Hybrid Index Maintenance for Contiguous Inverted Lists[*]

Stefan Büttcher and Charles L. A. Clarke

December 4, 2007

Abstract

Index maintenance strategies employed by dynamic text retrieval systems based on inverted files can be divided into two categories: merge-based and in-place update strategies. Within each category, individual update policies can be distinguished based on whether they store their on-disk posting lists in a contiguous or in a discontiguous fashion. Contiguous inverted lists, in general, lead to higher query performance, by minimizing the disk seek overhead at query time, while discontiguous inverted lists lead to higher update performance, requiring less effort during index maintenance operations.

In this paper, we focus on retrieval systems with high query load, where the on-disk posting lists have to be stored in a contiguous fashion at all times. We discuss a combination of re-merge and in-place index update, called Hybrid Immediate Merge. The method performs strictly better than the re-merge baseline policy used in our experiments, as it leads to the same query performance, but substantially better update performance. The actual time savings achievable depend on the size of the text collection being indexed; a larger collection results in greater savings. In our experiments, variations of Hybrid Immediate Merge were able to reduce the total index update overhead by up to 73% compared to the re-merge baseline.

[*]This work extends upon the previous publications "A hybrid approach to index maintenance in dynamic text retrieval systems" (S. Büttcher and C.L.A. Clarke, Proceedings of the 28th European Conference on Information Retrieval, London, UK, 2006) and "Hybrid index maintenance for growing text collections" (S. Büttcher, C.L.A. Clarke, and B. Lushman, Proceedings of the 29th ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, USA, 2006).

1 Introduction

The fundamental data structure of most text search engines is the *inverted index*, also known as *inverted file* (Zobel and Moffat, 2006). An inverted index consists of two principal components: the *dictionary* and the *posting lists* (also known as *inverted lists*). The dictionary provides a lookup interface that can be used to quickly locate the posting list for a given term. A term's posting list is a list of all its occurrences within the text collection for which the index has been created (each such occurrence is called a *posting*). A search engine processes incoming keyword queries by locating the query terms' posting lists and combining the information found in these lists, perhaps according to Boolean query constraints, or according to probabilistic retrieval methods, such as Okapi BM25 (Robertson et al., 1998).

The problem of constructing an inverted index from a given, static text collection has been studied extensively over the last two decades (cf. Moffat, 1992; Moffat and Zobel, 1994; Moffat and Bell, 1995; Heinz and Zobel, 2003) and is well understood. The dynamic case, however, where the search engine has to update an existing index, reacting to changes in the underlying text collection, has received some attention lately (cf. Cutting and Pedersen, 1990; Tomasic et al., 1994; Shieh and Chung, 2005; Lester et al., 2005; Lester et al., 2006), but is not yet fully understood.

Many applications, such as digital libraries, Internet news search engines, and file system search engines, have to deal with continually changing text collections and sometimes even have to meet real-time (or near-real-time) performance requirements. A news search engine, for instance, would lose much of its appeal to the users if it only updated its index once per week. To keep the users happy, the engine must react to changes in the text collection by updating its index data structures, and must do so in a time-efficient manner.

Different text collections exhibit different update patterns. A fully dynamic text collection, in general, allows the following three types of update operations:

- INSERT: a new document is added to the collection;
- DELETE: an existing document is removed from the collection;
- MODIFY: an existing document is modified (e.g., some text is appended at the end).

A search engine maintaining an inverted index for such a dynamic text collection must update its internal index structures whenever it detects (or receives notification about) a change to the collection. Updating the index may involve adding new postings to the index, removing postings from the index, or a combination of these two kinds of operations.

Within the scope of this paper, we restrict ourselves to the case of *incremental* text collections, where new documents may be added to the collection, but existing documents are never deleted or modified. Index update strategies that deal with deletions and modifications can easily be combined with the methods proposed in this paper. Possible ways of extending the techniques described here to non-incremental text collections are discussed by Chiueh and Huang (1998) and Büttcher (2007).

Our work is based on the assumption that the index maintained by the search engine is substantially larger than the available amount of main memory and therefore, at least partially, needs to be stored on disk. Index maintenance strategies for on-disk inverted files are fundamentally different from those for in-memory indices. In both cases, it is generally desirable to store each posting list in a contiguous fashion, increasing spatial locality and cache effectiveness of the query processing routines. However, the impact of non-contiguous posting lists on the search engine's query processing performance is far greater for on-disk indices than if the search engine stores all its data in main memory. With an in-memory index, a discontiguity in a query term's posting list usually results in a CPU cache miss, carrying a penalty of around 100 CPU cycles (comparable to accessing a few dozen in-memory postings in a sequential fashion). With an on-disk index, each discontiguity at query time requires a disk seek — an operation that is roughly as expensive as reading 100,000 contiguously stored postings from disk.

As a consequence of this dichotomy, our results are applicable to desktop search engines and small enterprise search solutions with scarce memory resources, but not necessarily to large-scale search engines (e.g., Web search engines) that are composed of hundreds or thousand of machines and that might be equipped with enough main memory to hold the entire index in memory at all times.

Despite the fundamental difference between hard disks and RAM, however, there is a close connection between incremental update strategies for in-memory and for on-disk inverted files. Virtually every on-disk index update strategy operates by accumulating index data for incoming documents in RAM, building an in-memory inverted index; whenever the size of the in-memory index exceeds a certain, predefined threshold (or, alternatively, after some period of inactivity), the entire in-memory index is transferred to disk and combined with the existing on-disk index data (cf. Cutting and Pedersen, 1990; Lester et al., 2005; Lester et al., 2006). By following this approach, costly on-disk index updates are amortized over a larger number of postings and a larger number of inverted lists. Furthermore, because the in-memory index is immediately queriable by the search engine, it allows for real-time index updates.

Different index update policies usually only differ in the way in which they combine the in-memory index data with the existing on-disk index structures. The two traditional index update strategies are *re-merge update* and *in-place update*. The re-merge policy operates by sequentially reading the existing on-disk index and merging it with the in-memory index, thus creating a new on-disk index that includes the in-memory data and that supersedes the old index. In-place update strategies, in contrast, do not read the entire on-disk index when the indexing process runs out of memory. Instead, whenever a new on-disk inverted list is created, some amount of free space is pre-allocated at the end of the list. When transferring in-memory index data to disk, postings for a given term are appended at the end of the respective list. A term's list is relocated whenever there is insufficient space for such an append operation, with some amount of free space pre-allocated again at the end of the list's new on-disk location.

The necessity to store each on-disk posting list in a contiguous fashion, and to enforce this contiguity at all times, is the main limitation of traditional merge-based and in-place index maintenance algorithms. To overcome this limitation, Büttcher and Clarke (2005) and Lester et al. (2005) investigated *indexing time vs. query time trade-offs*, showing how the search engine's index maintenance overhead can be reduced by allowing a small amount of fragmentation in the on-disk posting lists. At query time, this results in a slightly increased query processing cost, due to the greater number of disk seeks.

In this paper, we are not concerned with such trade-offs. Instead, we focus on the case where all on-disk posting lists have to be stored in contiguous form, so as to maximize query processing performance. We show that a combination (*hybrid*) of the two competing index update paradigms for contiguous inverted lists (re-merge, in-place) leads to better performance than each policy type alone. We also show that the search engine's index maintenance overhead can be reduced even further by moving away from the idea that the *entire* in-memory index needs to be transferred from memory to disk when the indexing process runs out of memory. The resulting technique, called *partial flushing*, allows the search engine to only transfer *some* posting lists to disk, instead of all of them. As two different on-disk posting lists may have a different update cost, when measured in an amortized fashion (i.e., per incoming posting or incoming byte of compressed postings data), partial flushing leads to better overall performance than treating all lists equally.

In the next section, we give a brief overview of existing work in the area of inverted index maintenance. Section 3 contains a description of our basic hybrid index maintenance strategy, called HYBRID IMMEDIATE MERGE *with contiguous posting lists* (HIM_C). This is followed by a discussion of partial flushing (Section 4). In Section 5, we present a theoretical complexity analysis of HYBRID IMMEDIATE MERGE, based on the assumption that terms in the text collection being indexed roughly follow a Zipfian distribution (Zipf, 1949). Section 6 presents the key results we obtained in our experiments with hybrid and non-hybrid index maintenance. It includes an experimental validation of the performance characteristics of hybrid index maintenance, derived theoretically in Section 5. We conclude the paper with a discussion of our findings and some directions for future work (Section 7).

2 Background and Related Work

2.1 Inverted Indices

The two principal components of an inverted index are: (i) the dictionary and (ii) the posting lists. The dictionary provides a mapping from the set of index terms to the positions of their posting lists. Each term's posting list is a sequence of all its appearances in the collection, sorted by order of occurrence in the text.

The posting lists, in general, need to be stored on disk, because they are too large to be kept in memory. The dictionary, in contrast, can usually either be stored in memory or on disk. Both solutions have their respective advantages and disadvantages. Keeping the dictionary in memory allows for very fast lookup operations, but has the disadvantage that precious memory space is consumed that could otherwise be used to buffer postings for incoming documents or to cache search results for frequent queries. Moreover, the space requirements of a complete dictionary for a large collection, such as the EDU corpus used in our experiments, may actually exceed the amount of main memory available to the search engine. Conversely, storing the dictionary on disk minimizes its memory requirements, but increases the search engine's query processing overhead, by introducing an additional disk seek per query term.

As an alternative to these two extremes, the dictionary can be interleaved with the on-disk posting lists (each posting list is preceded by the respective dictionary entry), keeping only a small number of dictionary entries in memory and the vast majority on disk. Dictionary interleaving leads to a substantial reduction of the dictionary's memory requirements, albeit at the cost of a slightly higher disk overhead at query time, usually less than 1 ms per query term (Büttcher, 2007, ch. 4). The technique, however, can only be applied if all lists in the on-disk index are arranged in some pre-defined order (e.g., lexicographically).

Inverted indices come in a few slightly different flavors. Based on the exact type of information contained in the posting lists, we can distinguish between:

- **docid indices**, where each posting is a simple integer, representing the numerical document ID of a document containing the given term;
- **frequency indices**, in which each posting is of the form $(d, f_{t,d})$, where d is a docid, and $f_{t,d}$ is the number of occurrences of the term t in the document d (the *within-document term frequency*);
- **document-centric positional indices**, in which each posting is of the form $(d, f_{t,d}, \langle p_1, \ldots, p_{f_{t,d}} \rangle)$, where each p_i is the word position of an occurrence of the term t within the document d;

- **schema-independent indices**, where each posting is a simple integer, denoting the word distance of a term occurrence from the beginning of the text collection.

A schema-independent index, in contrast to the three other index types, has no explicit notion of a *document*. The text collection is treated as a "flat" sequence of tokens, without any inherent structure. The actual structure of the text collection is defined by the user (or the application) on a per-query basis, for example by combining `<doc>` and `</doc>` tokens into documents. An in-depth discussion of the schema-independent approach to text retrieval is given by Clarke et al. (1994, 1995).

For the sake of simplicity, all algorithms presented in this paper (as well as all our experiments) assume that the search engine maintains a schema-independent index. However, the methods we discuss apply to all four types of inverted indices. In particular, the performance results obtained with schema-independent indices may be considered representative of document-centric positional indices, as both index representations lead to an inverted file of roughly the same size.

2.2 Index Construction

A text collection can be thought of as a sequence of tuples of the form $(token, position)$. Initially, these tuples are arranged in *collection order*, sorted by their second component. The task of constructing an inverted index from a given collection can, in essence, be thought of as changing the ordering of the tuple sequence, bringing it from its original collection order into *index order*, that is, sorted by their first component (ties broken by second component).

High-performance index construction algorithms for static text collections usually operate in two stages. In the first stage, the collection is divided into n sub-collections $\mathcal{C}_1, \ldots, \mathcal{C}_n$, where the size of each part is defined by the amount of main memory available to the indexing process. For each sub-collection $\mathcal{C}_k$, a separate inverted file $\mathcal{I}_k$ is built in-memory and transferred to disk when the indexing process runs out of memory. The partitioning operation can be performed on-the-fly: sub-collection $\mathcal{C}_{k+1}$ starts as soon as sub-index $\mathcal{I}_k$ reaches the memory limit imposed upon the indexing process and has to be transferred to disk. In the second stage, the n sub-indices $\mathcal{I}_1, \ldots, \mathcal{I}_n$ are combined into the final index, through an n-way merge operation. The general approach outlined above is referred to as *merge-based index construction* (Heinz and Zobel, 2003).

