In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems

Nicholas Lester **Justin Zobel** **Hugh E. Williams**

School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Victoria 3001, Australia
Email: {nml,jz,hugh}@cs.rmit.edu.au

Abstract

Indexes are the key technology underpinning efficient text search. A range of algorithms have been developed for fast query evaluation and for index creation, but update algorithms for high-performance indexes have not been evaluated or even fully described. In this paper, we explore the three main alternative strategies for index update: in-place update, index merging, and complete re-build. Our experiments with large volumes of web data show that re-merge is for large numbers of updates the fastest approach, but in-place update is suitable when the rate of update is low or buffer size is limited.

1 Introduction

High-performance text indexes are key to the use of modern computers. They are used in applications ranging from the large web-based search engines to the "find" facilities included in popular operating systems, and from digital libraries to online help utilities. The past couple of decades have seen dramatic improvements in the efficiency of query evaluation using such indexes (Witten, Moffat & Bell 1999, Zobel, Moffat & Ramamohanarao 1998, Scholer, Williams, Yiannis & Zobel 2002). 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 databases at a rate of around 8 gigabytes per hour, that is, a gigabyte every 8 minutes.

In contrast, the problem of efficient maintenance of inverted indexes has had relatively little investigation. Yet the problem is an important one. In some applications, documents arrive at a high rate, and even within the context of a single desktop machine a naive update strategy may be unacceptable — a search facility in which update costs were the system's major consumer of CPU cycles and disk cycles would not be of value.

In this paper we explore the three main strategies to maintaining text indexes, focusing on addition of new documents.

To our knowledge, there has been no previous evaluation of alternative update strategies.

The first strategy for update is to simply amend the index, 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 will be reduced. The second strategy is to re-build the index from scratch when a new document arrives. On a per-document basis this approach is extremely expensive, but if new documents are buffered then overall costs may well be acceptable. Indeed many intranet search services operate with exactly this model, re-crawling (say) every week and re-indexing. The third strategy is to make use of the strategies employed in algorithms for efficient index construction. In these algorithms, 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 of new documents, then merge it with the existing index.

In all of these approaches, performance depends on buffer size. Using large collections of web documents, we explore how these methods compare for different index sizes and buffer sizes. These experiments, using a version of our open-source LUCY 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; this expectation was not confirmed, with re-merge being substantially more efficient. Our results also show that incremental update of any kind is remarkably slow even with a large buffer; for a large collection our best speed is about 0.1 seconds per document, compared to roughly 0.003 seconds per document for batch index construction using the same implementation.

Overall, given sufficient buffer space for new documents and sufficient temporary space for a copy of the index, it is clear that re-merge is the strategy of choice. For a typical desktop application, however, where keeping of spare indexes may be impractical, in-place update is not unduly expensive and provides a reasonable pragmatic alternative.

2 Indexes for text retrieval

Inverted files are the only effective structure for supporting 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; 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 terms that occur in the collection is known as the *vocabulary*. 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 (Bahle, Williams & Zobel 2002).

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. However, ordering postings list entries by metrics other than document number can achieve significant efficiency gains during query evaluation (Anh & Moffat 2002, Persin, Zobel & Sacks-Davis 1996).

Another key to efficient evaluation of text queries is index compression. Well-known integer compression techniques (Golomb 1966, Elias 1975, Scholer et al. 2002) can be 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 memory (Scholer et al. 2002).

To realise maximal benefits from integer compression, a variety of techniques are 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 rather than the document numbers. 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, assuming terms are distributed randomly amongst documents, yields optimal bitwise codes (Witten et al. 1999); alternatively, byte-oriented codes allow much faster decompression (Anh & Moffat 2002, 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 performed sequentially 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.

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. This can greatly increase the speed at which conjunctive queries, such as Boolean AND 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).

