A Hybrid Approach to Index Maintenance in Dynamic Text Retrieval Systems

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

Abstract. In-place and merge-based index maintenance are the two main competing strategies for on-line index construction in dynamic information retrieval systems based on inverted lists. Motivated by recent results for both strategies, we investigate possible combinations of in-place and merge-based index maintenance. We present a hybrid approach in which long posting lists are updated in-place, while short lists are updated using a merge strategy. Our experimental results show that this hybrid approach achieves better indexing performance than either method (in-place, merge-based) alone.

1 Introduction

Traditional information retrieval systems deal with static text collections: Once indexed, no documents are ever added to or removed from the collection. Efficient index construction for static collections has been studied in detail over the last two decades. After contributions by Moffat and Bell [8, 9] and Heinz and Zobel [13, 5], the indexing problem for static collections seems solved. Following an inverted-file approach and combining the techniques described in the literature, it is possible to index text collections at a rate well above 50 GB per hour on a standard desktop PC, allowing the indexing of text collections in the terabyte range on a single PC (see section 4 for details).

For on-line index maintenance in dynamic search environments, the situation is different. Because index updates are interleaved with search queries, it is important that the index is *always* kept in a form that allows for efficient query processing. Traditional batch index construction techniques do not meet this criterion, as they do not make any guarantees about the contiguity of on-disk inverted lists during the index construction process. If queries have to be processed during index construction, this non-contiguity leads to a large number of disk seek operations and thus poor query processing performance.

In a truly dynamic environment, documents may be added to and removed from the collection at any point in time. For the purpose of this paper, we disregard deletions and focus exclusively on document insertions. It is possible, however, to integrate support for deletions into the methods described in this paper (see [1], for example).

Many index maintenance strategies that deal with document insertions in dynamic search systems have been examined in the past. The two dominant families are *in-place* update and *merge-based* index maintenance. When new documents are added to the collection, in-place strategies update the index by adding new postings at the end of existing on-disk inverted lists, relocating the lists if necessary. Merge-based strategies update the index by merging the existing on-disk inverted file with the new postings, resulting in a new inverted file that replaces the old one.

The main disadvantage of merge-based update is that, whenever an on-disk inverted file is updated, the entire file has to be read/written, even though most parts of the index remain unchanged and only a relatively small number of postings are added to the file. For large indices, this leads to a substantial decrease in index update performance. In-place strategies try to overcome this problem by leaving a certain amount of free space at the end of every on-disk inverted list. If, during an index update, there is enough space for the new postings, they are simply appended to the existing list. If not, the entire list has to be moved to a new location in order to make enough room for the new postings. These relocations make it impossible to keep the on-disk lists in any given order (e.g., lexicographical order), as the order is destroyed every time a list is relocated. This leads to a large number of non-sequential disk accesses during index updates, the main shortcoming of in-place update strategies.

We propose a hybrid approach to index maintenance, based on the idea that for long lists it takes more time to copy the whole list than to perform a single disk seek operation, while for short lists a disk seek is more expensive than copying the list as part of a longer, sequential read/write operation. Our approach exhibits an amortized indexing performance superior to that of existing merge-based strategies, while providing an equivalent or even slightly better level of query processing performance.

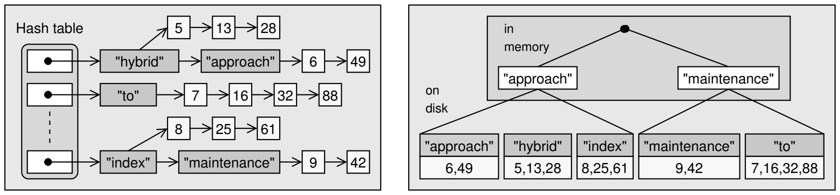
The remainder of this paper is organized as follows: The next section gives an overview of related work, divided into techniques for off-line and on-line index construction; the on-line part covers both merge-based and in-place strategies. In section 3, we present our hybrid approach to index maintenance, explain how in-place updates are realized, and which merge strategies are used for the merge part of the hybrid update. We present an experimental evaluation in section 4 and compare our new approach to existing merge-based maintenance strategies. The evaluation is done in terms of both indexing and query processing performance.

