g, suggesting that the costs are in practice no worse than linear, as the WT10g index is approximately the same size as memory, while only a small fraction of the GOV2 index can be loaded, so disk costs are likely to be greater.

5.4. Implementation considerations

Compared to our previous published results (Lester et al., 2004), the efficiency of the update schemes reported in this paper is markedly improved. However, much of the gain is due to implementation issues: the algorithms are identical to our previous versions. Here we discuss decisions that we made in developing code for this work.

5.4.1. Vocabulary management

The most significant change for the results described in this paper is the use of a B^+-tree as a vocabulary structure. Our previous implementation used a large in-memory hash table for vocabulary management (Zobel, Williams, & Heinz, 2001), motivated by the expectation that vocabulary size would remain much smaller than memory. However, this expectation was false. Use of a B^+-tree was a complete success: vocabulary memory requirements are only a fraction of the requirements of a hash table and the scalability has greatly increased; we are now able to search up to one terabyte of web data on a standard desktop computer.

The choice of a B^+-tree tree has no disadvantages compared to a hash table. Despite requiring an average of one disk access per term lookup or update, it is as fast as the hash table at query time, due to factors such as caching of leaves in memory. Storage of data in main memory is not a solution to search engine performance problems. While it seems attractive, it fails to account for the opportunity cost of doing so. Main-memory is a versatile resource for performance tuning: there is virtually no application component that does not benefit from an increase in available main memory. Committing large amounts of RAM to hold infrequently used data precludes using that memory for other purposes—in this case, making updates less frequent and buffering data read from disk.

5.4.2. Managing document identifiers

Consider again the task of index construction. A collection is processed in chunks, with the result that a partial inverted index—a run—is created for that chunk; the runs are then merged to create the final, complete inverted index. The documents that contribute to each run do not overlap, that is, each partial index is for a distinct set of documents from the collection. We make use of this property to improve the performance of both index construction and update.

In the final merge step in index construction, we do not decode and recode the partial postings lists that are being merged. Instead, we copy and contiguously organize the partial postings list into a single, consolidated list directly. Performed naïvely, this would mean that the d-gap approach is not followed: at the beginning of each partial list, an exact document number is encoded and not a difference from the previous identifier in the previous run. We therefore decode only the first value in each partial list, replacing it in-place with a d-gap value computed from the document identifier that terminates the previous partial list.

To support index update in this way, we must store the final document identifier from each on-disk list in its original, decoded form. We do this by keeping each final document identifier in the corresponding entry in the vocabulary, which must be accessed regardless in order to determine the disk location of the list. Alternatively, the last value could be stored in main memory, but this requires vocabulary accumulation during construction, the lack of which is one of the primary advantages of the index construction algorithm we use.

Note that the variable byte index compression schemes that we use greatly simplify these processes. It is unclear how these techniques could be implemented efficiently using variable bit schemes.

5.4.3. Document deletion

While we do not evaluate mechanisms to delete documents from an inverted index, the strategy used to add new documents to the index a suggests plausible strategy for deleting documents. Ideally, addition of new documents to and deletion of expired documents from an index should occur in the same set of operations.

Using the re-build strategy, deletion is straight-forward, as the deleted documents are simply not added to the index during the next re-build. Document deletion under the re-merge strategy is also straightforward, since the entire index is processed for each physical update. However, this requires that the contents of inverted lists be examined to determine the location of expired postings, whereas without deletions the existing contents of each inverted list need merely be copied. Deletion is more difficult under the in-place strategy, as it is necessary to establish which lists need to be updated to effect the removal of each document.

A simple alternative is to mark documents as expired in the global document map, maintain their index information, and not return these documents in response to queries. Unless the deletion rate is high, occasional housekeeping should then be sufficient for long-term list maintenance.

6. Conclusions

The structure of inverted indexes is well-understood, and their construction and use in querying has been an active research area for almost fifteen years. However, there is little published research on how to maintain inverted indexes when documents are added to a collection. We have undertaken a detailed experimental exploration of the costs of updating inverted indexes for text databases, using techniques that ensure maximum query performance.

The costs of a naïve approach to update are unacceptably high, but it was not clear which of the practical alternatives—re-merge and in-place update—would be most efficient in practice; nor was it clear how to implement an effective in-place update scheme. It was not even clear that these schemes would be substantially faster than the baseline of re-building the index whenever a sufficient number of new documents had arrived. Our results, summarized in Table 1, are to our knowledge the first experimental evaluation of these alternatives in a modern setting. As shown in this table, bulk load remains by far the most efficient method of index construction. For small collections and buffer sizes, re-build (used by some commercial systems) is indeed competitive, and is faster than a straightforward implementation of in-place update. However, re-merge is the most efficient update method for all but the smallest buffer sizes.