Inverted indexes are key to fast query evaluation, but construction of the inverted index is a resource intensive task. On 2001 hardware and using techniques described twelve years ago (Harman & Candela 1990), the inversion process would require around one day per gigabyte. The latest techniques in index construction have dramatically reduced this time, to around 8 minutes per gigabyte 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 occurrence 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 are impractically slow, but the strategy of creating temporary indexes in memory, writing them as blocks, then merging the blocks to yield the final index is highly efficient.

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, from 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 does have the disadvantages that it requires two passes over the collection and 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).

To evaluate a ranked query with an inverted index, most text retrieval systems read 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 occurs 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 types of query — such as Boolean or phrase queries — can also be resolved using an inverted index.

The techniques described here have been implemented in the LUCY text search engine, written by the Search Engine Group at RMIT.[1]. This search engine was used for all experiments described in this paper.

3 Index update strategies

For text retrieval systems, the principles of index maintenance — that is, of 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 small number of 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.) For each of these terms it is necessary to retrieve its postings list from disk, 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, naively implemented, carries unacceptable costs. On 100 gigabytes of text, the postings lists for the commonest of the indexed terms is likely to be tens of megabytes long, and the median list tens to hundreds of kilobytes. 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 amortise the costs over a series of updates.

The problem of index maintenance for text data has not been broadly investigated: there is only a little published work on how to efficiently modify an index as new documents are accumulated or existing documents are deleted or changed (Clarke, Cormack & Burkowski 1994, Cutting & Pedersen 1990, Tomasic, Garcia-Molina & Shoens 1994). This work pre-dates the major innovations in text representation and index construction that were described in the previous section.

There are several possible approaches to cost amortisation for index maintenance. One approach is to adapt the techniques used for index construction. In efficient index construction techniques, a temporary index is built in memory until space is exhausted. This temporary index is then written to disk as a *run*. When all documents have been processed, the runs are merged to give a final index. The re-merge strategy could be used for index maintenance: as new documents are processed, they are indexed in memory, and when memory is exhausted this run of new information could be merged with the existing index in a single linear pass. While the index would be unavailable for some time during the merge (tens of minutes on an index for 100 gigabytes of text), the overall cost is much lower than the naive approach. To avoid the system itself being unavailable at this time, a copy of the index can be kept in a separate file, and the new index is switched in once the merge is complete.

Update deferral using a temporary in-memory index can be used to improve the naive update strategy. Once main memory is exhausted, the postings lists on disk are individually merged with entries from the temporary index. Updating postings lists in-place still 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.

A more primitive, but still commonly used, approach to cost amortisation is to but re-build the entire index from the stored collection. This approach has a number of disadvantages, including the need to store the entire collection and that the index is not available for querying during the re-building process. Re-building is intuitively worse than the re-merge strategy, but that does not mean that it is unacceptable. Consider for example a typical 1-gigabyte university web site. A re-build might take 10 minutes — 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.

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 1992*a*, Biliris 1992*b*, Carey, DeWitt, Richardson & Shekita 1986, Carey, DeWitt, Richardson & Shekita 1989, Lehman & Lindsay 1989). However, these results are difficult to apply to text indexes: they present very different technical problems to 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.

4 Update algorithms

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

[1] Available at `http://www.seg.rmit.edu.au/lucy`

In-place. The in-place algorithm updates postings lists for each term that occurred in the new documents. The list updates are not performed in a specific order other than that imposed by the data structures used to accumulate the postings. This is almost certainly not the optimal disk access pattern, and is thus a topic for further research. Free space for the postings lists is managed using a list of free locations on the disk. These are ordered by disk location so that a binary search can be used to determine whether an existing postings list can be extended using additional free space occurring immediately after it. A first fit algorithm is used to search for free space if a postings list has to be moved to a new location or a new postings list must be created. The entire algorithm is described below.

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, discard it and advance to the next in-memory postings list.
   (c) Otherwise, determine a new disk location with sufficient space to hold the on-disk and in-memory postings lists, using a first-fit algorithm.
   (d) Read the on-disk postings list from its previous location and write it to the new location.
   (e) Append the in-memory postings list to the new location.
   (f) Discard the in-memory postings list and advance to the next.