2 Related Work

This section gives an overview of existing work on index construction techniques for retrieval systems based on inverted lists. We first cover the case in which a static collection is indexed, using an off-line method, and then explain how the off-line construction method can be adapted to deal with dynamic environments.

2.1 Off-Line Index Construction

Inverted files have proved to be the most efficient data structure for high performance indexing of large text collections [14]. For every term that appears in



(a) In-memory index with hash table and linked lists. (b) On-disk index with 2-level search tree.

Fig. 1. Basic structure of in-memory index and on-disk inverted file. Vocabulary terms in memory are arranged in a hash table, using linked lists to organize the individual posting lists. Terms on disk are sorted in lexicographical order, allowing for efficient index access at query time.

the given collection, the inverted file contains a list of all positions at which the term occurs (the term's *posting list*).

The process of creating this inverted file can be roughly described as follows: Input documents are read, one at a time, and postings are accumulated in an in-memory index, using a hash table with move-to-front heuristics [13] to look up vocabulary terms. Postings for the same term are stored in memory in compressed form, either in an augmentable bitvector [5] or in a linked list [2]. When the entire collection has been indexed, all terms are sorted in lexicographical order and the in-memory data are written to disk, forming an inverted file. In order to decrease disk I/O, the data in the inverted file are stored in compressed form. A two-level search tree is used to access on-disk posting lists during query processing later on. The general structure of the in-memory and the on-disk index can be seen in Figure 1.

If the amount of main memory available is not sufficient to index the whole collection at a single blow, the process is repeated several times, each time creating an inverted file for a part of the total collection. The size of these subcollections depends on the available main memory. It is determined on-the-fly during the indexing process and does not require multiple passes over the collection.

After the whole collection has been indexed, all sub-indices created so far are brought together through a multi-way merge process, resulting in the final index. Since the posting lists in the sub-indices are stored in lexicographical order, this can be done very efficiently by organizing the input indices in a priority queue and employing standard input buffering techniqes (read-ahead). For a given term, its final posting list is created by concatenating the sub-lists from the individual sub-indices. This does not require the decompression of the postings and thus allows for a very efficient creation of the final index.

2.2 On-Line Index Construction

It is possible to use the same techniques employed for off-line index construction in a dynamic retrieval system in which update operations are interleaved with search queries. Whenever a query is being processed and a posting list has to be fetched from the (incomplete) index, sub-lists are retrieved from the existing on-disk indices and the in-memory index. By concatenating these sub-lists, the whole posting list is constructed, and the query can be processed (shown in Figure 2). We refer to this method as the *No Merge* strategy.

Although, in principle, *No Merge* does solve the problem of dealing with a dynamic text collection, it is not a good solution. Because the point at which the whole collection has been indexed is never reached in a dynamic environment, the final index is never created and all the on-disk inverted files have to be queried individually. Since the number of these files can be quite large, this severely harms query processing performance, as it requires a large number of disk seeks – at least one per inverted file.

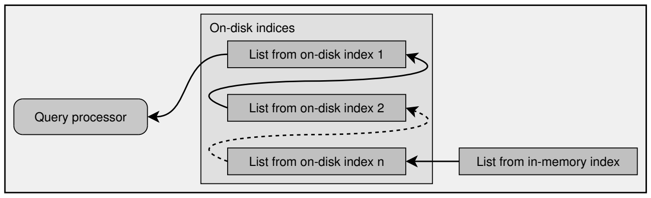

On-disk indices
List from on-disk index 1
List from on-disk index 2
List from on-disk index n
Query processor
List from in-memory index

Fig. 2. Query processing during batched index construction. A term's posting list is constructed by concatenating sub-lists from all on-disk indices and the in-memory index containing the postings for the sub-index currently being created.

In order to avoid these disk seeks, the on-disk indices should be re-organized in some way whenever a new sub-index is created. Unfortunately, this requires additional disk operations and can be quite time-consuming. Büttcher and Clarke [1] characterize this as a trade-off between indexing performance and query processing performance; maximum indexing performance is obtained by following the off-line index construction strategy described above, while query performance is optimized by immediately merging the in-memory postings with the existing on-disk index, creating a new, optimized on-disk index whenever main memory is exhausted. Depending on the ratio of document insertions and search queries, different strategies lead to optimal overall system performance.