Merge-based indexing algorithms may differ in the way in which they build the index partitions $\mathcal{I}_k$. Published work on this topic, however, suggests that hash-based implementations with *move-to-front* heuristic lead to the best overall performance (Zobel et al., 2001). In such implementations, an extensible in-memory posting list is maintained for each index term. A central, in-memory hash table is used to lookup the memory address of the corresponding list for each incoming token, and the new posting is appended at the end of the extensible list. The move-to-front heuristic pulls the hash table entry for a term T to the front of the respective collision chain whenever a new posting is added to T's posting list.

Hash-based indexing is very suitable for dynamic text retrieval systems, as all incoming postings can immediately be accessed by the query processing routines. For sort-based indexing methods (e.g., Moffat, 1992), this is not the case.

2.3 Index Compression

If the index's posting lists are kept on disk, then it is highly desirable to store them in compressed form, in order to save space and also to reduce the search engine's disk transfer overhead. Various methods for compressing the postings in an inverted list exist. Essentially all of them make use of the fact that the postings within a given list are sorted in increasing order. A posting list can then be transformed into an equivalent Δ representation, containing differences ("Δ values") between two consecutive postings instead of the postings themselves:

$$P = \langle\, p_1,\ p_2,\ p_3,\ \ldots,\ p_{|P|}\ \rangle, \quad \Delta(P) = \langle\, p_1,\ p_2 - p_1,\ p_3 - p_2,\ \ldots,\ p_{|P|} - p_{|P|-1}\ \rangle. \tag{1}$$

The Δ values tend to be rather small and can be encoded using standard integer coding techniques, such as the γ code (Elias, 1975), Golomb/Rice codes (Golomb, 1966), or the Huffman-based LLRUN algorithm (Fraenkel and Klein, 1985).

If overall performance (indexing performance, query processing performance, or index maintenance performance) is the optimization criterion, then encoding/decoding performance (measured in seconds per posting) is just as important as raw compression effectiveness (measured in bits per posting). In that case, simple byte-aligned or word-aligned encoding methods, such as *vByte* ("variable-byte"; Scholer et al., 2002) or *Simple-9* (Anh and Moffat, 2005), usually lead to better results than the more complicated, variable-bit encoding methods listed above.

Within the context of index maintenance, where a large portion of the overall complexity stems from the necessity to combine two or more list fragments for the same term into a single posting list, it is important to note that such a merge operation does not require any substantial decompression activity. The compression algorithm, however, needs to be modified slightly so that it emits compressed lists of the form:

$$\Delta'(P) = \langle\, p_1, p_{|P|}, p_2 - p_1, p_3 - p_2, \ldots, p_{|P|-1} - p_{|P|-2}\ \rangle. \tag{2}$$

instead of what is shown in (1). Combining two list fragments P and Q with

$$\Delta'(P) = \langle\, p_1,\ p_{|P|},\ p_2 - p_1,\ \ldots,\ p_{|P|-1} - p_{|P|-2}\ \rangle,\ \ \Delta'(Q) = \langle\, q_1,\ q_{|Q|},\ p_2 - q_1,\ \ldots,\ q_{|Q|-1} - q_{|Q|-2}\ \rangle, \tag{3}$$

where $q_1 > p_{|P|}$, then only requires re-encoding $p_{|P|}$ and q_1, resulting in:

$$\Delta'(P \circ Q) = \langle\, p_1,\ q_{|Q|},\ p_2 - p_1,\ \ldots,\ p_{|P|} - p_{|P|-1},\ q_1 - p_{|P|},\ \ldots,\ q_{|Q|-1} - q_{|Q|-2}\ \rangle. \tag{4}$$

By following this approach, the cost of an index merge operation is usually reduced by 15%–20% compared to a naïve implementation that decompresses all posting lists (Büttcher, 2007, ch. 4).

2.4 Index Maintenance

Most index maintenance strategies for incremental text collections can be classified as being either merge-based strategies or in-place strategies. Merge-based techniques are usually easier to implement than in-place methods, as they only require the search engine to be able to merge two or more on-disk index partitions — an ability that every merge-based indexing algorithm for static text collections already possesses by nature. The difference between merge-based index maintenance and the merge-based index construction method from Section 2.2 is that the index maintenance strategy does not wait until the very end of the index construction process (after all, what is the *end* of an incremental text collection?). Instead, the current in-memory index is merged directly with the existing on-disk index, as soon as the indexing process runs out of memory. The resulting on-disk inverted file replaces the old one, the in-memory index is discarded, and the search engine continues its indexing operations.

The index update policy outlined above is known as Remerge (cf. Lester et al., 2004). In this paper, we refer to it as Immediate Merge, to emphasize that each merge operation takes place immediately, whenever the search engine runs out of buffer space. Immediate Merge is easy to implement, guarantees the contiguity of all on-disk posting lists, and usually leads to acceptable performance results. It has, however, a fundamental limitation: because the engine needs to process the entire on-disk index whenever it runs out of memory, the total index maintenance disk complexity (measured by the number of postings transferred from/to disk), is:

$$\sum_{k=1}^{\lfloor N/M \rfloor} \left((k-1)\cdot M + k\cdot M\right) \quad \in \; \Theta\left(N \cdot \frac{N}{M}\right), \tag{5}$$

where N is the number of tokens in the collection, and M is the memory limit of the indexing process (number of postings that fit into main memory). If $N \gg M$, then Immediate Merge's extremely high disk activity may render the method unattractive for practical applications.

In-place update strategies do not share this shortcoming. An in-place policy, whenever it transfers an in-memory posting list to disk, allocates some amount of free space at the end of the newly created on-disk list. The next time, when the indexing process transfers postings for the same term to disk, the new postings may be appended at the end of the existing list, without the need to read the entire list from disk (as is the case with Immediate Merge). If there is not enough free space to host the new postings, then the entire list is relocated to a new on-disk position, again with some free space pre-allocated at the end of the list. Just like Immediate Merge, in-place update maximizes query performance, as all on-disk posting lists are stored in contiguous form. If the pre-allocation decisions have to be made without any knowledge of the future growth of the respective posting list, then proportional pre-allocation policies, allocating $k \cdot s$ bytes for a list of size s, usually lead to best performance results (Tomasic et al., 1994; Lester et al., 2006). If the search engine makes use of historical data on the growth of each inverted list to predict its future growth, slightly better results can be obtained (Shieh and Chung, 2005).

We refer to the basic version of in-place index maintenance with proportional pre-allocation as Inplace. Because of the geometric progression defined by the proportional pre-allocation policy, Inplace relocates a list of size s at most $\log_k(s)$ times. Similarly, the total number of bytes transferred from/to disk during such relocation operations is at most

$$2 \cdot \sum_{i=0}^{\infty} \frac{s}{k^i} \quad = \quad \frac{2s}{1 - 1/k} \tag{6}$$

(the factor 2 accounts for the fact that relocating a list of length s requires s bytes to be read and then s bytes to be written). For instance, with a pre-allocation factor $k = 2$, at most $4s$ bytes are transferred from/to disk.

Equation 6 implies an overall linear index maintenance disk complexity of the Inplace strategy, clearly better than the quadratic complexity exhibited by Immediate Merge. Unfortunately, this advantage is not reflected by the experimental results obtained for Inplace and Immediate Merge. Lester et al. (2006), for instance, report that their implementation of Inplace is generally outperformed by Immediate Merge, except when the search engine's in-memory buffers are very small and on-disk index updates have to be carried out frequently. Lester (2006) ascribes the low update performance of Inplace to the large number of disk seek operations necessary to perform the in-place list updates. As on-disk lists, due to repeated relocation operations, are not arranged in any predefined order, but essentially appear in random order in the index, every in-place list update necessitates a random-access disk operation. The cost of such an operation is in the order of milliseconds. For short lists, comprising only a few hundred postings, it may be several thousand times greater than the cost associated with writing the actual postings data to disk. The fact that on-disk posting lists essentially appear in random order has also implications on the dictionary data structure employed by the search engine. Dictionary interleaving schemes (Büttcher, 2007, ch. 4) are incompatible with in-place update methods, and the engine has to maintain an explicit dictionary for all terms in the index — most likely too large to be held completely in main memory.

In comparison, we can say that merge-based index maintenance seems appropriate when we are dealing with large numbers of short lists, where the overhead of a random-access disk operation is not worthwhile. Long lists, on the other hand, where the amount of new data is small compared to the existing, and unchanged, on-disk posting list, are likely to benefit from the random-access capabilities of in-place update.

2.5 Previous Approaches to Hybrid Index Maintenance

Index maintenance policies that combine the advantages of merge-based and in-place update strategies are not entirely new. The existing approaches, however, all have in common that they are either guided by heuristics or motivated by details of the implementation, as opposed to an analysis of the performance characteristics of the computer's storage medium.

Cutting and Pedersen (1990), for instance, discuss an index maintenance mechanism that is very similar to the Immediate Merge policy, but operates on a B-tree as the basic data structure. Posting lists that are too large to be stored in a single node of the B-tree are transferred to an auxiliary storage area, the *heap file*, and are updated in place. Similarly, Shoens et al. (1994) describe a dual-structure index organization in which short lists are kept in fixed-size buckets (with multiple lists sharing the same bucket), while long lists are kept in a separate index and are updated in place. Initially, every list is considered *short*. Whenever a bucket overflows during an index update operation, the longest list in that bucket is removed from the bucket, is declared *long*, and is updated in place thereafter. However, no constraint is specified regarding the length of the longest list in any given bucket. Thus, a list may be declared to be *long* when it is in fact rather short. Brown et al. (1994) discuss an index structure that distinguishes between short, medium, and long lists and in which lists, under certain conditions, may be stored in a non-contiguous fashion. Finally, Lester et al. (2006) examine the effects of storing short posting lists, smaller than 512 bytes, directly in the index's dictionary data structure, instead of in the postings file. Their results with this hybrid index organization are encouraging. However, as we shall see in Section 6.4, the threshold chosen by Lester et al. is far too small. For a typical consumer hard drive, with a seek latency around 10 ms and a sequential read/write bandwidth of 50 MB/sec, the optimal threshold value is approximately 1,000 times greater than 512 bytes.

3 Hybrid Index Maintenance

As we have discussed in the previous section, the main limitation of Inplace is its non-sequential disk access pattern, requiring a large number of rather expensive disk seek operations. The main limitation of Immediate Merge, on the other hand, is its quadratic update complexity (reading and writing the entire on-disk index every time the indexing process runs out of memory). Suppose we do not apply the same update policy to the entire index, but may choose between the two methods on a list-by-list basis. Consider an individual inverted list of size s_i and s_{i+1} bytes, after the i-th and the $(i+1)$-st update cycle, respectively:

- With Immediate Merge, the list-specific update cost during the $(i+1)$-st update cycle, measured by the amount of disk activity, is:
$$C_{remerge}(s_i, s_{i+1}) = \frac{s_i + s_{i+1}}{b}, \tag{7}$$
where b is the hard drive's read/write bandwidth for sequential transfer operations, measured in bytes per second. For typical hard drives, the value of b is usually not a constant, but depends on the disk track(s) being accessed. For simplicity, however, we assume that b is a (device-specific) constant.

- With Inplace, the list-specific update cost is:
$$C_{inplace}(s_i, s_{i+1}) = C_{random} + \frac{s_{i+1} - s_i}{b}, \tag{8}$$
where C_{random} is the hard drive's random access latency, measured in seconds per random access operation. As with b, the value of C_{random} depends on the exact disk access pattern of the process, as the time needed to perform a disk seek is a function of the distance travelled by the disk's read/write head. To simplify the cost model, however, it will be treated as a constant.

By comparing equations 7 and 8, we see that the Inplace policy results in better performance than Immediate Merge for lists that meet the criterion $s_i > (b \cdot C_{random})/2$. Conversely, for lists that are shorter than $(b \cdot C_{random})/2$, Immediate Merge leads to better performance. It follows that the total disk overhead associated with updating on-disk inverted lists is not optimized by exclusively applying either Inplace or Immediate Merge to the entire index, but by splitting the index into two sections. One section contains *short* lists (less than $(b \cdot C_{random})/2$ bytes) and is updated according to Immediate Merge. The other section contains *long* lists (more than $(b \cdot C_{random})/2$ bytes) and is updated according to the Inplace policy.

This idea leads to a new index maintenance policy, called Hybrid Immediate Merge, formalized in Algorithm 1. The algorithm takes a single parameter ϑ, the threshold used to decide when a list becomes long and should be moved to the in-place index section. The algorithm maintains a set $\mathcal{L}$ of long lists, which is initially empty. Whenever, during a re-merge operation, the indexing process encounters a short list whose new size exceeds the threshold ϑ, the list is marked as long, added to the set $\mathcal{L}$, and transferred to the in-place index section (cf. lines 14–17 in the algorithm).