Note that 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 organisations of postings lists, the entire existing list must be decoded. In our experiments, we test both append and (in one data set) full-decode implementations.

Re-merge. The re-merge algorithm updates the on-disk index by performing a merge between the on-disk postings and 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 order, using the hash values of the terms as the sorting key. This allows 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 posting list has a hash value 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 postings term.
   (b) Otherwise, if the in-memory posting term has a hash value 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 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.

If queries must be processed while maintenance is under way, two copies of the index must be kept, as queries cannot be resolved using the new index until it is complete. The 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 in order to reduce this redundancy. The benefit is that unlike the in-place algorithm, this ensures that lists are stored contiguously, that is, there is no fragmentation.

Re-build. The re-build algorithm discards the current index after constructing an entirely new index. The new index is built on the stored collection and the new documents added since the last update. In order to service queries during the re-building process, a copy of the 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. 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 current 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 maintenance is required. This necessitates that the entire collection be stored and re-processed in the indexing process. Moreover, existing postings are ignored.

Similarly to the re-merge algorithm, a separate copy of the index must be maintained to resolve queries during

the maintenance process. 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 main-memory is available to construct runs.

5 Experiments

Experiments were performed on a dual Pentium III 866 MHz machine, with 256 Mb of main memory on a 133MHz front side bus, and a quad Xeon 2 GHz machine with 2 GB main memory on a 400 MHz front side bus.

Two sets of experiments were run on the dual Pentium III using 1 Gb and 2.75 Gb collections taken from the TREC WT10g collection. TREC is a large-scale international collaboration intended primarily for comparison of text retrieval methods (Harman 1995), and provides large volumes of 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 first experiment with the Pentium III was to update the index of the 1 Gb collection, where an initial index on 500 Mb of data (75,366 documents) was updated with 500 Mb of new data (75,368 documents). In these experiments, we varied the size of the buffer used to hold new documents, to measure the relative efficiency of the different methods as the number of documents to be inserted in a batch was varied.

The second experiment used the 2.75 Gb collection, where an initial index on 2.5 Gb (373,763 documents) was updated with 250 Mb (39,269 documents). A smaller number of updates was used due to time constraints; at around a second per document in the slower cases, and 15 separate runs, this experiment took a week to complete.

A third experiment was run on the quad Xeon machine using 21 Gb of data, where an initial index of 20 Gb of data (3,989,496 documents) was updated with 1 Gb (192,264 documents). The data for the third experiment was taken from the TREC WT100g collection, a superset of the WT10g 1997 web crawl.

These experiments were chosen to explore the characteristics of the three strategies when operating under different conditions. In particular, we explored the behaviour of the approaches when the index fits into memory, and contrasted this with the behaviour when the index is many multiples of main-memory size. The different collection sizes were chosen to explore the maintenance cost of each of the three strategies with different amounts of data.

6 Results

The results of the timing experiments are shown in Figures 1, 2, and 3. In all experiments the machines are under light load, that is, no other significant tasks are accessing the disk or memory.

Figure 1 shows the results of the first experiment, where 500 Mb of data was added to an initial index on 500 Mb. The in-place and re-merge strategies were run with buffered numbers of documents ranging between 10 and 10,000. The re-build strategy was limited to buffering numbers of documents ranging from 100 to 10,000 due to the excessive running times required for lower numbers.[2]

The results support our intuitive assessment that the re-building strategy is less efficient than re-merging for all non-trivial scenarios. Both strategies outperform the in-place update for large document buffers, which can be attributed to the advanced index construction algorithms that underly their operation. However, their performance degrades at a faster rate than in-place update with smaller buffer sizes, to eventually become slower than in-place update. This highlights that both algorithms need to process the entire index or collection every update, even for small updates.