In-Place Index Maintenance

In-place index update techniques were examined very early in the history of dynamic information retrieval systems. In general, two different flavors of in-place update exist: those that keep each posting list in a contiguous part of the on-disk index and those that allow posting lists to be split up by index update operations. Keeping posting lists in a contiguous part of the inverted file maximizes query performance, but requires frequent relocations of most lists in the index. If lists have to be kept contiguous, and no free space is reserved at the end of an existing list, it has to be relocated every time an update is applied to the index. Preallocation strategies, such as predictive overallocation [10], reduce the number of times a posting list has to be relocated, but do so at the cost of increased disk space consumption.

Tomasic, Shoens, and García-Molina [12, 11] present a very thorough discussion of in-place index maintenance and propose a system that is based on the distinction between short and long posting lists, storing short lists in fixed-size buckets together with other short lists and long lists in (not necessarily contiguous) sections of the inverted file.

The main problem of in-place strategies is the very high number of disk seeks necessary to update an inverted file. Relocating inverted lists means destroying the original ordering of the lists on disk. This has two important implications:

1. Accessing the index terms in a pre-defined order (e.g., in lexicographical order) does not lead to a sequential disk access pattern during index updates and thus requires a large number of disk seek operations.
2. Since on-disk index terms are unordered, a second index structure is necessary, mapping terms to the positions of their posting lists. For large collections, this map cannot be kept in memory, but has to be stored on disk.

Because random disk accesses are very slow, compared with sequential access patterns, in-place strategies might reduce the number of read/write operations necessary to perform an index update, but this is not really reflected by their effective indexing maintenance performance.

One possible implementation of the in-place strategy is to keep each posting list in a separate file inside the file system. Whenever in-memory postings have to be combined with the on-disk index, the respective file is opened and the in-memory postings are appended to the existing on-disk list. Fragmentation management etc. are deferred to the file system. This is the implementation we are using in our experiments.

Merge-Based Index Maintenance

Merge-based update strategies share the common idea that disk read/write operations are most efficient when they are carried out in a sequential manner, minimizing disk head movement. The most popular form of merge update is the *Immediate Merge* strategy: Whenever main memory is exhausted, the in-memory postings are merged with the existing on-disk index, processing the on-disk index in a sequential fashion, resulting in a new on-disk index that immediately replaces the old one. Since, at any given time, there is only a single active on-disk index, this strategy minimizes the number of disk head movements necessary to fetch a posting list from disk and thus maximizes query processing performance.

The Immediate Merge strategy was first described by Cutting and Pedersen [4]. In their work, they present an index structure based on a B-tree and give a comparison of leave node caching (a form of in-place update) and index remerging. Using the same amount of memory, the merge approach exhibits vastly superior performance. Lester et al. [7] conducted similar experiments, comparing Immediate Merge with more recently developed in-place strategies, and obtained similar results, indicating that merge-based index maintenance is usually more efficient than in-place update.

The main problem of Immediate Merge is that, whenever main memory is exhausted, the entire on-disk index has to be re-processed. Hence, after N tokens have been added to the collection, the total number of postings that have been transferred from/to disk is:

$$\sum_{i=1}^{\lfloor \frac{N}{M} \rfloor} (2i-1) \cdot M \;\in\; \Theta\left(\frac{N^2}{M}\right),$$

where M is the available main memory (the number of postings that can be kept in memory at a time). This quadratic time complexity renders Immediate Merge infeasible for text collections much larger than the available main memory.

Recently, Büttcher and Clarke [1] and Lester et al. [6] have proposed merge-based update strategies that do not share this shortcoming. By allowing a controlled number of on-disk indices to exist in parallel, indexing efficiency is greatly increased, while query processing performance remains almost unchanged compared to the Immediate Merge strategy. The basic idea is to maintain a set of sub-indices of exponentially growing size. If M is the amount of main memory available to the indexing system, then on-disk indices can be of size M, $k \cdot M$, $k^2 \cdot M$, …, for some small value of k (usually, $k = 2$ or $k = 3$ is chosen). For any given index size, there can be at most $k - 1$ indices of that size at the same time. Whenever the creation of a new on-disk index leads to a situation where there are k indices of the same size $k^n \cdot M$, they are merged into a new index of size $k^{n+1} \cdot M$. This process is repeated until, for every integer n, there are at most $k - 1$ indices of the same magnitude $k^n \cdot M$.

