Fast Construction of the HYB Index

HANNAH BAST and MARJAN CELIKIK, Albert Ludwigs University

As shown in a series of recent works, the HYB index is an alternative to the inverted index (INV) that enables very fast prefix searches, which in turn is the basis for fast processing of many other types of advanced queries, including autocompletion, faceted search, error-tolerant search, database-style select and join, and semantic search. In this work we show that HYB can be constructed at least as fast as INV, and often up to twice as fast. This is because HYB, by its nature, requires only a half-inversion of the data and allows an efficient in-place instead of the traditional merge-based index construction. We also pay particular attention to the cache efficiency of the in-memory posting accumulation, an issue that has not been addressed in previous work, and show that our simple multilevel posting accumulation scheme yields much fewer cache misses compared to related approaches. Finally, we show that HYB supports fast dynamic index updates more easily than INV.

Categories and Subject Descriptors: H.3.1 [**Content Analysis and Indexing**]: Indexing Methods; H.3.3 [**Content Analysis and Indexing**]: Retrieval Models; H.5.2 [**User Interfaces**]: Theory and Methods

General Terms: Algorithms, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Autocompletion, indexing, index construction, inverted index, HYB index

ACM Reference Format:
Bast, H. and Celikik, M. 2011. Fast construction of the HYB index. ACM Trans. Inf. Syst. 29, 3, Article 16 (July 2011), 33 pages.
DOI = 10.1145/1993036.1993040 http://doi.acm.org/10.1145/1993036.1993040

16

1. INTRODUCTION

The *inverted index* (also known as *inverted file*) is still the standard indexing data structure for full-text search, and for good reason: it can be stored in little space with respect to the size of the original text (5%–10% for document-level index and 25% or more for word-level index [Witten et al. 1999; Zobel and Moffat 2006]), it can be constructed relatively fast, and it enables very fast full-text search. The inverted index consists of two main components: a *vocabulary* that is usually held in main memory and *posting lists*, that are usually held contiguously on disk, which is the key for fast I/O access and compression. Its one major shortcoming is that it supports only basic keyword queries efficiently, namely finding all documents that contain all or some of the given query words.

A large variety of data structures for advanced text search have been proposed as alternatives to the inverted index over the years. Prominent examples are suffix arrays [Manber and Myers 1990; Crauser and Ferragina 2002], suffix trees [Ukkonen 1995;

This article includes material presented in preliminary form at SPIRE'09.
The work is partially supported by DFG-SPP 1307, project Efficient Search in Very Large Text Collections, Databases, and Ontologies.
Authors' address: H. Bast and M. Celikik, Department for Computer Science, Albert Ludwigs University, Freiburg, Germany; email: celikik@informatik.uni-freiburg.de.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 1046-8188/2011/07-ART16 $10.00
DOI 10.1145/1993036.1993040 http://doi.acm.org/10.1145/1993036.1993040

Fig. 1. Search on DBLP, based on the HYB index, with autocompletion and faceted search.

Farach 1997], FM-indexes [Ferragina and Manzini 2005], signature files [Faloutsos and Christodoulakis 1984; Zobel et al. 1998], and data structures for multidimensional range searches [Arge et al. 1999; Ferragina et al. 2003]. One practical weakness of many of these data structures when it comes to full-text search is their construction. Compared to the inverted index the construction either takes much longer, or consumes substantially more main memory or disk space, or both.

Bast and Weber [2006] proposed a data structure alternative to the inverted index, called HYB, that was shown to be as compressible as the inverted index (INV), but provides efficient support for a certain kind of *prefix* or *autocompletion search* (see Section 2.2 for an example). It was also shown in the original paper as well as in a series of subsequent works (see Bast and Weber [2007] for a summary), how this special kind of prefix search allows fast processing of a large class of advanced queries, including:

—autocompletion search (suggest completion of the last query word that would lead to a good hit) [Bast and Weber 2006];
—faceted search and query expansion with a large number of synonyms [Bast and Weber 2007];
—fast error-tolerant search (find documents that contain mistyped versions of the keywords and vice-versa) [Celikik and Bast 2009a; Bast and Celikik 2010];
—database-style select and join [Bast and Weber 2007];
—fast semantic-search (answer basic SPARQL graph-pattern queries on an ontology) [Bast et al. 2007].