The in-place variant that decodes the postings lists before updating them is also shown in Figure 1. As expected, it is less efficient than the more optimised in-place strategy in all cases. List decoding is not a significant overhead for large document buffer sizes but, as buffer size decreases, the per-document overhead increases and decoding becomes impractical.

The results of the second experiment, where 250 Mb of data was added to an initial index of 2.5 Gb, are shown in Figure 2. The re-building strategy was again always worse than re-merging. All three strategies show comparable behaviour to the first experiment, but all schemes are slower because of the processing costs associated with a larger index. As discussed previously, the re-build and re-merge strategies' performance degraded faster than the in-place, making them both slower than in-place update at a buffer size of 100 documents. This is a larger buffer size than the corresponding point in Figure 1, showing that the in-place scheme has degraded less under the increased index size. In contrast to the other two strategies, the in-place algorithm only needs to process the sections of the index that it updates. However, this does not make its performance independent of the index size, as the postings lists that it manipulates have size that is proportional to the index.

Figure 3 shows the results of the third experiment, which were performed on the quad Pentium IV. Unfortunately, a lack of time prevented us from running this experiment with the re-build strategy. The results shown using the re-merge and in-place strategies are consistent with earlier results, with the re-merge scheme outperforming the in-place strategy for large document buffer sizes. These results are not directly comparable to the previous experiments, since the experiment was performed on a different machine. However, the relative performance of the strategies is comparable, and the point at which in-place becomes more efficient than re-merge is higher than in the two previous experiments. The index size in this experiment is approximately ten times the size of the index used in the second experiment, which corresponds to the relative improvement in the efficiency of the in-place algorithm. The results support the expectation that the in-place algorithm continues to work well as index size increases.

The fragmentation of the final index produced for the

[2] In these experiments and those discussed below, buffer sizes ranged up to approximately 200 Mb

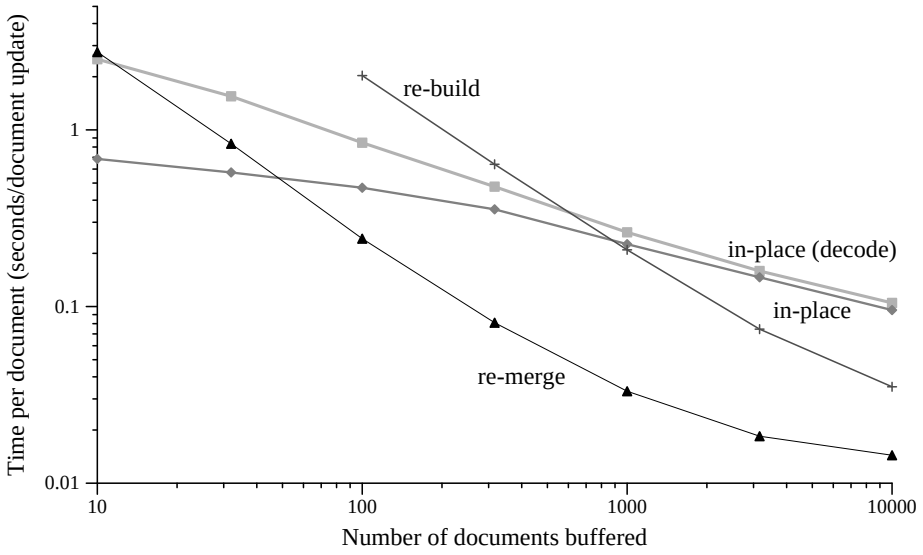

Figure 1: *Running times per input document for the three update strategies for the 1 GB collection on the dual Pentium III, for a range of buffer sizes.*