This strategy is referred to as *geometric partitioning* or *Logarithmic Merge*. At any given time, the number of on-disk sub-indices is bounded by $\left\lceil \log_k \left(\lfloor \frac{N}{M} \rfloor \right) \right\rceil$, where N again is the number of input tokens processed so far. The total number of postings that have been transferred from/to disk is $\Theta\left(N \cdot \log\left(\frac{N}{M}\right)\right)$. Limiting the number of sub-indices to the logarithm of the collection size keeps query performance at a very high level, but allows the retrieval system's indexing complexity to decrease dramatically from $\Theta(N^2)$ to $\Theta(N \cdot \log(N))$ disk operations. However, although this is a great improvement, Logarithmic Merge shares the same basic problem of all merge-based maintenance strategies: When a new sub-index is created, posting lists (or parts thereof) have to be transferred from/to disk even though they have not been changed.

3 A Hybrid Approach to Index Maintenance

In the previous section, we have discussed the advantages and disadvantages of merge-based and in-place index update schemes. In merge-based index maintenance strategies, especially for Immediate Merge, a great amount of time is spent copying postings from an old index to the new one. In-place strategies avoid this overhead by leaving unchanged portions of a posting list untouched during an index update operation, at the cost of many, possibly too many, disk seek operations.

It seems obvious that there is a certain system-specific number X such that for posting lists shorter than X postings it is faster to read the list (as part of a longer, sequential read operation) than to perform a disk seek, while for lists longer than X a disk seek is faster than reading the list. In other words, a hybrid index maintenance strategy, in which short lists are maintained using a merge-based update method, while long lists are updated in-place, seems promising. It can be expected that such a method performs better than either pure merge-based or pure in-place index maintenance.

We first explain how in-place updates are realized in our retrieval system and then present two different hybrid strategies. The first is a combination of in-place update and Immediate Merge. The second is a combination of in-place update and Logarithmic Merge.

3.1 In-Place and the File System

Some of the optimizations for the in-place index update scheme, such as pre-allocation strategies, have been used in file systems for a long time. Other techniques, such as keeping very short posting lists inside dictionary data structure instead of allocating separate space in the index file, are more recent, but have their analoga in file system implementations as well: The Reiser4[1] file system stores very small files within the directory tree itself, avoiding the overhead that is associated with allocating a separate block for every file.

Since implementing an in-place strategy and fine-tuning it so that it delivers good indexing performance is a time-consuming task, we chose to realize the in-place portion of our hybrid strategy by using existing file system services. Whenever we refer to an in-place-updatable posting list in the remainder of this paper, this means that the list is stored in an individual file. Updates to the list are realized by appending the new postings to the existing file data.

We rely on the file system's ability to avoid relocations and fragmentation by using advanced pre-allocation strategies and hope that several decades of file system research have resulted in file system implementations that are no worse at this than a custom implementation of the in-place index update scheme. Since in our hybrid approach only long lists are updated in-place, this seems like a reasonable assumption.

3.2 In-Place + Immediate Merge

The combination of in-place update and Immediate Merge is straightforward. Whenever a new on-disk index is created by merging the in-memory data with the existing on-disk index, terms are divided into two categories:

1. Terms whose posting list contains less than X postings (*short lists*).
2. Terms whose posting list contains at least X postings (*long lists*).

The merge process is performed as usual. The only modification is that, whenever a term with a long list is encountered during the merge, its postings are appended to a file that contains the postings for that term (the file is created if it does not exist yet) instead of being added to the new index that results from the merge process. Short lists do not have their own files. They are added to the new index, following the standard Immediate Merge strategy.

[1] http://www.namesys.com/v4/v4.html

3.3 In-Place + Logarithmic Merge

Integrating in-place update into the Logarithmic Merge strategy is slightly more complicated than the combination with Immediate Merge. This is for two reasons:

- Most sub-index merge operations do not involve all existing on-disk indices, but only a subset. Therefore, the total number of postings for a given term is unknown at merge time.
- Even if we always knew the the total number of postings for a term, it is not clear what implications the total size of a term's posting list has for a merge operation that only involves a small part of that list.