Because each on-disk posting list is completely stored in either the merge-maintained or the in-place-updated index section, it is guaranteed that all on-disk inverted lists are stored in a contiguous fashion at all times, so as to maximize query performance. We therefore refer to this method as Hybrid Immediate Merge *with contiguous posting lists* (HIM_C) to distinguish it from versions of hybrid index maintenance that do not enforce the contiguity of the on-disk posting lists (cf. Büttcher et al., 2006).

3.1 Finding the Optimal Threshold Value

The long-list threshold ϑ, in Algorithm 1, is passed to the update policy as an explicit parameter value. In an actual implementation, this is not necessary. The index maintenance process may calculate the optimal value of the parameter ϑ automatically, by measuring the relative performance of sequential and random-access disk operations.

Combining equations 7 and 8, we know that the overall index update is minimized by choosing $\vartheta = (b \cdot C_{random})/2$, where b is the hard drive's average sequential read/write performance (measured in bytes

Algorithm 1 On-line index construction according to HYBRID IMMEDIATE MERGE with contiguous posting lists (HIM_C). Input parameter: long list threshold ϑ.

```
let $I_{merge}$ denote the primary on-disk index (merge-maintained, initially empty)
let $I_{inplace}$ denote the secondary on-disk index (updated in-place, initially empty)
$I_{mem} \leftarrow \emptyset$  // initialize in-memory index
$currentPosting \leftarrow 1$
$\mathcal{L} \leftarrow \emptyset$  // the set of long lists, initially empty
while there are more tokens to index do
  $T \leftarrow$ next token
  $I_{mem}.addPosting(T,\ currentPosting)$
  $currentPosting \leftarrow currentPosting + 1$
  if $I_{mem}$ contains more than $M - 1$ postings then
    // merge $I_{mem}$ and $I_{merge}$ by traversing their lists in lexicographical order
    $I'_{merge} \leftarrow \emptyset$  // create new on-disk index partition
    for each term $T \in (I_{mem} \cup I_{merge})$ do
      if $(T \in \mathcal{L}) \ \vee \ (I_{mem}.getPostings(T).byteSize + I_{merge}.getPostings(T).byteSize > \vartheta)$ then
        $I_{inplace}.addPostings(T,\ I_{merge}.getPostings(T))$
        $I_{inplace}.addPostings(T,\ I_{mem}.getPostings(T))$
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{T\}$
      else
        $I'_{merge}.addPostings(T,\ I_{merge}.getPostings(T))$
        $I'_{merge}.addPostings(T,\ I_{mem}.getPostings(T))$
      end if
    end for
    $I_{merge} \leftarrow I'_{merge}$  // replace old merge-maintained index partition with new one
    $I_{mem} \leftarrow \emptyset$  // re-initialize the in-memory index
  end if
end while
```

per second), and C_{random} is its average random access latency (measured in seconds per random access operation). However, we have not yet discussed how to compute the exact value of C_{random}. One might suspect that C_{random} is simply the hard drive's seek latency. But this is not the case. By unifying IMMEDIATE MERGE and INPLACE update into a hybrid update policy, we have increased the cost of each random-access in-place list update. This is because each such update now interrupts a sequential read/write operation affecting the merge-maintained index section.

Within the context of HYBRID IMMEDIATE MERGE, the total cost of an in-place index update consists of four components:

1. Seeking to the disk position within the in-place index section corresponding to the existing on-disk inverted list.

2. Reading some small amount of data from the end of the list.

3. Writing the data just read, plus the new data, back to disk.

4. Seeking back to the merge-maintained index section, to proceed with the ongoing re-merge operation.

The necessity of steps 1, 3, and 4 is obvious. Step 2 is necessary for two reasons. Firstly, on-disk postings are stored in compressed form, and compression is applied to larger chunks (*posting list segments*) instead of individual postings. In order to add new postings to a given list segment, the segment header needs to be read from disk so that segment size and compression parameters can be obtained. Secondly, hard disk drives do not allow data access at the byte level, but only in larger blocks, typically 512 or 4,096 bytes. Changing only parts of a disk block requires the operating system to first load the old version into memory, apply all changes, and then write the modified version back to disk.

The overall cost of an in-place index update is the sum of all four components and can be estimated as follows. Steps 1 and 4 each carry the cost of a single disk seek operation. The cost of step 2 depends on the hard drive's rotational latency: on average, the update process needs to wait half a disk rotation before it can read the desired data. The cost of step 3, finally, depends on the rotational latency (because the update process has to wait a full rotation, until the disk's read/write is located above the end of the on-disk posting list again) and the disk's sequential read/write performance. Hence, we have:

$$\begin{aligned} C_{inplace}(s_i, s_{i+1}) &= 2 \cdot C_{seek} + 1.5 \cdot C_{rot} + \frac{s_{i+1} - s_i}{b}, && (9) \\ C_{random} &= 2 \cdot C_{seek} + 1.5 \cdot C_{rot}, && (10) \\ \vartheta_{opt} &= \frac{b \cdot (2 \cdot C_{seek} + 1.5 \cdot C_{rot})}{2}. && (11) \end{aligned}$$

Assuming, for example, an average disk seek latency of $C_{seek} = 8$ ms, a rotational velocity of 7,200 rpm (i.e., $C_{rot} = 8.33$ ms), and an average disk read/write throughput of $b = 40$ million bytes per second, we obtain

$$C_{random} = 2 \cdot 8 \text{ ms} + 1.5 \cdot 8.33 \text{ ms} = 28.5 \text{ ms} \quad (12)$$

and thus

$$\vartheta_{opt} = \frac{28.5 \text{ ms} \cdot 40 \cdot 10^6 \text{ bytes/s}}{2} = 570,000 \text{ bytes.} \quad (13)$$

As we shall see in Section 6, this theoretical estimate is in line with our experimental results. It is also in line with earlier findings by Lester et al. (2006) indicating that the optimal threshold value is greater than 512 bytes (in their experiments, Lester et al. tried various threshold values and observed the general trend that a greater threshold ϑ leads to better update performance; they stopped at $\vartheta = 512$ bytes).

4 Partial Flushing

The basic approach to incremental index updates, as introduced in Section 1, assumes that the entire in-memory index has to be transferred to disk when the indexing process runs out of memory. With the non-hybrid IMMEDIATE MERGE policy, this is certainly the case. With INPLACE or HYBRID IMMEDIATE MERGE, however, it is no longer true. Because each inverted list in an in-place-maintained index can be updated individually, without the need to update the entire index, an indexing process employing the INPLACE policy may release memory resources by transferring an arbitrary subset of the in-memory posting lists to disk. If it maintains its on-disk index structures according to HYBRID IMMEDIATE MERGE, the indexing process is not quite as flexible as in the case of INPLACE, but it can still choose from an arbitrary set of *long* lists (updated in place) when transferring in-memory index data to disk in an attempt to make room for incoming postings. In particular, it may release memory resources without transferring any of the *small* lists to disk, thus avoiding a costly re-merge operation. The general idea of not transferring the entire in-memory index to disk when the indexing process runs out of memory, but only a small portion of all in-memory inverted lists, is referred to as *partial flushing* (Büttcher and Clarke, 2006).

In essence, partial flushing reduces the re-merge overhead, at the cost of a greater number of in-place updates. In order for the method to be effective, care has to be taken that the additional in-place list updates triggered by a partial flush do not outweigh the savings achieved by delaying the next re-merge operation. For example, suppose the in-memory postings for a certain term account for 1% of the indexing process's main memory consumption. By transferring these postings to disk, at the cost of a single random-access in-place list update ($= C_{random}$), some amount of main memory can be released, and the time between two subsequent re-merge operations (transferring the entire in-memory index to disk) can be increased by 1%. If the cost of a full on-disk index update (including the re-merge operation carried out in the merge-maintained index section) is more than 100 times the cost of a single in-place list update, then transferring the term's in-memory postings to disk is worthwhile. Otherwise, it is not.

The relative cost of a single random-access in-place list update (C_{random}) and a complete on-disk index update ($C_{complete}$) implicitly defines a partial flushing threshold ϑ_{PF} that identifies the set of in-memory posting lists that should participate in a partial flush, and also the set of lists that should not. By instructing the indexing process to keep track of the cost of the previous complete index update, it is possible to make the system automatically determine the optimal partial flushing threshold $\vartheta_{\text{PF,opt}}$:

$$\vartheta_{\text{PF,opt}} = \frac{M \cdot C_{random}}{C_{complete}} \tag{14}$$

where M is the total size of the in-memory index (in bytes), C_{random} is the average in-place update overhead of a single list, and $C_{complete}$ is the total update cost measured during the last re-merge operation (lines 11–24 in Algorithm 1).

Because of the Zipfian term distribution (Zipf, 1949) exhibited by most natural-language text collections, the vast majority of postings in the index belongs to a rather small number of terms. It is therefore not unusual that the size of the in-memory index can be reduced by over 50%, even if only a few thousand lists participate in the partial flush. The effectiveness of partial flushing, however, diminishes with every subsequent partial flush operation. This is because, by transferring postings for *long* lists to disk, a greater portion of the in-memory index is occupied by *short* lists — lists than cannot participate in a partial flush. For this reason, after several consecutive partial flushings, the method is no longer beneficial, and a complete update of all on-disk inverted lists has to be performed.

At first glance, it might seem that partial flushing is effective as long as at least one list is transferred to disk. However, this is not so. Each partial flush carries a cost by itself, without taking into account the cost of transferring one or more lists to disk. Among other things, the index maintenance process first needs to identify the set of lists that should participate in the partial flushing operation and then needs to reclaim the memory space occupied by those lists so that it can be used for incoming postings. Within the memory management regime employed by our search engine, which was originally designed with high-performance index construction in mind, not taking the special requirements of partial flushing into account, reclaiming even a single byte of memory space always requires the traversal of the entire in-memory index (cf. Büttcher, 2007, ch. 4). Consequently, it may happen that the net effect of a partial flush is negative even though the condition stated in Equation 14 is met by all lists participating in the operation.

The problem can be solved in the following way: in addition to the partial flushing threshold ϑ_{PF}, the system also employs an effectiveness criterion (cut-off threshold) ω that is used to decide when to turn off partial flushing and to carry out a complete on-disk index update instead. As soon as the system reaches the point at which partial flushing reduces the relative main memory requirements of the indexing process

Algorithm 2 HIM_C with partial flushing. Building a schema-independent index for a potentially infinite text collection. Input parameters: $\vartheta \geq 1$, the long-list threshold; C_{random}, the cost of a single random-access in-place list update; $\omega \in (0, 1)$, the effectiveness criterion, used to determine when partial flushing is no longer useful.

```
1: let $I_{merge}$ denote the primary on-disk index (merge-maintained, initially empty)
2: let $I_{inplace}$ denote the secondary on-disk index (updated in-place, initially empty)
3: $I_{mem} \leftarrow \emptyset$ // initialize in-memory index
4: $currentPosting \leftarrow 1$
5: $\mathcal{L} \leftarrow \emptyset$ // the set of long lists is empty initially
6: $\vartheta_{\text{PF}} \leftarrow \infty$ // initially, disable partial flushing
7: $\omega' \leftarrow 0$ // assume the last partial flush had no effect at all (force full index update in first iteration)
8: while there are more tokens to index do
9:   // accumulate postings in memory
10:   while there are more tokens to index and $I_{mem}$ contains less than $M$ postings do
11:     $T \leftarrow$ next token
12:     $I_{mem}.addPosting(T,\ currentPosting)$
13:     $currentPosting \leftarrow currentPosting + 1$
14:   end while
15:   // perform on-disk index update (either full or partial, depending on $\omega'$)
16:   $totalByteSize \leftarrow I_{mem}.getMemoryConsumption()$ // obtain size of $I_{mem}$, in bytes
17:   if $(\omega' \geq \omega)$ then
18:     // flush partial in-memory index to disk
19:     for each $T \in \mathcal{L}$ do
20:       if $I_{mem}.getPostings(T).byteSize > \vartheta_{\text{PF}}$ then
21:         // transfer in-memory postings for $T$ to in-place index
22:         $I_{inplace}.addPostings(T,\ I_{mem}.getPostings(T))$
23:         $I_{mem}.deleteTerm(T)$
24:       end if
25:     end for
26:     $\omega' \leftarrow 1 - \frac{I_{mem}.getMemoryConsumption()}{totalByteSize}$ // keep track of effectiveness of partial flushing
27:   else
28:     // merge $I_{mem}$ and $I_{merge}$ by traversing their lists in lexicographical order
29:     $startTime \leftarrow getTime()$
30:     $I'_{merge} \leftarrow \emptyset$ // create new on-disk index partition
31:     for each term $T \in (I_{mem} \cup I_{merge})$ do
32:       if $(T \in \mathcal{L}) \vee (I_{mem}.getPostings(T).byteSize + I_{merge}.getPostings(T).byteSize > \vartheta)$ then
33:         $I_{inplace}.addPostings(T,\ I_{merge}.getPostings(T))$
34:         $I_{inplace}.addPostings(T,\ I_{mem}.getPostings(T))$
35:         $\mathcal{L} \leftarrow \mathcal{L} \cup \{T\}$
36:       else
37:         $I'_{merge}.addPostings(T,\ I_{merge}.getPostings(T))$
38:         $I'_{merge}.addPostings(T,\ I_{mem}.getPostings(T))$
39:       end if
40:     end for
41:     $I_{merge} \leftarrow I'_{merge}$ // replace old merge-maintained inverted file with new one
42:     $I_{mem} \leftarrow \emptyset$ // re-initialize the in-memory index
43:     $endTime \leftarrow getTime()$
44:     // merge operation finished; update variables related to partial flushing
45:     $\vartheta_{\text{PF}} \leftarrow \frac{totalByteSize \cdot C_{random}}{endTime - startTime}$ // adjust partial flushing threshold
46:     $\omega' \leftarrow 1$ // reset effectiveness level to 100%
47:   end if
48: end while
```

by less than ω, partial flushing is temporarily disabled, and the next update of the search engine's on-disk index structures will be a complete update, affecting all on-disk posting lists.