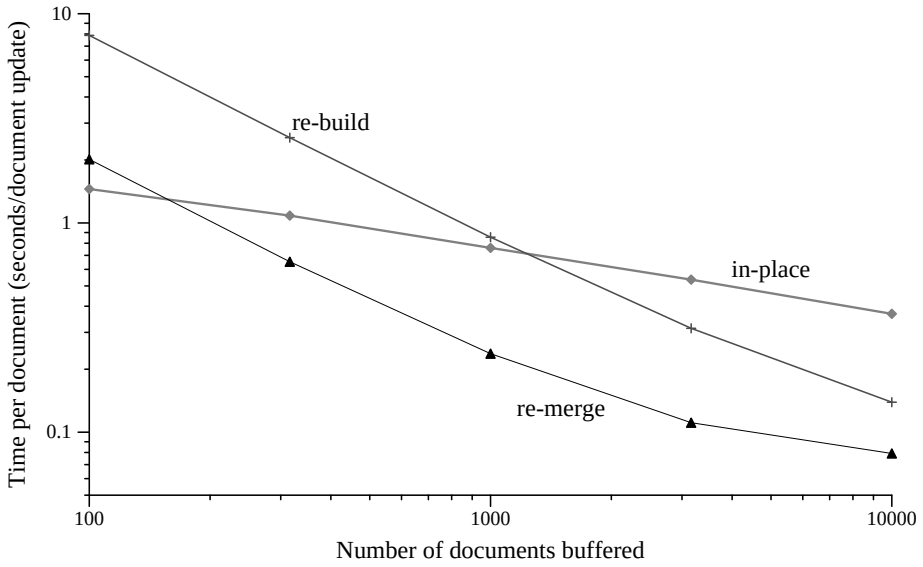

Figure 2: *Running times per input document for the three update strategies for the 2.75 GB collection on the dual Pentium III, for a range of buffer sizes.*

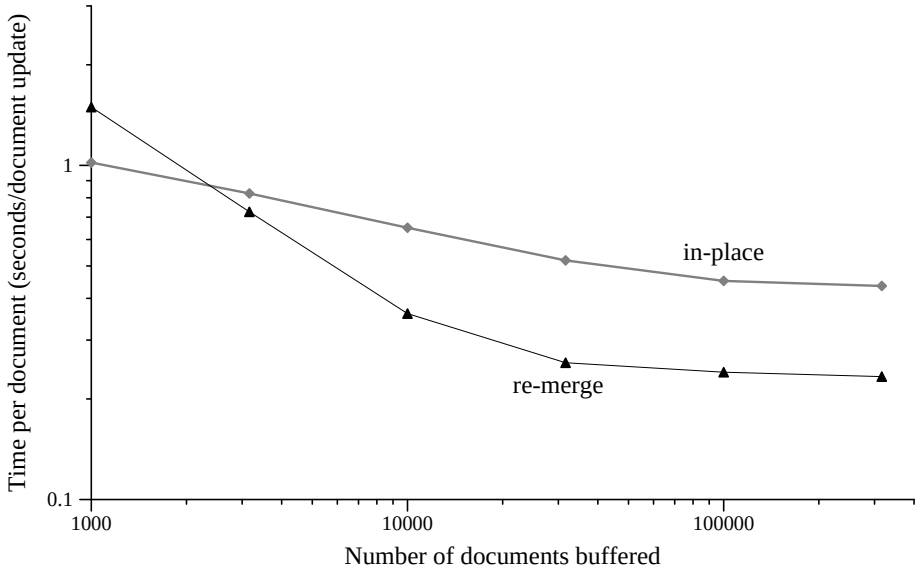

Figure 3: *Running times per input document for the three update strategies for the 21 GB collection on the quad Xeon, for a range of buffer sizes.*

in-place strategy in each of the experiments is shown in Figure 4. (Note that the results from different experiments are not directly comparable to each other, due to the differing sizes of the collections.) The figures shown are fragmentation as a percentage of the total space used to hold the postings lists. The high fragmentation suggests that space management is a significant problem in implementing the in-place strategy.