We address these problems by choosing a very conservative strategy. When main memory is exhausted for the very first time and the first on-disk inverted file is created, the predefined long list threshold value X is used to divide the posting lists of all terms into short lists and long lists. Once this decision has been made, it will never be changed. From the indexing system's point of view, a posting list that does not make it into the set of long lists when the first index is created will always remain a short list, regardless of how many postings it actually contains.

Clearly, this solution is not optimal, as it does not take into account possible changes in the term distribution that take place after the first on-disk index has ben created. More adaptive strategies fall into the category "future work".

3.4 Partial Flush

The main rationale behind our hybrid approach is to avoid the unnecessary disk transfers that are caused by reading/writing unchanged portions of posting lists during merge operations. Therefore, it makes sense to defer the merge part of the hybrid update for as long as possible. This is achieved by the *partial flush* strategy: When main memory is exhausted and postings have to be transferred to disk, the indexing system only writes those postings to disk that belong to long lists (i.e., postings whose list resides in an individual file and is updated in-place). If, by doing so, the total memory consumption of the indexing system is decreased below a certain threshold, the system does not perform the merge part of the update, but continues its normal operation. Only if the total memory consumption cannot be decreased below the predefined threshold value, the merge part is performed and all postings are transferred from memory to disk.

For our experiments, we set the partial flush threshold value to 85% of the available memory. Depending on the value of the long list threshold X, this made the system perform 2-3 in-place update sequences per re-merge operation.

4 Experimental Evaluation

In this section, we give an experimental evaluation of our hybrid index maintenance strategy. We compare it with the *Immediate Merge* and *Logarithmic Merge* strategies, paying attention to both index maintenance and query processing performance. We also compare it with *No Merge* (the dynamic variant

	No Merge	Imm. Merge	Log. Merge	Hybrid (IP+LM)
Total indexing time	4h01m	40h14m	7h14m	6h08m
Indexing throughput	106.1 GB/h	10.6 GB/h	59.0 GB/h	69.5 GB/h
Average time per query	5.028 sec	2.840 sec	3.011 sec	2.940 sec

Table 1. Comparing indexing and query processing performance of the off-line method, Immediate Merge, Logarithmic Merge, and the best hybrid strategy (In-Place + Log. Merge). The indexing time for the No Merge strategy does not include the final merge operation that is part of the off-line method. Including the final merge, the total indexing time is 5h30m (77.6 GB/h).

of off-line index construction, described in section 2.2) and show that the hybrid update strategy achieves indexing performance very close to that of the off-line method.

4.1 Experimental Setup

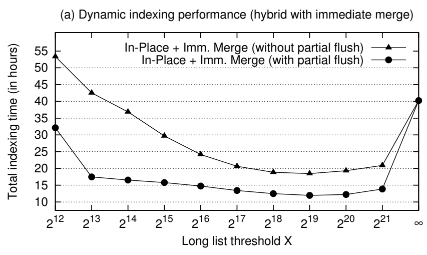
For our experiments, we employed the GOV2 collection used in the TREC Terabyte track [3]. GOV2 contains 25.2 million documents with a total uncompressed size of about 426 GB. The collection was indexed using 1024 MB of main memory for the in-memory index. As query set, we used 100 Okapi BM25 queries derived from the topics of the 2004 and 2005 TREC Terabyte ad-hoc retrieval tasks. After stopword removal, the average query length was 3.0 terms.

For all experiments, the same update/query sequence was used. It contained 1360 search queries, interleaved with update commands (document insertions). Queries were randomly drawn from the query set described above, with each query occurring 13-14 times in the entire sequence. No caching is performed by the search engine itself. Since the average amount of data read from the input documents between two consecutive search queries is approximately 325 MB, the entire content of the file system cache is replaced after about 6 queries. This means that cache effects are negligible, and repeating each query 13-14 times does not affect the experimental results.

The experiments were conducted on a PC based on an AMD Athlon64 3500+ (2.2 GHz) with a 7200-rpm SATA hard drive. The input files were read from a RAID-0 built on top of two 7200-rpm SATA drives.