The modified indexing procedure, employing hybrid index maintenance with partial flushing, is given by Algorithm 2. It automatically adjusts the partial-flushing threshold ϑ_{PF}, by keeping track of the duration of the last complete index update. The algorithm takes three input parameters: the partial flushing threshold ϑ, the cost associated with an in-place list update C_{random}, and the partial flushing cut-off criterion ω. Note, however, that in an actual implementation of the algorithm it is possible to automatically estimate the optimal values of all three parameters, by continually monitoring the performance of all ongoing index maintenance operations. The effectiveness cut-off parameter ω, for instance, can be obtained and adjusted dynamically, by measuring the duration of a partial flushing operation and by comparing it to the amount of main memory freed by the operation, in a way very similar to finding the optimal value of ϑ_{PF}:

$$\omega_{opt} = \frac{C_{complete}}{C_{partial}}, \tag{15}$$

where $C_{complete}$, again, is the total cost of the last full update of the search engine's on-disk index structures, and $C_{partial}$ is the overall cost of the last partial flushing.

5 Complexity Analysis

We now present a complexity analysis of the $\mathrm{HIM_C}$ index maintenance policy proposed in Section 3. Our analysis is based on the assumption that the terms in the text collection for which an index is maintained follow a Zipfian distribution. This assumption is not uncommon and has, for example, also been made by Cutting and Pedersen (1990), for very similar purposes.

The analysis consists of two parts. In the first part (Section 5.1), we derive the total number of long lists (lists containing more than a certain number ϑ of postings) and the total number of postings contained in short lists (lists containing fewer than ϑ postings). In the second part (Section 5.2), we use the results from the first part to estimate the number of disk operations (sequential and random-access) carried out by $\mathrm{HIM_C}$. Because disk activity is the main bottleneck of all index maintenance strategies, we obtain a rather accurate approximation of the total index maintenance cost of the system.

In our analysis, we assume that disk operations always carry the same cost, regardless of what part of the disk they affect (in reality, storing data on the outer tracks of the hard disk is more efficient than storing them on the inner tracks). We also assume that all postings consume the same amount of space, regardless of which term they belong to, which is slightly incorrect, as postings for a more frequent term can be compressed better than postings for a less frequent term. However, the difference only amounts to a logarithmic factor and may be ignored.

5.1 Generalized Zipfian Distributions: Long Lists and Short Lists

According to Zipf's law (Zipf, 1949), terms in a collection of text in a natural language roughly follow a distribution of the following form:

$$f(T_i) \;=\; \left[\ \frac{c}{i^{\alpha}}\ \right], \tag{16}$$

where T_i is the i-th most frequent term, $f(T_i)$ is the collection frequency of T_i, i.e., the number of times the term appears in the collection ("$[\cdots]$" denotes rounding to the nearest integer); c and α are collection-specific constants. In his original work, Zipf postulated $\alpha = 1$.

Equation 16 is equivalent to the assumption that the probability of an occurrence of the term T_i, according to a unigram language model (that is, treating each token as an independent event), is

$$\Pr[T_i] \;=\; \frac{c}{N \cdot i^\alpha}, \tag{17}$$

where N is the total number of tokens in the text collection. In order for this probability distribution to be well-defined, the following equation needs to hold:

$$\sum_{i=1}^{\infty} \Pr[T_i] \;=\; \sum_{i=1}^{\infty} \frac{c}{N \cdot i^\alpha} \;=\; 1. \tag{18}$$

Because N and c are constants, this implies the convergence of the sum

$$\sum_{i=1}^{\infty} \frac{1}{i^\alpha}$$

and thus requires that α be greater than 1. If $\alpha > 1$, then the sum's value is given by the Riemann zeta function $\zeta(\alpha)$ and can be approximated in the following way:

$$\sum_{i=1}^{\infty} \frac{1}{i^\alpha} \;\approx\; \gamma + \int_1^\infty \frac{1}{x^\alpha}\,dx \;=\; \gamma + \frac{1}{\alpha - 1}, \tag{19}$$

where

$$\gamma \;=\; -\lim_{n\to\infty} \left(\ln(n) - \sum_{i=1}^{n} \frac{1}{i} \right) \;\approx\; 0.577216 \tag{20}$$

is the Euler-Mascheroni constant. The quality of the approximation in (19) depends on the value of the collection-specific parameter α. For realistic values ($1 < \alpha < 1.4$), the approximation error is less than 1%. The integral representation of the Riemann zeta function is far more accessible to analysis than its representation as a sum. We therefore exclusively use the former when modeling the term distribution in a given text collection.

As a notational simplification, we refer to the term $\gamma + \frac{1}{\alpha-1}$ in Equation 19 as γ_α^*. From this definition, it follows that the value of the constant c in equations 16 ff. is:

$$c = \frac{N}{\gamma_\alpha^*} \quad \text{and thus} \quad f(T_i) = \frac{N}{\gamma_\alpha^* \cdot i^\alpha}, \tag{21}$$

where N again is the number of tokens in the text collection.

Long Lists and Short Lists

We can now calculate $L(N, \vartheta)$, the number of terms that occur more than ϑ times in a text collection comprising N tokens. Their corresponding lists are referred to as *long lists*. From Equation 21, we know:

$$f(T_i) > \vartheta \;\Leftrightarrow\; i^\alpha < \frac{N}{\gamma_\alpha^* \cdot \vartheta} \;\Leftrightarrow\; i < \left(\frac{N}{\gamma_\alpha^* \cdot \vartheta} \right)^{1/\alpha}.$$

Hence, we have

$$L(N, \vartheta) \;=\; \left\lfloor \left(\frac{N}{\gamma_\alpha^* \cdot \vartheta} \right)^{1/\alpha} \right\rfloor \;\in\; \Theta\left(\frac{N^{1/\alpha}}{\vartheta^{1/\alpha}} \right). \tag{22}$$

The total number of postings found in these lists, denoted as $\hat{L}(N, \vartheta)$, then is

$$\begin{aligned}
\sum_{i=1}^{\lfloor L(N,\vartheta) \rfloor} f(T_i) &= \frac{N}{\gamma_\alpha^*} \cdot \sum_{i=1}^{\lfloor L(N,\vartheta) \rfloor} \frac{1}{i^\alpha} \\
&\approx \frac{N}{\gamma_\alpha^*} \cdot \left(\gamma + \int_1^{L(N,\vartheta)} x^{-\alpha} \, dx \right) \\
&= \frac{N}{\gamma_\alpha^*} \cdot \left(\gamma + \left(\frac{(L(N,\vartheta))^{1-\alpha}}{1-\alpha} - \frac{1}{1-\alpha} \right) \right) \\
&= \frac{N}{\gamma_\alpha^*} \cdot \left(\gamma + \frac{1}{\alpha - 1} \right) - \frac{N}{\gamma_\alpha^*} \cdot \left(\frac{(L(N,\vartheta))^{1-\alpha}}{\alpha - 1} \right) \\
&= N - \frac{N}{\gamma_\alpha^* \cdot (\alpha - 1)} \cdot \left(\frac{N}{\gamma_\alpha^* \cdot \vartheta} \right)^{\frac{1-\alpha}{\alpha}} \\
&= N - N^{1/\alpha} \cdot \frac{\vartheta^{(1-1/\alpha)}}{(\alpha-1)(\gamma_\alpha^*)^{1/\alpha}}.
\end{aligned} \tag{23}$$

It immediately follows that the total number of postings found in short lists (lists containing no more than ϑ entries), denoted as $\hat{S}(N, \vartheta)$, is

$$\begin{aligned}
\hat{S}(N, \vartheta) &= N - \hat{L}(N, \vartheta) \\
&= \frac{N^{1/\alpha} \cdot \vartheta^{(1-1/\alpha)}}{(\alpha-1)(\gamma_\alpha^*)^{1/\alpha}} \;\in\; \Theta\left(\vartheta^{(1-1/\alpha)} \cdot N^{1/\alpha} \right).
\end{aligned} \tag{24}$$

5.2 Complexity of Hybrid Immediate Merge

The total index maintenance cost of incrementally building and updating an inverted index for a text collection of size N consists of three components:

1. Reading and tokenizing the input data; building in-memory inverted files.
2. Merging in-memory posting lists with posting lists found in the merge-maintained section of the on-disk index (*short lists*).
3. Updating posting lists found in the in-place-updated section of the on-disk index (*long lists*).

We determine the cost of each component individually.

Reading, Tokenizing, Indexing

Reading and tokenization can be done in linear time. By employing a hash-based inversion algorithm (cf. Section 2.2) to incrementally build an in-memory inverted file, the indexing cost can be kept linear, too. In total, therefore, the cost of the first component is $\Theta(N)$.

Merge Operations

Let M be the amount of main memory available to the indexing process, measured by the number of tokens that can be indexed before the process runs out of memory. When building and maintaining an index for a collection of N tokens, the system runs out of memory $\lfloor N/M \rfloor$ times. Whenever this happens, a re-merge operation is carried out, reading the old merge-maintained index section from disk and replacing it by a new on-disk inverted file. The number of postings written to disk during the k-th re-merge operation is:

$$\hat{S}(k \cdot M, \vartheta) = \frac{(k \cdot M)^{1/\alpha} \cdot \vartheta^{(1-1/\alpha)}}{(\alpha - 1)(\gamma_\alpha^*)^{1/\alpha}}. \tag{25}$$

Thus, the total number of postings written to disk, during all re-merge operations, is:

$$\begin{aligned} & \sum_{k=1}^{\lfloor N/M \rfloor} \hat{S}(k \cdot M, \vartheta) && (26)\\ = \; & \frac{\vartheta^{(1-1/\alpha)}}{(\alpha - 1)(\gamma_\alpha^*)^{1/\alpha}} \cdot \sum_{k=1}^{\lfloor N/M \rfloor} (k \cdot M)^{1/\alpha} && (27)\\ \in \; & \Theta\left(\vartheta^{(1-1/\alpha)} \cdot \frac{N^{1+1/\alpha}}{M}\right). && (28) \end{aligned}$$

The number of postings read from disk is bounded from above by the number of postings written to disk (nothing is read twice). The number of random-access disk operations carried out is negligible, as the disk access pattern of a merge operation is essentially sequential. Therefore, the overall cost of the merge operations, measured by the search engine's disk activity, is in fact:

$$\Theta\left(\vartheta^{(1-1/\alpha)} \cdot \frac{N^{1+1/\alpha}}{M}\right). \tag{29}$$

In-Place List Updates

When updating an on-disk inverted file according to the Inplace policy, the total number of postings transferred from/to disk is linear in the size of the collection, as discussed in Section 2. Therefore, the only interesting aspect of the third component, the in-place list updates, is the number of random-access disk operations carried out by the index maintenance process.

Let us assume that, whenever the search engine runs out of memory, every list in the in-place index section needs to be updated and that each in-place on-disk index update, on average, requires a constant number of random access operations. From a theoretical point of view, the first assumption is an oversimplification, as the number of lists in the in-place index is unbounded, while the size of the memory buffer is a constant. However, for a realistic long-list threshold of $\vartheta \approx 10^6$, corresponding to roughly 500,000 postings, it is in fact the case that every list in the in-place section needs to be updated during every index upate cycle, at least for the first few thousand cycles. Thus, for the purpose of modeling the actual update performance in a realistic environment, the assumption is valid.