To explore fragmentation of the index during maintenance, the fragmentation was sampled after every update of the on-disk postings for the 1 Gb collection. Document buffer sizes 200, 1000 and 5000 were plotted, with the results shown in Figure 5. These results indicate that after an initial period where the fragmentation rapidly rises after updating the initially-packed organisation of the postings lists, the fragmentation remains relatively stable. Interestingly, the degree of fragmentation in the index appears to be related to the size of the updates applied to it, not to the number of updates. The results in Figure 4 indicate that this relationship also depends on the size of the existing index, indicating that the size of the updates applied to the index relative to the size of the index may be a key factor in the level of fragmentation.

The oscillation that can be observed in the fragmentation results is due to the movement of large postings lists. Postings lists for frequently occurring words such as "the" are large, and 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. Fragmentation then jumps 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 fragmentation to fall, and can continue until another large postings list needs to be relocated to the end of the index, starting the process again.

7 Conclusions

Inverted indexes are data structures that support querying in applications as diverse as web search, digital libraries, application help systems, and email searching. Their structure is well-understood, and their construction and use in querying has been an active research area for almost fifteen years. However, despite this, there is almost no publically-available information on how to maintain inverted indexes when documents are added, changed, or removed from a collection.

In this paper, we have investigated three strategies for inverted index update: first, an in-place strategy, where the existing structure is added to and fragmentation occurs; second, a re-merge strategy in which new structures are merged with the old to create a new index; and, last, a re-build strategy that entirely re-constructs the index at each update. We have experimented with these three approaches using different collection sizes, and by varying the number of documents that are buffered in main-memory before the update process.

Our results show that when reasonable numbers of documents are buffered, the re-merge strategy is fastest. This result is largely because the index fragments under the in-place strategy, necessitating frequent reorganisation of large parts of the index structure and rendering it less efficient. 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.

We believe that optimisation 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 new and largely unoptimised. 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

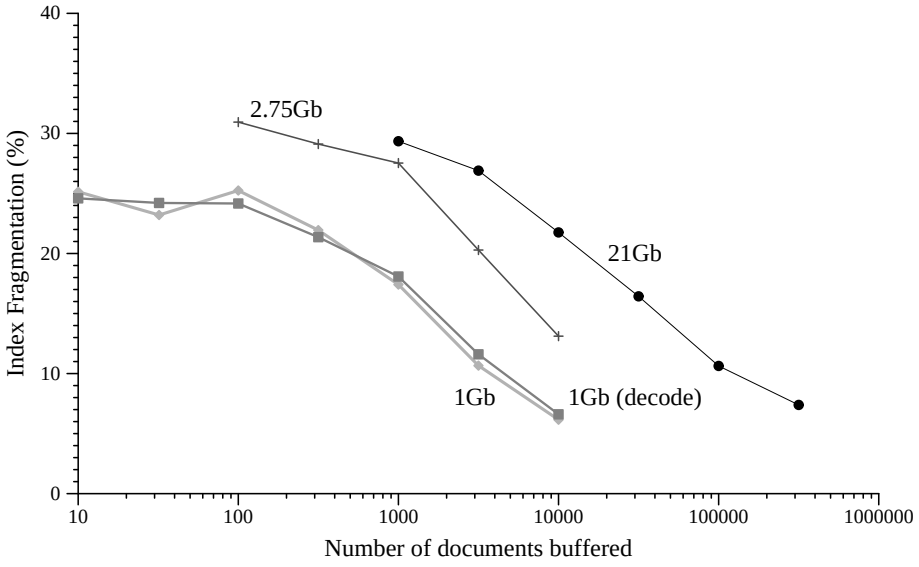

Figure 4: *Index fragmentation of the in-place strategies for all experiments. Note that curves for different collections are not directly comparable.*