4.2 Results

Our first series of experiments consists of indexing the entire collection using the *No Merge* strategy and two pure merge-based strategies (1. Immediate Merge; 2. Logarithmic Merge for base $k = 2$). The results (reported in Table 1) are consistent with earlier findings [6, 1] and show that the off-line method really should not be used in a dynamic environment. Logarithmic merge exhibits indexing performance close to off-line index construction and query processing performance close to Immediate Merge.



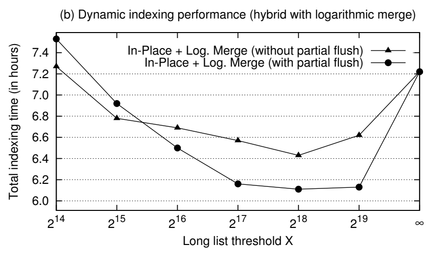


Fig. 3. Dynamic indexing performance of the two hybrid index maintenance strategies (In-Place + Immediate Merge and In-Place + Logarithmic Merge) with different long list threshold values X. The rightmost data points ($X = \infty$) represent pure Immediate Merge and pure Logarithmic Merge, respectively.

The next series of experiments evaluates the indexing performance of the two hybrid strategies described in the previous section. The long list threshold X, used to determine which posting lists are updated in-place and which are maintained by the merge strategy, is varied in order to study the effects of different values and to find the optimal configuration. The results presented in Figure 3 show that hybrid index maintenance achieves a huge improvement over the pure merge-based methods. For the hybridization of Immediate Merge (Figure 3a), the time needed to index the whole collection can be reduced by roughly 50% if X values between 2^{18} and 2^{20} are used to realize the split. With partial flush enabled, the improvement is even greater; the total indexing time drops by 70% – from 40 hours to just under 12 hours.

As expected, the improvements achieved by hybridizing Logarithmic Merge are not as dramatic as in the case of Immediate Merge. Still, our experimental results (depicted in Figure 3b) show that the combination of in-place update and Logarithmic Merge reduces the total indexing time by 11% (15% with partial flush), compared with pure Logarithmic Merge. For X around 2^{18} and partial flush turned on, the hybrid Logarithmic Merge indexes the whole collection in 367 minutes, only 37 minutes slower than off-line index construction. This represents an indexing throughput of 69.5 GB/h.

Since all these improvements are worthless unless the hybrid strategies exhibit query processing performance similar to their pure merge-based counterparts, we also measured average query time for both hybrid strategies and different threshold values X. The average query times depicted in Figure 4 show that both hybrid Immediate Merge and hybrid Logarithmic Merge exhibit query processing performance very close to the pure merge strategies. Figure 4a indicates that the large number of in-place-updatable lists associated with small X values overburdens the file system. Fragmentation increases, and as a consequence query performance drops. For hybrid Immediate Merge, greater X values mean less fragmentation and thus better query performance. The effect that file fragmentation has on query processing performance is quite substantial. For $X = 2^{10}$, average query time increases by 25%, from 2.84 seconds to 3.55 seconds. The lowest query time is achieved when $X = \infty$ (pure Immediate Merge).

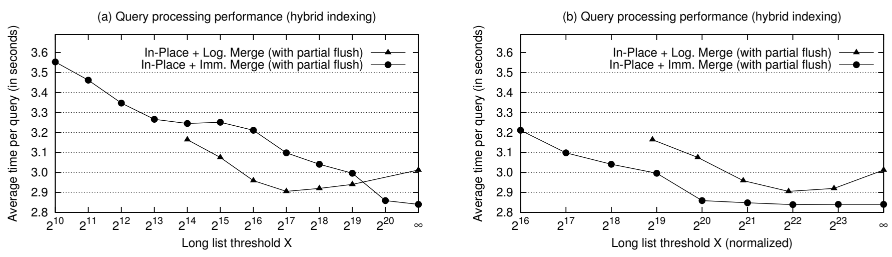


Fig. 4. Query processing performance of the two hybrid index maintenance strategies (In-Place + Immediate Merge and In-Place + Logarithmic Merge) with different long list threshold values X. The rightmost data points ($X = \infty$) represent pure Immediate Merge and pure Logarithmic Merge, respectively. Small X values lead to high file fragmentation and thus poor query performance.