The number of lists in the in-place section of the on-disk index, after the indexing process runs out of memory for the k-th time, is:

$$L(k \cdot M, \vartheta) = \left(\frac{k \cdot M}{\gamma_\alpha^* \cdot \vartheta}\right)^{1/\alpha} \tag{30}$$

Table 1: Performance characteristics of the two computer systems used in the experiments. The two systems mainly differ in their hardware configuration, with the Opteron storing its on-disk index data on a 2-way RAID-0 partition, achieving a sequential disk throughput twice as high as the Athlon64's single-disk setup.

Computer system	**CPU freq.**	**RAM**	**Disk performance**		
			Seek latency	Rotational velocity	Throughput
Athlon64	2203 MHz	2,048 MB	8 ms	7,200 rpm	40 MB/sec
Opteron	2814 MHz	2,048 MB	8 ms	7,200 rpm	75 MB/sec

(cf. Equation 22). Therefore, the total number of in-place list updates is:

$$\sum_{k=1}^{\lfloor N/M \rfloor} \left(\frac{k \cdot M}{\gamma_\alpha^* \cdot \vartheta} \right)^{1/\alpha} \quad \in \quad \Theta\left(\vartheta^{-1/\alpha} \cdot \frac{N^{1+1/\alpha}}{M} \right). \tag{31}$$

Total Cost

Combining the results for all three components, and treating the long-list threshold ϑ and the time complexity of a random-access disk operation as constants, we obtain the following overall index maintenance complexity:

$$a \cdot N \; + \; b \cdot \frac{N^{1+1/\alpha}}{M} \; + \; c \cdot \frac{N^{1+1/\alpha}}{M} \tag{32}$$

$$= \quad \hat{a} \cdot N \; + \; \hat{b} \cdot \frac{N^{1+1/\alpha}}{M}, \tag{33}$$

for some constants a, b, c, $\hat{a}$, $\hat{b}$ (with $\hat{b} = b + c$). Compared to the non-hybrid Immediate Merge baseline policy with its $\Theta\left(N \cdot \frac{N}{M}\right)$ update complexity, this is a substantial improvement. When run on a text collection with Zipf parameter $\alpha = 1.33$, for instance, HIM$_\mathrm{C}$ is expected to exhibit a total complexity of $\Theta\left(\frac{N^{1.75}}{M}\right)$.

6 Experiments

We evaluated Hybrid Immediate Merge *with contiguous posting lists* (HIM$_\mathrm{C}$), with and without partial flushing, on two different computer systems and using two different text collections. The results are compared to the non-hybrid Immediate Merge baseline policy.

Our experiments are limited to the index maintenance performance of the individual methods and do not cover their query processing performance. However, because all methods covered in this paper store their on-disk posting lists in a contiguous manner, we strongly suspect that the difference between the individual methods would have been extremely small and would have been unlikely to provide any additional insight into the problem of updating an inverted index.

6.1 Data Sets and Hardware Configuration

The two computer systems used in our experiments are: a 2.2 GHz AMD Athlon64 with 2 GB of RAM and a 2.8 GHz AMD Opteron, also with 2 GB of RAM. An overview of these two systems is given in Table 1. The main difference between the two systems, besides their different CPUs, is the fact that the Opteron system stores its on-disk index data on a two-disk RAID-0 partition (software RAID) instead of the single

Table 2: Summary of the two text collections used in the experiments. Despite both collections being the result of a Web crawl, the statistical characteristics of GOV2 and EDU are quite different ($\alpha_{\text{GOV2}} = 1.333$, $\alpha_{\text{EDU}} = 1.263$). This difference is witnessed by the greater number of distinct terms in EDU than in GOV2.

Collection	Type	Text size	Index size	Documents	Tokens	Unique terms
GOV2	Web crawl	426 GB	62 GB	25.2 million	43.5 billion	49.5 million
EDU	Web crawl	644 GB	108 GB	44.5 million	68.4 billion	266.8 million

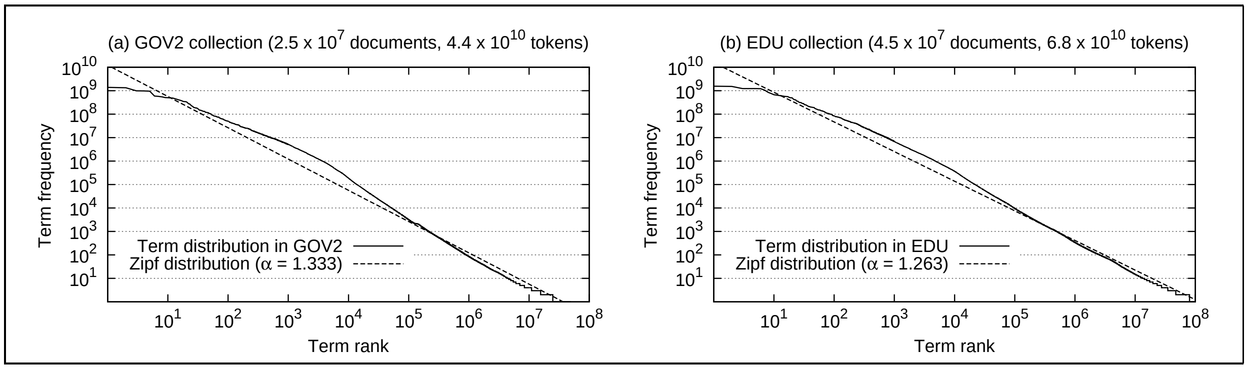

Figure 1: Rank/frequency curves for GOV2 and EDU, along with the corresponding best-fit Zipf curves. Zipf parameters for best fit: $\alpha_{\text{GOV2}} = 1.333$, $\alpha_{\text{EDU}} = 1.263$.

hard drive. The Opteron's RAID achieves a sequential read/write performance that is roughly twice as high as the Athlon64's single hard drive. This difference allows us to examine the effect that the hard drive's read/write throughput has on the optimal value of the hybridization threshold ϑ.

The two text collections used in the experiments are: GOV2 (Clarke et al., 2004), the result of a Web crawl of the `.gov` domain carried out in early 2004, and EDU, the result of a Web crawl of the `.edu` domain in early 2006. GOV2 is available from the University of Glasgow[1]. EDU is not publicly available. A summary of the two collections is given in Table 2.

Using two different text collections allows us to experimentally validate the theoretically suggested impact of the Zipf parameter α on the overall index maintenance performance of HIM_C. According to Equation 33, a greater value of the Zipf parameter α results in a better performance of HIM_C, compared to Immediate Merge. We computed approximate best-fit α values for both text collections by plotting each collection's term rank/frequency curve and choosing α in such a way that the area between the actual rank/frequency curve and the curve suggested by modeling the collection according to the Zipfian distribution was minimized. The minimization step was performed after a log/log transformation (i.e., plotting term ranks and term frequencies in logarithmic scale). For GOV2, we obtained a best-fit parameter value of $\alpha_{\text{GOV}} = 1.333$; for EDU, we obtained $\alpha_{\text{EDU}} = 1.263$. The rank/frequency curves for both collections are shown in Figure 1.

6.2 Implementation and Index Update Sequence

All experiments were conducted using the freely available Wumpus[2] search engine, running as a 64-bit user process under Linux[3] 2.6.18. All on-disk index data were stored as ordinary files in the file system, realized

[1]http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm
[2]http://www.wumpus-search.org/
[3]http://www.kernel.org/

Table 3: Index update command sequences used to simulate an incremental text collection.

```
@addfile GOV2/gov2-corpus/GX234/13.txt
@addfile GOV2/gov2-corpus/GX059/89.txt
@addfile GOV2/gov2-corpus/GX131/65.txt
@addfile GOV2/gov2-corpus/GX024/85.txt
@addfile GOV2/gov2-corpus/GX069/63.txt
...
```

```
@addfile EDU/edu-corpus/EDU069/04.txt.gz
@addfile EDU/edu-corpus/EDU045/31.txt.gz
@addfile EDU/edu-corpus/EDU035/75.txt.gz
@addfile EDU/edu-corpus/EDU025/50.txt.gz
@addfile EDU/edu-corpus/EDU033/71.txt.gz
...
```

by Linux's implementation of `ext3`, using one file per index section / inverted file. To keep the amount of fragmentation introduced by the file system layer low, we made sure that no more than 70% of the machine's total disk space was utilized at any given time.

The inverted files used in our experiments were schema-independent (cf. Section 2.1), with every posting being a simple integer value, denoting the distance of a term occurrence from the beginning of the text collection. All postings, both in the on-disk and in the in-memory index, were stored in compressed form, using the byte-aligned vByte method (Scholer et al., 2002). Disk access operations were carried out on large chunks of data (we used a buffer size of 4 MB for all sequential read/write operations), cached by the operating system. However, at the end of each on-disk index update (partial flush or complete index update cycle), all data currently in the operating system's file system cache, but not yet applied to the on-disk index, were forced to be written to disk by executing an `fsync` system call.

Our implementation of the Inplace policy employed a simple proportional pre-allocation strategy with a pre-allocation factor $k = 2$, allocating disk space for *long* posting lists in multiples of 512 KB (making it impossible to use in a stand-alone Inplace implementation, but good enough in the context of HIM_C). Whenever the index maintenance process ran out of space in the in-place index section, the size of the file was increased by performing an `ftruncate` system call, making the file big enough to find a new position for the posting list being relocated.

In our experiments, we simulated a dynamic search environment in which the search engine needs to maintain an index for a continually growing text collection. For this purpose, GOV2 was split up into 27,204 different files, with each file on average containing 1.6 million tokens in 926 documents. EDU was divided into 6,912 files, with each file containing 9.9 million tokens in 6,438 documents on average. The search engine was instructed to build an index for the given collection, processing all files in the collection (in random order) and enforcing the contiguity of each on-disk posting list at all times. The beginning of the actual index update sequences used in our experiments is shown in Table 3.

Most of our experiments were repeated twice, and we did not see a single case in which the overall time difference between two iterations of the same experiment was greater than 1%. Thus, the performance figures given in the remainder of this section may be considered reliable.

6.3 Hybrid vs. Non-Hybrid Immediate Merge

In an initial set of experiments, we evaluated the basic HIM_C policy, for two different threshold values $\vartheta = 125{,}000$ and $\vartheta = 2{,}000{,}000$, using GOV2 as the underlying text collection and allowing the indexing process to hold up to 512 MB of postings data in its in-memory buffers.

Figure 2(a) shows the number of posting lists in the in-place index section, as the index grows. It can

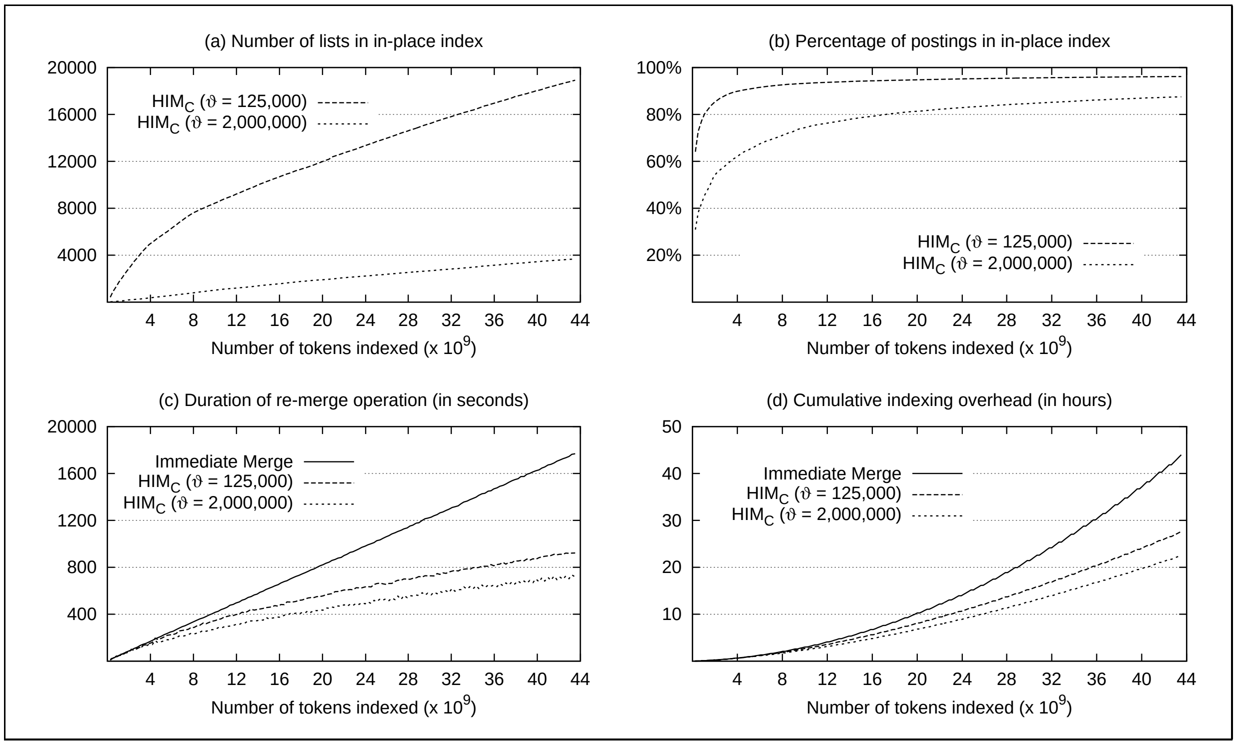

Figure 2: Comparative performance evaluation of HIM_C and Immediate Merge on the GOV2 collection. Hardware: AMD Opteron. Memory limit: 512 MB.

be seen that the number of lists in the in-place section is very small, even for the rather low threshold value of $\vartheta = 125{,}000$. After the complete index has been built, less than 20,000 lists have been transferred to the in-place index ($\vartheta = 2{,}000{,}000$: less than 4,000 lists). The fact that the number of lists in the in-place index is so small (i.e., that there are so few terms with long lists) means that dictionary maintenance is far less challenging with HIM_C than in the case of the non-hybrid Inplace method, where millions of dictionary entries need to be maintained and updated together with the posting lists. For example, with HIM_C, it is entirely feasible to keep the dictionary entries for all in-place-updated posting lists in main memory — something that is not always possible for Inplace (cf. Section 2.4).