Our implementation of a straightforward in-place strategy, although carefully engineered, is the slowest of the methods considered, due primarily to the difficulty of managing the free map. In contrast to the other approaches, in-place update leads to fragmentation in the index, so the ongoing space requirements are higher. However, an in-place approach is desirable if it can be made more efficient, since it is the only strategy in which two copies of the index are not needed during update. The combined strategies of overallocating the space required for postings lists and keeping short lists in the vocabulary has a dramatic effect on update costs for the in-place strategy, especially for larger buffer sizes. (As discussed previously, the gain is almost entirely due to the second of these strategies.)

We believe that optimization of the in-place strategy is a promising area for future work. Unlike the re-build and re-merge strategies—which are the product of more than ten years of index construction research—the in-place strategy is largely uninvestigated. We plan to investigate how space can be pre-allocated during construction to reduce later fragmentation, what strategies work best for choosing and managing free-space, and whether special techniques for frequently-used or large entries can reduce overall costs.

Acknowledgments

This work was supported by the Australian Research Council and an RMIT VRII grant. We thank the two reviewers for their suggestions.

References

Anh, V. N., & Moffat, A. (1998). Compressed inverted files with reduced decoding overheads. In R. Wilkinson, B. Croft, K. van Rijsbergen, A. Moffat, & J. Zobel (Eds.), *Proceedings of the ACM-SIGIR international conference on research and development in information retrieval*, Melbourne, Australia (pp. 291–298).

Anh, V. N., & Moffat, A. (2002). Impact transformation: effective and efficient web retrieval. In M. Beaulieu, R. Baeza-Yates, S. Myaeng, & K. Jävelin (Eds.), *Proceedings of the ACM-SIGIR international conference on research and development in information retrieval*, Tampere, Finland (pp. 3–10).

Anh, V. N., & Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval, 8*(1), 151–166.

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern information retrieval*. Addison-Wesley Longman.

Barbará, D., Mehrotra, S., & Vallabhaneni, P. (1996). The Gold text indexing engine. In S. Y. W. Su (Ed.), *Proceedings of the IEEE international conference on data engineering* (pp. 172–179). Los Alamitos, California, New Orleans, Louisiana: IEEE Computer Society Press.

Biliris, A. (1992a). An efficient database storage structure for large dynamic objects. In F. Golshani (Ed.), *Proceedings of the IEEE international conference on data engineering* (pp. 301–308). Tempe, Arizona: IEEE Computer Society.

Biliris, A. (1992b). The performance of three database storage structures for managing large objects. In M. Stonebraker (Ed.), *Proceedings of the ACM-SIGMOD international conference on the management of data*, San Diego, California (pp. 276–285).

Carey, M. J., DeWitt, D. J., Richardson, J. E., & Shekita, E. J. (1986). Object and file management in the EXODUS extensible database system. In W. W. Chu, G. Gardarin, S. Ohsuga, & Y. Kambayashi (Eds.), *Proceedings of the international conference on very large databases* (pp. 91–100). Kyoto, Japan: Morgan Kaufmann.

Carey, M. J., DeWitt, D. J., Richardson, J. E., & Shekita, E. J. (1989). Storage management for objects in EXODUS. In W. Kim & F. H. Lochovsky (Eds.), *Object-oriented concepts, databases, and applications* (pp. 341–369). New York: Addison-Wesley Longman.

Clarke, C. L. A., & Cormack, G. V. (1995). *Dynamic inverted indexes for a distributed full-text retrieval system*. Technical Report MT-95-01, Department of Computer Science, University of Waterloo, Waterloo, Canada, multiText Project Technical Report.

Clarke, C. L. A., Cormack, G. V., & Burkowski, F. J. (1994). *Fast inverted indexes with on-line update*. Technical Report CS-94-40, Department of Computer Science, University of Waterloo, Waterloo, Canada.

Clarke, C., Craswell, N., & Soboroff, I. (2004). Overview of the TREC 2004 terabyte track. In E. M. Voorhees, & L. P. Buckland (Eds.), *The thirteenth text retrieval conference* (*TREC 2004*). NIST Special Publication SP 500-261.

Comer, D. (1979). Ubiquitous B-tree. *Computing Surveys, 11*(2), 121–137.