For the hybridized Logarithmic Merge, the situation is different. Threshold values between 2^{16} and 2^{20} actually increase query performance (compared to pure Logarithmic Merge) because in this interval the negative effect of file fragmentation is smaller than the impact of having to fetch partial posting lists from different on-disk indices, as it is the case for Logarithmic Merge. This leads to a query time reduction of up to 4% ($X \approx 2^{17}$).

The reader might be surprised that the query times reported in Figure 4a are lower for Logarithmic Merge than for Immediate Merge. The reason for this is that in the case of Logarithmic Merge, the X value is used to split up short lists and long lists when the first on-disk index is created, while with Immediate Merge it is possible for a list initially classified as short to change its status to *long* later on. This asymmetry is taken into account in Figure 4b in which the X values for hybrid Logarithmic Merge are adjusted, pretending the decision whether a list is long or short is made after 50% of the collection has been indexed. Figure 4a shows the expected situation, where Logarithmic Merge has slightly higher query times than Immediate Merge – due to the greater number of disk seeks.

5 Conclusion & Future Work

We have presented a novel family of index maintenance strategies to be used in dynamic information retrieval systems. These strategies are combinations of merge-based and in-place index maintenance methods and offer better indexing performance than either in-place or merge-based index maintenance alone, while providing an equivalent level of query processing performance.

In our experiments, using optimal parameter settings, the combination of in-place update and Logarithmic Merge achieved an indexing throughput of 69.5 GB/h on a typical desktop PC – only 10% less than the 77.6 GB/h of our off-line index construction method. This demonstrates that on-line index construction, in which update operations and search queries are interleaved, can be performed very efficiently and almost as fast as the traditional batched index construction.

One of the shortcomings of our results is that the work being done within the file system is completely invisible to us. We do not know the preallocation strategy used by the file system, we do not know how much internal fragmentation there is in the individual files, and we do not know how often files are relocated in order to avoid fragmentation. The main focus of our future work in this area will be to investigate other implementations of hybrid index maintenance that take these issues into account.

References

1. S. Büttcher and C. L. A. Clarke. Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems. In *Proc. of the 14th ACM Conf. on Information and Knowledge Management (CIKM 2005)*, Bremen, Germany, 2005.
2. S. Büttcher and C. L. A. Clarke. Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems. UW-TR-CS-2005-32. Technical report, University of Waterloo, Canada, October 2005.
3. C. Clarke, N. Craswell, and I. Soboroff. The TREC Terabyte Retrieval Track. *SIGIR Forum*, 39(1):25–25, 2005.
4. D. R. Cutting and J. O. Pedersen. Optimization for Dynamic Inverted Index Maintenance. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1990)*, pages 405–411, New York, USA, 1990. ACM Press.
5. S. Heinz and J. Zobel. Efficient Single-Pass Index Construction for Text Databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.
6. N. Lester, A. Moffat, and J. Zobel. Fast On-Line Index Construction by Geometric Partitioning. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005)*, Bremen, Germany, 2005.
7. N. Lester, J. Zobel, and H. E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *CRPIT '26: Proceedings of the 27th Conference on Australasian Computer Science*, pages 15–23. Australian Computer Society, Inc., 2004.
8. A. Moffat. Economical Inversion of Large Text Files. *Computing Systems*, 5(2):125–139, 1992.
9. A. Moffat and T. C. Bell. In-Situ Generation of Compressed Inverted Files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.
10. W.-Y. Shieh and C.-P. Chung. A Statistics-Based Approach to Incrementally Update Inverted Files. *Inf. Processing and Management*, 41(2):275–288, 2005.
11. K. A. Shoens, A. Tomasic, and H. García-Molina. Synthetic Workload Performance Analysis of Incremental Updates. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1994)*, pages 329–338, 1994.
12. A. Tomasic, H. García-Molina, and K. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *Proceedings of the 1994 ACM SIGMOD Conference*, pages 289–300, New York, USA, 1994. ACM Press.
13. J. Zobel, S. Heinz, and H. E. Williams. In-Memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letters*, 80(6), 2001.
14. J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted Files versus Signature Files for Text Indexing. *ACM Trans. on Database Systems*, 23(4):453–490, 1998.