In contrast to the small number of lists in the in-place section, the total number of postings in those lists is rather large, as shown in Figure 2(b). For $\vartheta = 125{,}000$, after indexing only 1 billion tokens ($\approx$ 2.5% of GOV2), more than 80% of all postings are found in the in-place index section. For $\vartheta = 2{,}000{,}000$, the same percentage is reached after indexing 17 billion tokens ($\approx$ 40% of GOV2) . As a consequence, the merge-maintained index section remains relatively small in both cases, resulting in a lower re-merge cost than in the case of the non-hybrid Immediate Merge update policy. This is shown in Figure 2(c). After indexing 50% of GOV2, for example, the cost of a re-merge operation is reduced by between 32% and 47% (between 467 and 600 seconds with HIM_C, instead of 885 seconds with Immediate Merge). This, of course, directly translates into a reduced cumulative indexing overhead, between 22 hours and 28 hours for HIM_C, compared to 44 hours for Immediate Merge (cf. Figure 2(d)).

The relative gains achieved by HIM_C, compared to the non-hybrid Immediate Merge strategy, increase as the index grows (cf. Figure 3). For instance, HIM_C with $\vartheta = 2 \cdot 10^6$ is about 50% faster than Immediate

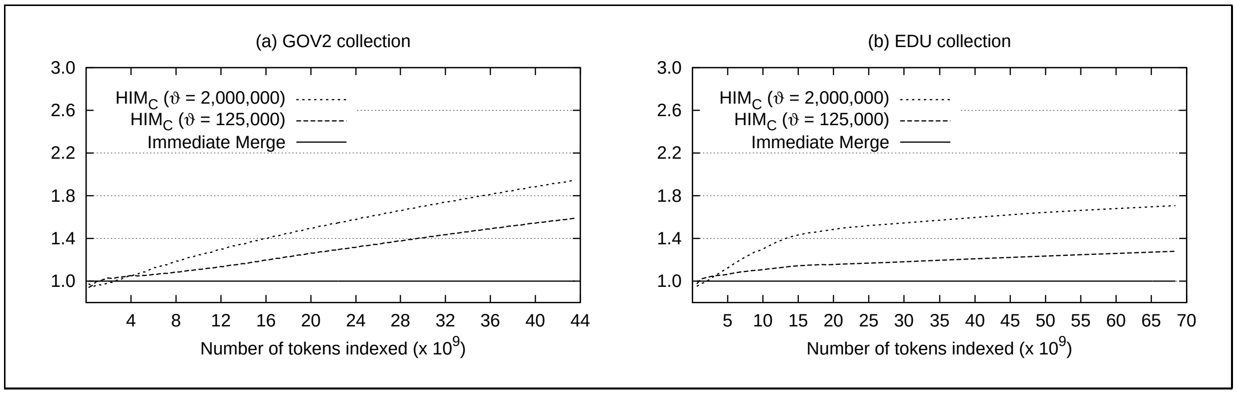

Figure 3: Relative performance of HIM_C, compared to Immediate Merge. For both collections, HIM_C's performance advantage increases as the index grows. Hardware: AMD Opteron. Memory limit: 512 MB.

Merge when indexing the first 20 billion tokens from GOV2. When indexing the first 40 billion tokens, it is about 90% faster. This confirms our initial expectation that the relative performance of HIM_C grows with the collection size (cf. equations 5 and 33). The curves in Figure 3 seem to be slightly convex, but do not appear to be bounded.

By comparing Figure 3(a) and Figure 3(b), we see that the performance gains are higher for GOV2 than for the EDU collection. After indexing 40 billion tokens, the relative performance gain on the GOV2 collection is 90% (for ϑ=2,000,000). For EDU, it is only 55%. This confirms our expectation that the index maintenance performance of HIM_C is higher for collections with greater Zipf-α value than for collections with smaller α ($\alpha_{\text{GOV2}} = 1.333$, $\alpha_{\text{EDU}} = 1.263$).

6.4 Optimal Threshold Values

It is worth pointing out that the two threshold values examined in our initial set of experiments ($\vartheta = 125{,}000$ and $\vartheta = 2{,}000{,}000$) represent a rather large spectrum of possible parameter values. Yet, the savings achieved relative to the Immediate Merge baseline are substantial in both cases: for GOV2, the total index maintenance overhead is reduced by between 37% and 49%; for EDU, between 22% and 41%. This implies that HIM_C is not overly sensitive to the choice of the exact threshold value ϑ.

Nonetheless, optimal update performance is, of course, achieved by carefully tuning the value of ϑ and making it reflect the relative performance of sequential and random-access disk operations carried out by re-merge update and in-place update, respectively. Theoretical considerations in Section 3.1 suggested that the index update cost is minimized by choosing

$$\vartheta \quad = \quad \frac{b \cdot (2 \cdot C_{seek} + 1.5 \cdot C_{rot})}{2}. \tag{34}$$

By replacing b, C_{seek}, and C_{rot} with the actual performance values of our systems, we obtain

$$\vartheta_{opt,Athlon64} \quad = \quad (40 \cdot 10^6 \text{ bytes/s} \cdot 28.5 \text{ ms})/2 \quad = \quad 570{,}000 \text{ bytes} \tag{35}$$

Table 4: Sequential disk access overhead, random access overhead, and total index maintenance cost of HIM_C for the GOV2 collection on both computer systems. Memory limit: 512 MB.

Long list threshold ϑ ($\times 10^3$)	125	250	500	1,000	2,000	4,000	∞
Bytes transferred from/to disk ($\times 10^{12}$)	2.07	2.19	2.43	2.83	3.45	4.21	11.09
In-place list updates performed ($\times 10^6$)	1.88	1.27	0.86	0.54	0.31	0.17	0.00
Total running time in hours (Opteron)	27.65	22.30	20.30	19.75	22.58	23.83	43.98
Total running time in hours (Athlon64)	43.10	33.79	30.99	31.83	33.85	37.89	75.29

in the case of the Athlon64 system with its single-disk setup, and

$$\vartheta_{opt,Opteron} \quad = \quad (75 \cdot 10^6 \text{ bytes/s} \cdot 28.5 \text{ ms})/2 \quad = \quad 1{,}068{,}750 \text{ bytes} \tag{36}$$

for the Opteron with its two-disk RAID-0 storage system.

To validate these predictions, we had our search engine again build an index for GOV2 in an incremental fashion, using both computer systems (Athlon64 and Opteron), and employing a wide variety of threshold values between $\vartheta = 125{,}000$ and $\vartheta = 4{,}000{,}000$. The results of these experiments, shown in Table 4, confirm our theoretical model. The system's update performance is in fact maximized by choosing $\vartheta_{Athlon64} \approx 500{,}000$ bytes and $\vartheta_{Opteron} \approx 1{,}000{,}000$ bytes.

It is interesting to see that, although the optimal ϑ value differs by about a factor 2 between the two systems, the relative speedup achieved with the optimal threshold value is roughly the same on both systems. On the Opteron, the total indexing overhead is reduced by 56% (from 43.98 hours to 19.75 hours). On the Athlon64, it is reduced by 59% (from 75.29 hours to 30.99 hours). This indicates that hybrid index maintenance can even be beneficial for indices that are stored on high-throughput disk arrays, such as multi-way RAID setups.

6.5 Partial Flushing

In our next series of experiments, we evaluated the performance of the partial flushing technique from Section 4. As before, we had the search engine build an incremental index for both collections, GOV2 and EDU, employing HIM_C with partial flushing as the index maintenance policy.

Figure 4 shows the impact that partial flushing has on the indexing process's main memory consumption and on the frequency with which the search engine needs to carry out re-merge operations on the on-disk index data. The line-plots correspond to the search engine's memory consumption. The small circles correspond to partial flushing events.

Because of the Zipfian term distribution exhibited by the text collections (the majority of all postings is found in a small number of lists), the amount of main memory freed by flushing all long lists to disk is quite remarkable. Transferring the in-memory postings for the 1,200 most frequent terms in GOV2 to disk, for instance, reduces the indexing process's memory consumption by 53% (from 512 MB to 239 MB). Thus, as a result of applying the partial flushing technique, the amount of data indexed between two consecutive re-merge operations is increased noticeably. When building an index for GOV2, the engine runs out of memory for the first time after 280 million tokens. The next out-of-memory event is already delayed by two partial flushing operations and takes place after 280+360=640 million tokens. This trend continues, and after indexing 8 billion tokens, the distance between re-merge operations is roughly 650 million tokens, with

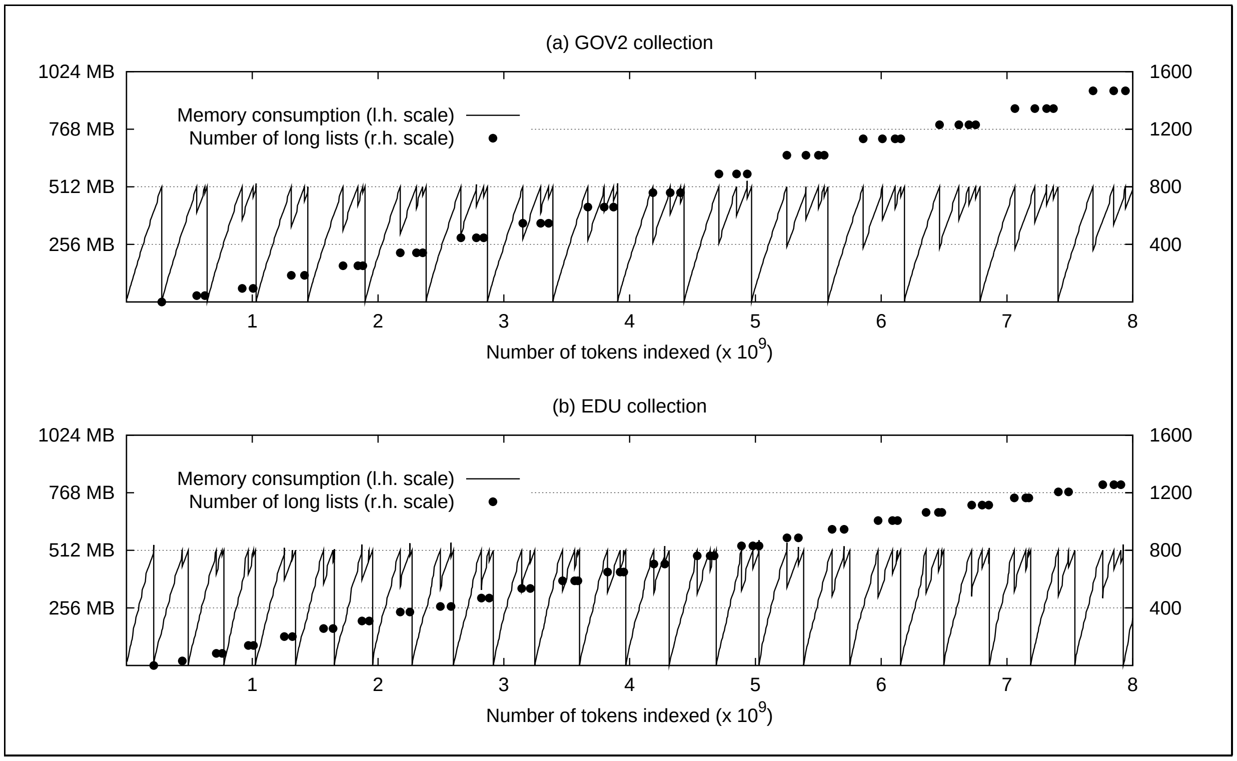

Figure 4: The impact of partial flushing on the search engine's memory consumption and on the frequency of re-merge operation. Each drop to 0 MB corresponds to a re-merge event. Each circle corresponds to a partial flush. Hardware: AMD Opteron. Memory limit: 512 MB. $\mathrm{HIM_C}$ parameter configuration: $\vartheta = 10^6, \vartheta_{\mathrm{PF}} = 1$ (i.e., *all* long lists participate in partial flushing).

four applications of the partial flushing sub-routines between each consecutive pair of re-merge operations.