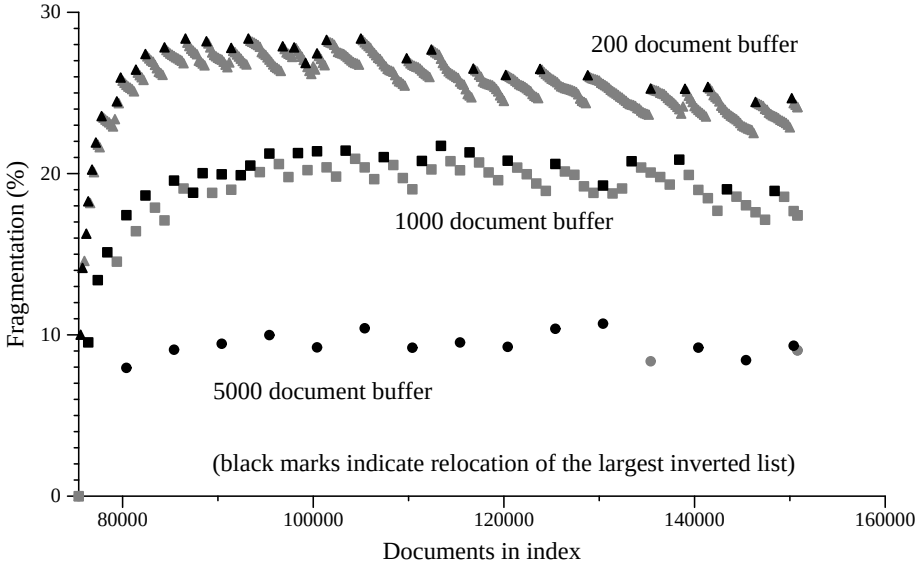

Figure 5: *Index fragmentation of the in-place strategy on the 1 Gb collection during the maintenance process.*

reduce overall costs.

Acknowledgements

This work was supported by the Australian Research Council. We are grateful for the comments of two anonymous reviewers.

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, "Proc. ACM-SIGIR Int. Conf. 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, "Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval", Tampere, Finland, pp. 3–10.

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

Bahle, D., Williams, H. E. & Zobel, J. (2002), Efficient phrase querying with an auxiliary index, *in* K. Järvelin, M. Beaulieu, R. Baeza-Yates & S. H. Myaeng, eds, "Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval", Tampere, Finland, pp. 215–221.

Biliris, A. (1992*a*), An efficient database storage structure for large dynamic objects, *in* F. Golshani, ed., "Proc. IEEE Int. Conf. on Data Engineering", IEEE Computer Society, Tempe, Arizona, pp. 301–308.

Biliris, A. (1992*b*), The performance of three database storage structures for managing large objects, *in* M. Stonebraker, ed., "Proc. ACM-SIGMOD Int. Conf. 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, "Proc. Int. Conf. on Very Large Databases", Morgan Kaufmann, Kyoto, Japan, pp. 91–100.

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", Addison-Wesley Longman, New York, pp. 341–369.

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.

Cutting, D. R. & Pedersen, J. O. (1990), Optimizations for dynamic inverted index maintenance, *in* J.-L. Vidick, ed., "Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval", ACM, Brussels, Belgium, pp. 405–411.

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

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 & Management* **31**(3), 271–289.

Harman, D. & Candela, G. (1990), "Retrieving records from a gigabyte of text on a minicomputer using statistical ranking", *Jour. 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)", National Institute of Standards and Technology Special Publication 500-246, Gaithersburg, MD, pp. 91–104.

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

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, "Proc. Int. Conf. on Very Large Databases", Amsterdam, The Netherlands, pp. 375–383.

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", *Jour. 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* "Proc. Text Retrieval Conf. (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, "Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval", Tampere, Finland, pp. 222–229.

Tomasic, A., Garcia-Molina, H. & Shoens, K. (1994), Incremental updates of inverted lists for text document retrieval, *in* "Proc. ACM-SIGMOD Int. Conf. on the Management of Data", ACM, Minneapolis, Minnesota, pp. 289–300.

Witten, I. H., Moffat, A. & Bell, T. C. (1999), *Managing Gigabytes: Compressing and Indexing Documents and Images*, second edn, Morgan Kaufmann, San Francisco, California.

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