Cutting, D. R., & Pedersen, J. O. (1990). Optimizations for dynamic inverted index maintenance. In J.-L. Vidick (Ed.), *Proceedings of the ACM-SIGIR international conference on research and development in information retrieval* (pp. 405–411). Brussels, Belgium: ACM.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory, IT-21*(2), 194–203.

Garcia, S., Williams, H., & Cannane, A. (2004). Access-ordered indexes. In V. Estivill-Castro (Ed.), *Proceedings of the Australasian computer science conference* (pp. 7–14).

Golomb, S. W. (1966). Run-length encodings. *IEEE Transactions on Information Theory, IT-12*(3), 399–401.

Harman, D. (1995). Overview of the second text retrieval conference (TREC-2). *Information Processing and Management, 31*(3), 271–289.

Harman, D., & Candela, G. (1990). Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science, 41*(8), 581–589.

Hawking, D., Craswell, N., & Thistlewaite, P. (1999). Overview of TREC-7 very large collection track. In E. M. Voorhees, & D. K. Harman (Eds.), *The eighth tert REtrieval conference* (*TREC-8*) (pp. 91–104). National Institute of Standards and Technology Special Publication 500-246, Gaithersburg, MD.

Heinz, S., & Zobel, J. (2003). Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology, 54*(8), 713–729.

Heinz, S., Zobel, J., & Williams, H. E. (2002). Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems, 20*(2), 192–223.

Kaszkiel, M., Zobel, J., & Sacks-Davis, R. (1999). Efficient passage ranking for document databases. *ACM Transactions on Information Systems, 17*(4), 406–439.

Lehman, T. J., & Lindsay, B. G. (1989). The Starburst long field manager. In P. M. G. Apers, & G. Wiederhold (Eds.), *Proceedings of the international conference on very large databases*, Amsterdam, The Netherlands (pp. 375–383).

Lester, N., Zobel, J., & Williams, H. (2004). In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In: V. Estivill-Castro (Ed.), *Proceedings of the Australasian computer science Conference* (pp. 15–22).

Moffat, A., & Bell, T. A. H. (1995). In situ generation of compressed inverted files. *Journal of the American Society of Information Science, 46*(7), 537–550.

Moffat, A., & Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems, 14*(4), 349–379.

Persin, M., Zobel, J., & Sacks-Davis, R. (1996). Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science, 47*(10), 749–764.

Robertson, S. E., Walker, S., Hancock-Beaulieu, M., Gull, A., & Lau, M. (1992). Okapi at TREC. In *Proceedings of the text retrieval conference* (*TREC*) (pp. 21–30).

Scholer, F., Williams, H. E., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. In K. Järvelin, M. Beaulieu, R. Baeza-Yates, & S. H. Myaeng (Eds.), *Proceedings of the ACM-SIGIR international conference on research and development in information retrieval*, Tampere, Finland (pp. 222–229).

Shieh, W.-Y., & Chung, C.-P. (2005). A statistics-based approach to incrementally update inverted files. *Information Processing and Management, 41*(2), 275–288.

Shoens, K., Tomasic, A., & García-Molina, H. (1994). Synthetic workload performance analysis of incremental updates. In W. B. Croft, & C. J. van Rijsbergen (Eds.), *Proceedings of the international ACM SIGIR conference on research and development in information retrieval*, Dublin, Ireland (pp. 329–338).

Spink, A., Wolfram, D., Jansen, B. J., & Saracevic, T. (2001). Searching the web: the public and their queries. *Journal of the American Society for Information Science, 52*(3), 226–234.

Tomasic, A., Garcia-Molina, H., & Shoens, K. (1994). Incremental updates of inverted lists for text document retrieval. In *Proceedings of the ACM-SIGMOD international conference on the management of data* (pp. 289–300). Minneapolis, Minnesota: ACM.

Williams, H. E., & Zobel, J. (1999). Compressing integers for fast file access. *Computer Journal, 42*(3), 193–201.

Williams, H., Zobel, J., & Bahle, D. (2004). Fast phrase querying with combined indexes. *ACM Transactions on Information Systems, 22*(4), 573–594.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: compressing and indexing documents and images* (2nd ed.). San Francisco, California: Morgan Kaufmann.

Zobel, J., Moffat, A., & Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems, 23*(4), 453–490.

Zobel, J., Williams, H. E., & Heinz, S. (2001). In-memory hash tables for accumulating text vocabularies. *Information Processing Letters, 80*(6), 271–277.