Again, we notice a difference between GOV2 and EDU. When building an index for EDU, the relative number of postings found in *short* lists is greater for EDU than for GOV2 (caused by EDU's smaller α value; cf. Equation 24). In the context of partial flushing, this results in a lower effectiveness of the method. For example, when flushing the postings for the 1,200 most frequent terms to disk, the system's main memory consumption is only decreased by 40% for EDU (from 512 MB to 309 MB), as opposed to the 53% seen with GOV2.

As discussed in Section 4, partial flushing does not necessarily achieve its optimal performance by flushing postings for *all* long lists to disk. Instead, a list should only be flushed when the expected benefit exceeds the cost of performing an in-place update operation for that list (cf. Equation 14). We compared the slightly more sophisticated variant of partial flushing, taking into account both the cost of an in-place update and the expected savings obtained by performing the update, to the version in which all long lists are flushed to disk, to the basic $\mathrm{HIM_C}$ policy without partial flushing, and to IMMEDIATE MERGE. The results of our experiments, for both collections, are shown in Figure 5.

Plot 5(a) depicts the cumulative index maintenance overhead of all four update policies on the GOV2 collection. While partial flushing with $\vartheta = 1$ (i.e., flushing all long lists) does in fact have a slight edge over $\mathrm{HIM_C}$ without partial flushing, setting the threshold ϑ_{PF} according to a proper cost analysis of partial flushing results in even greater savings. While $\mathrm{HIM_C}$+PF with $\vartheta_{\mathrm{PF}} = 1$ indexes GOV2 in 16.4 hours (17%

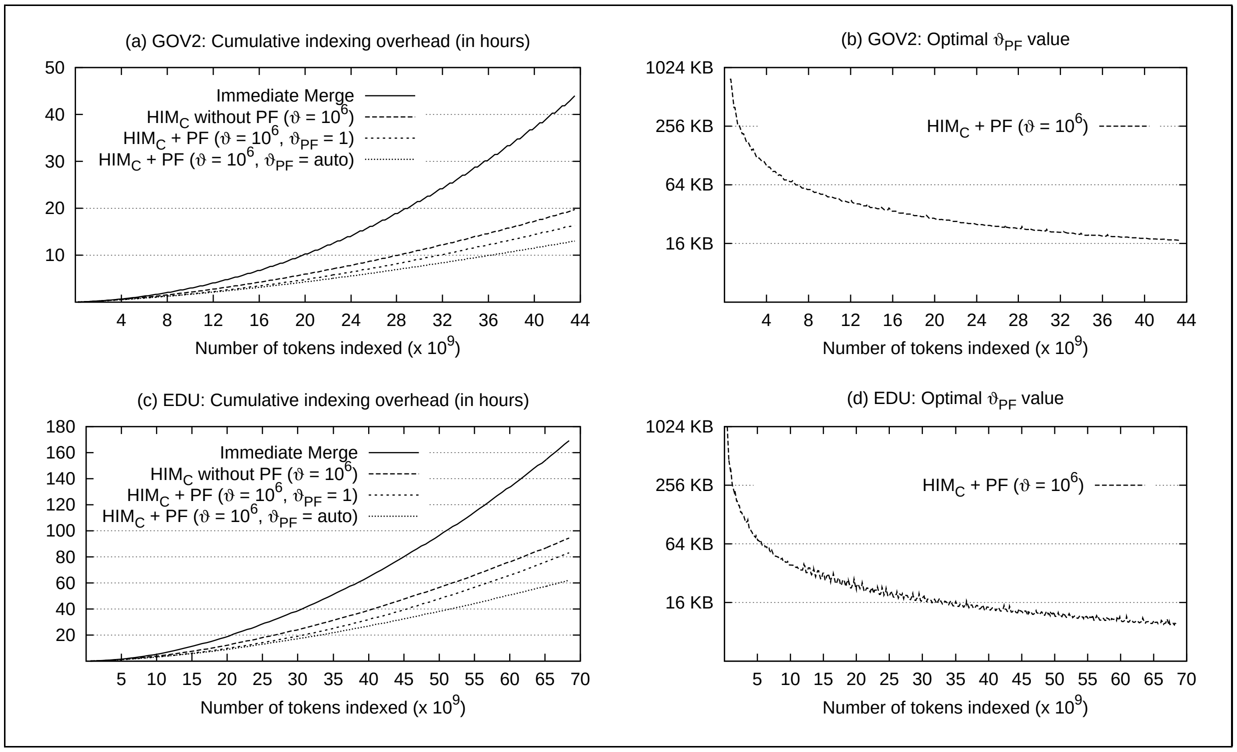

Figure 5: Index maintenance performance of HIM_C with partial flushing, compared to HIM_C without partial flushing and the Immediate Merge baseline. Top: GOV2 collection. Bottom: EDU collection. Hardware: AMD Opteron. Memory limit: 512 MB.

less than without partial flushing), $\vartheta_{\text{PF}} = \textit{auto}$ finishes the same job in just 13 hours (34% less than without partial flushing). Compared to the Immediate Merge baseline policy, the total indexing overhead is reduced by 70%.

Plot 5(b) shows how the optimal partial flushing threshold ϑ_{PF} develops over time. As the index grows, re-merge operations are getting more and more costly (cf. Figure 2(c)). Consequently, after a while, it becomes economical to flush even rather short list fragments with a relatively high amortized in-place update cost (amortized over the amount of data being written to disk).

Just like the basic HIM_C without partial flushing, HIM_C+PF realizes a speedup by taking advantage of the relative performance of sequential and random-access disk operations. Compared to the variant without partial flushing, HIM_C+PF ($\vartheta = \textit{auto}$) increases the number of random disk access operations, by performing 631,556 instead of 539,921 in-place list updates in total. However, the associated cost is by far outweighed by the reduced amount of data transferred from/to disk: from $2.83 \cdot 10^{12}$ bytes down to $1.28 \cdot 10^{12}$ bytes.

Plots 5(c) and 5(d) show the same results as plots (a) and (b), but for the EDU collection instead of GOV2. For EDU, the savings achieved are quite similar to what we have seen before; partial flushing (with $\vartheta = \textit{auto}$) reduces the total indexing time from 94.5 hours to 62.2 hours (-34%), for a total reduction of 63% compared to Immediate Merge.

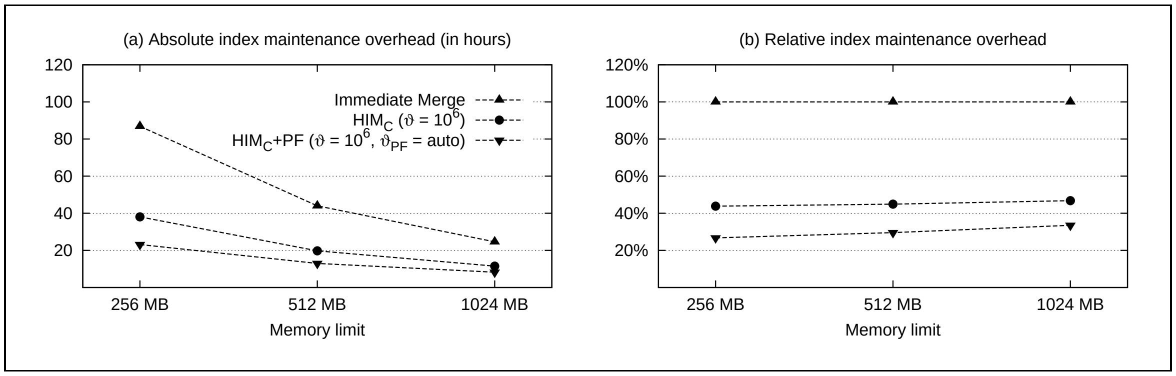

Figure 6: Impact of the indexing process's memory limit on the overall performance of the index maintenance algorithms. Left: Absolute index maintenance overhead. Right: Relative overhead, compared to Immediate Merge. Text collection: GOV2. Hardware: Opteron.

6.6 Impact of Memory Buffer Size

In a final series of experiments, we examined the impact that the available amount of main memory has on the update performance of the various index maintenance policies. Equation 33 suggests that the relative performance of HIM_C, compared to the Immediate Merge baseline, is independent of M, the amount of main memory that may be used to buffer incoming postings.

Figure 6 shows the total index maintenance overhead when building an incremental index for GOV2. Plot (a) depicts the absolute overhead, while plot (b) shows the relative overhead, compared to Immediate Merge. The results are roughly in line with the prediction from Section 5. When decreasing the size of the memory buffers from 1024 MB to 256 MB, HIM_C's relative update overhead remains almost constant (46.8% vs. 43.8%). For HIM_C+PF, we see a slight drop, from 33.5% to 26.7%. This drop, however, does not contradict our complexity analysis, as the analysis did not make any predictions for partial flushing.

6.7 Validation of Theoretical Complexity Analysis

In the previous parts of this section, we have experimentally examined the performance of several variants of HIM_C. We have shown that their actual performance is generally in line with what is predicted by the theoretical complexity analysis in Section 5. For instance, HIM_C leads to better results for GOV2 than for EDU, due to the greater Zipf-α value in GOV2; the relative performance of the method increases as the index grows.

However, the complexity analysis makes very specific predictions regarding the index maintenance complexity of our method. Under the assumption that the size of the memory buffers, M, is a constant, we can express the total index update time complexity for a collection comprised of N tokens as:

$$C_\alpha(N) = a \cdot N + b_1 \cdot N^{c_1} + b_2 \cdot N^{c_2}, \tag{37}$$

where a corresponds to the raw indexing overhead (reading input files, tokenizing input, etc.), b_1 and c_1 correspond to random access disk operations (in-place list updates), and b_2 and c_2 correspond to the sequential disk transfer overhead (re-merge operations). c_1 and c_2 depend on the statistical properties of the text collection for which an index is maintained, while a, b_1, and b_2 depend on both the collection and

Table 5: Index maintenance complexity functions for various update policies and parameter settings, obtained by computing a least-squares fit to the actual index maintenance overhead observed in the experiments. All functions are of the form: $C(n) = a \cdot n + b \cdot n^c$, where n is the number of tokens in the index. Memory limit for all experiments: $M = 1024$ MB.

Update policy	**Collection**	$\boldsymbol{\vartheta}$	$\boldsymbol{\vartheta}_{\text{PF}}$	**In-place list updates**	**Bytes transferred**
Imm. Merge	GOV2	n/a	n/a	n/a	$15.0 \cdot n + 3.22 \cdot 10^{-9} \cdot n^{1.992}$
HIM$_\text{C}$	GOV2	$1.0 \cdot 10^6$	n/a	$9.31 \cdot 10^{-15} \cdot n^{1.829}$	$15.0 \cdot n + 1.97 \cdot 10^{-6} \cdot n^{1.666}$
HIM$_\text{C}$+PF	GOV2	$1.0 \cdot 10^6$	1	$3.96 \cdot 10^{-15} \cdot n^{1.895}$	$15.0 \cdot n + 6.14 \cdot 10^{-6} \cdot n^{1.576}$
HIM$_\text{C}$+PF	GOV2	$1.0 \cdot 10^6$	auto	$1.36 \cdot 10^{-12} \cdot n^{1.632}$	$15.0 \cdot n + 8.20 \cdot 10^{-6} \cdot n^{1.564}$
Imm. Merge	EDU	n/a	n/a	n/a	$14.8 \cdot n + 6.65 \cdot 10^{-9} \cdot n^{1.977}$
HIM$_\text{C}$	EDU	$1.0 \cdot 10^6$	n/a	$1.21 \cdot 10^{-14} \cdot n^{1.823}$	$14.8 \cdot n + 4.45 \cdot 10^{-7} \cdot n^{1.761}$
HIM$_\text{C}$+PF	EDU	$1.0 \cdot 10^6$	1	$6.64 \cdot 10^{-15} \cdot n^{1.879}$	$14.8 \cdot n + 1.43 \cdot 10^{-6} \cdot n^{1.686}$
HIM$_\text{C}$+PF	EDU	$1.0 \cdot 10^6$	auto	$2.48 \cdot 10^{-13} \cdot n^{1.710}$	$14.8 \cdot n + 2.01 \cdot 10^{-6} \cdot n^{1.673}$

on the performance of the computer system. Based on the assumption that the text collection follows a Zipfian distribution with Zipf parameter α, the analysis in Section 5 predicted $c_1 = c_2 = 1 + 1/\alpha$. For GOV2 ($\alpha_{\text{GOV2}} = 1.333$), this resulted in an overall index maintenance complexity function of the form: $C_{1.333}(N) = a \cdot N + b \cdot N^{1.750}$. For EDU ($\alpha_{\text{EDU}} = 1.263$), it resulted in: $C_{1.263}(N) = a \cdot N + b \cdot N^{1.792}$.

These specific predictions have not been verified yet. In order to do so, we had our system again build an index for GOV2 and EDU in an incremental fashion, employing an in-memory buffer of size 1024 MB. We sampled the the total number of bytes transferred from/to disk and the total number of in-place list updates carried out, with one sample taken after the completion of each complete on-disk index update (i.e., after each re-merge event). This gave us a set of 27 to 149 data points (the exact number depends on which collection is indexed and which update policy is used). We assumed that the linear component in Equation 37 was the same for all update policies and approximated it by measuring the total amount of disk activity caused by a static, merge-based index construction method (we measured 15.0 bytes per token for GOV2 and 14.8 bytes per token for EDU). We then estimated the values of the remaining parameters in Equation 37 by computing a non-linear least-squares fit for both sets of data points, in-place updates performed and bytes transferred from/to disk. The outcome of this computation is shown in Table 5.

For the Immediate Merge strategy, the least-squares fit returned exponents $c_2 = 1.992$ (GOV2) and $c_2 = 1.977$ (EDU), respectively, fairly close to the expected value of $c_2 = 2$. The deviation from the theoretically predicted exponent is due to the fact that the engine's disk activity stems not only from postings being transferred from/to disk, but also from the dictionary entries (i.e., the index terms themselves) that are part of the index, too. Since the number of distinct terms does not grow linearly with the size of the collection, we see a slightly sub-quadratic overall complexity. The effect is stronger for EDU than for GOV2 because of the greater number of terms in EDU.

For the performance of the basic version of HIM$_\text{C}$, without partial flushing, our theoretical complexity analysis had predicted exponents $c_1 = c_2 = 1.792$ on the EDU collection, and in fact the least-squares fit ($c_1 = 1.823$, $c_2 = 1.761$) is very close to this prediction. On the GOV2 collection, however, the outcome of the least-squares computation ($c_1 = 1.829$, $c_2 = 1.666$) is quite different from the prediction $c_1 = c_2 = 1.750$. To understand the reason for this discrepancy, it is helpful to have another look at Figure 1. The figure shows the term distribution in GOV2 and EDU, as well as the best-fit Zipfian distribution for both collections. By comparing plot 1(a) to plot 1(b), we see that EDU is much better represented by its Zipfian distribution than GOV2. In both cases, the Zipfian approximation underestimates the collection frequency of all terms in the range $10^1 \leq \text{rank} \leq 10^5$. However, the error is greater for GOV2 than for EDU. For example, the Zipfian distribution for GOV2 (with $\alpha = 1.333$) underestimates the collection frequency of the term at rank 10^3 by a factor 5. Unfortunately, in our experiments, it is exactly this term range ($10^1 \leq \text{rank} \leq 10^5$) that is affected the most by the hybridization (cf. Figure 2(a)). As a result, the analysis from Section 5 underestimates the number of in-place list updates and overestimates the number of bytes transferred from/to disk.

Apart from the basic version of HIM_C, the regression analysis also returns specific complexity levels for the variants with partial flushing, which is convenient because we did not obtain any theoretical results for partial flushing. For a flushing threshold ϑ_{PF}=1 (i.e., all long lists are flushed to disk), the least-squares fit reveals that partial flushing affects the search engine's index maintenance performance by more than just a constant factor. Because the distance between two re-merge operations increases as the index gets larger (cf. Figure 4), the number of in-place updates per indexed token keeps growing, while the number of bytes transferred keeps getting smaller. This effect can be seen for both GOV2 and EDU (GOV2: c_1 jumps from 1.829 to 1.895; EDU: from 1.823 to 1.879). Hence, although we have not witnessed this phenomenon in our experiments, the regression analysis predicts that HIM_C+PF with ϑ_{PF}=1, in the long run, will be outperformed by HIM_C without partial flushing, as the cost of the in-place updates, asymptotically, is the dominant component of the method's overall complexity.

For partial flushing with ϑ_{PF}=*auto* (i.e., the version where a list only participates in a partial flush if the savings stemming from its participation are expected to outweigh the cost of the additional in-place update), the least-squares fit indicates a lower overall complexity for both aspects of the method's performance — in-place updates performed and bytes transferred from/to disk. The asymptotical disk transfer activity is roughly on par with ϑ_{PF}=1. The number of in-place list updates, however, is reduced greatly and is actually lower than for HIM_C without partial flushing. The reason for this is that ϑ_{PF}=*auto* not only reduces the number of re-merge operations, but also the number of in-place updates carried out for lists that are large enough to be considered *long*, but that, during a single index update cycle, do not accumulate enough postings for an in-place update to be economical. The method, therefore, is able to reduce both components of the search engine's total index maintenance disk overhead.

As a final note, it is worth pointing out that the overall performance of HIM_C+PF with ϑ_{PF}=*auto*, at least for the GOV2 collection ($c_1 = 1.632$, $c_2 = 1.564$), is actually quite close to that of geometric partitioning (Lester et al., 2005) with $p = 2$ (i.e., two on-disk index partitions). The latter exhibits a slightly lower asymptotic disk complexity of $\Theta(n^{1.414})$. HIM_C+PF, however, has the advantage that it stores all its on-disk posting lists in a contiguous fashion, leading to better query performance than geometric partitioning.

7 Discussion and Conclusions

We have presented and analyzed Hybrid Immediate Merge *with contiguous posting lists* (HIM_C), an index maintenance strategy to be used in dynamic text retrieval systems for incremental text collections. The method is a combination of the well-known Remerge and Inplace update policies for on-disk inverted files. Posting lists are divided into short lists and long lists. Short lists are updated according to the Remerge policy, while long lists are updated in place. In contrast to previous approaches to hybrid index maintenance, our method is not based on heuristics, but takes into account the actual performance characteristics of the storage system (random access performance vs. sequential read/write performance) when deciding whether a list should be updated in place or not.

Our experiments with two text collections of non-trivial size showed that HIM_C can reduce the search engine's index update overhead by up to 56%, compared to REMERGE. A refined version of HIM_C, called *partial flushing*, was able to achieve a reduction of 73%.

A theoretical performance analysis of HIM_C, based on the assumption that the terms in the text collection being indexed roughly follow a Zipfian distribution, was able to predict the general performance of the method, and was also able to predict the relative performance of HIM_C on different text collections with different term distributions. It did, however, fail to predict HIM_C's exact index maintenance complexity, measured by the number of bytes transferred to disk and the number of in-place list updates performed. We found that the reason for the discrepancy between theoretical performance and measured performance was the disagreement between Zipfian distribution and actual term distribution. For both collections used in our experiments, the Zipfian distribution consistently underestimated the frequency of terms in the range $10^1 \leq \text{rank} \leq 10^5$ (the Zipfian distribution was off by up to a factor 5 for one of the collections).

Unfortunately, this discrepancy is impossible to avoid, as a Zipfian distribution (after a log/log transformation) is always represented by a straight line in the rank/frequency plot. The text collections used in our experiments (and in fact all other collections that we looked at), however, exhibit a curved rank/frequency plot. Term distribution models that better reflect the actual term distribution in a given collection are known (e.g., Lavrenko, 2001), but require more complicated parameter estimation procedures than the simple Zipf distribution with its single parameter α. Furthermore, we do not know yet whether those models can be used to arrive at an asymptotical performance prediction of the type derived in Section 5.

Despite this slightly unsatisfactory result, it is clear that HIM_C constitutes a substantial improvement over the basic REMERGE policy. This was not only confirmed by our experimental performance figures, but also by a regression analysis which we conducted and which showed that HIM_C with partial flushing reduces the overall disk complexity of the search engine's index maintenance routines from REMERGE's $\Theta(n^2)$ to between $\Theta(n^{1.6})$ and $\Theta(n^{1.7})$, depending on the term distribution in the text collection. Unlike other index update strategies (e.g., geometric partitioning; Lester et al., 2005), HIM_C does not achieve this speedup by sacrificing its query performance. HIM_C and REMERGE both store their on-disk posting lists in a contiguous fashion and therefore exhibit the same query performance. Thus, the method does not represent a trade-off, but is indeed superior to the REMERGE baseline.

In our evaluation of HIM_C, we strictly limited ourselves to the case of incremental text collections, where new documents are added to the collection, but existing documents are never removed or modified. Of course, this is not a very realistic assumption. Fortunately, it is not very difficult to add support for document deletions to a maintenance policy designed for incremental updates, for example by defining a garbage collection policy that periodically sweeps the entire index whenever the relative number of garbage postings in the index exceeds a certain, predefined threshold (Büttcher, 2007, ch. 6). In fact, such a garbage collection policy has more flexibility in combination with HIM_C than with the non-hybrid REMERGE strategy, as it may be applied to individual lists instead of the entire index, perhaps even in a *piggy-backing* fashion, during query processing (Chiueh and Huang, 1998). At this point, we do not have any experimental evidence indicating the exact effect of document deletions and of HIM_C's greater garbage collection flexibility. We surmise, however, that HIM_C's performance advantage over REMERGE is slightly lower in the presence of deletions than for the strictly incremental case.

Acknowledgements

This work was in part supported by the University of Waterloo through the David R. Cheriton scholarship program. The authors would like to thank the reviewers for their thorough reviews and their valuable suggestions.

References

Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, January 2005.

Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 192–202, Santiago de Chile, Chile, September 1994.

Stefan Büttcher. *Multi-User File System Search*. PhD thesis, University of Waterloo, Canada, August 2007.

Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. Technical report, University of Waterloo, Canada, October 2005.

Stefan Büttcher and Charles L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In *Proceedings of the 28th European Conference on Information Retrieval*, pages 229–240, London, UK, April 2006.

Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 356–363, Seattle, USA, August 2006.

Tzi-cker Chiueh and Lan Huang. Efficient real-time index updates in text retrieval systems. Technical report, SUNY at Stony Brook, NY, USA, August 1998.

Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Schema-independent retrieval from heterogeneous structured text. Technical report, University of Waterloo, Canada, November 1994.

Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

Charles L. A. Clarke, Nick Craswell, and Ian Soboroff. Overview of the TREC 2004 Terabyte track. In *Proceedings of the 13th Text REtrieval Conference*, Gaithersburg, USA, November 2004.

Douglass R. Cutting and Jan O. Pedersen. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, Brussels, Belgium, September 1990.

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

Aviezri S. Fraenkel and Shmuel T. Klein. Novel compression of sparse bit-strings — preliminary report. *Combinatorial Algorithms on Words, NATO ASI Series*, 12:169–183, 1985.

Solomon W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12:399–401, July 1966.

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

Victor Lavrenko. A mathematical model of vocabulary growth. CIIR technical report IR-221. Technical report, University of Massachusetts, Amherst, MA, USA, 2001.

Nicholas Lester. *Efficient Index Maintenance for Text Databases.* PhD thesis, RMIT University, Melbourne, Australia, 2006.

Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Conference on Australasian Computer Science*, pages 15–23, Dunedin, New Zealand, 2004.

Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM Conference on Information and Knowledge Management*, pages 776–783, Bremen, Germany, November 2005.

Nicholas Lester, Justin Zobel, and Hugh E. Williams. Efficient online index maintenance for text retrieval systems. *Information Processing & Management*, 42(4):916–933, July 2006.

Alistair Moffat. Economical inversion of large text files. *Computing Systems*, 5(2):125–139, 1992.

Alistair Moffat and Timothy C. Bell. In-situ generation of compressed inverted files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.

Alistair Moffat and Justin Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Okapi at TREC-7. In *Proceedings of the Seventh Text REtrieval Conference*, Gaithersburg, USA, November 1998.

Falk Scholer, Hugh E. Williams, J. Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, Tampere, Finland, August 2002.

Wann-Yun Shieh and Chung-Ping Chung. A statistics-based approach to incrementally update inverted files. *Information Processing and Management*, 41(2):275–288, 2005.

Kurt A. Shoens, Anthony Tomasic, and Héctor García-Molina. Synthetic workload performance analysis of incremental updates. In *Proceedings of the 17th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 329–338, Dublin, Ireland, January 1994.

Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 289–300, Minneapolis, USA, 1994.

George K. Zipf. *Human Behavior and the Principle of Least-Effort.* Addison-Wesley, Cambridge, USA, 1949.

Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.

Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.