Figure 1 shows a screenshot of some of these features in action for the HYB-based search on `dblp.mpi-inf.mpg.de` (CompleteSearch DBLP).

It should be noted that prefix search and all these advanced types of queries are difficult for INV. The one question that was left open in these works was how to efficiently construct the HYB index. The construction described in Bast and Weber [2006] actually works by first constructing and then postprocessing an ordinary inverted index, yielding a total index construction for HYB that is about twice as much as that of INV.

In this article we show that HYB can be constructed at least as fast as INV, and often up to twice as fast. This is remarkable in two respects. First, because HYB is as practical but more powerful than INV with regard to efficient support of advanced queries. Second, because fast index construction for INV has been the subject of extensive research and is well understood, leaving little room for further improvement of the state of the art (see Section 2).

It should be noted that the power of HYB comes at a price: the smallest unit of processing is always a block, as described in Section 2.2. The size of such a HYB block is a small but constant fraction of the size of the whole collection. Each query word requires the scanning of at least one such block, whereas the time for INV is proportional to the length of the inverted list for that query word, which is very small when the word is rare. This price is justified for two reasons. First, queries often contain at least one frequent word, and in that case the query times of INV and HYB are in the same order of magnitude. Second, it is often good enough when all queries are fast, and of lesser interest that some queries are super-fast. For a more detailed discussion of this issue, we refer to Bast and Weber [2006].

As we will see, our index construction is single-pass in the sense that the bulk of the postings (around 95% in our experiments) are read from or written to disk only once in the whole process while each posting is permitted to be touched in memory multiple times (we explicitly point out throughout the paper whenever our algorithm needs to touch a posting on disk for the second time). The best construction algorithms for INV are single-pass in the sense that they make a single parsing pass over the original text data, whereas later passes of already (inverted and) compressed versions of that data on disk are not counted as additional passes.

Some of the material in this article has been presented before in a conference paper at SPIRE'09 [Celikik and Bast 2009b]. This article extends the conference paper in the following ways. Throughout the article, the introductory parts have been rewritten, in particular now giving more background information and a better overview over the respective material. The construction algorithm and data structures are now explained in full detail. We have provided proofs for the lemmas that were left without proofs in the conference paper. We have included a new section on I/O efficient sampling (Section 4.3); a new section that is an improvement of our in-place block writing algorithm (Section 6.2); a refinement for our word-id compression algorithm as well as a more thorough section about our cache-friendly posting accumulation, providing additional theoretical and experimental evidence (Section 5.1). In our experiments, we now compare *two* versions of INV: the Zettair implementation (as in the conference paper) and our own implementation following the state of the art, including an optimization that works for both HYB and INV (new). Our experiments are now done on a variety of platforms with varying ratios of CPU to disk speed. Furthermore, we include and compare to a merge-based version of our algorithm and include an experiment about the running time of our HYB construction when the block size increases. Finally, there are two additional new sections, about large vocabularies (Section 7.5) and about dynamic index updates (Section 8).

2. BACKGROUND AND RELATED WORK

2.1. The Inverted Index

An inverted index maintains the set of distinct words of a text collection in a *vocabulary*, with a *posting list* or *inverted list* assigned to each word. A posting list comprises the list of postings for that word and it is usually stored contiguously on disk. Each posting corresponds to a word occurrence and is of the form $(d, f_{d,w})$ or $(d, s_{d,w})$, where $f_{d,w}$ and $s_{d,w}$ represent the frequency and score of the word w in the document d, respectively. A

Table I.
Positional inverted index. A posting list consists of a word (term) and postings. Each postings is a triple consisting of a doc-id (first row), a position (second row), and a score

	D9002	D9002	D9002	D9004	D9004
algorithm	5	9	12	4	21
	0.8	0.8	0.8	0.2	0.2

word-level or positional index additionally includes the word positions in the document. Table I gives an example of an inverted list.

An extensive amount of research has been done on static inverted index construction and many inversion approaches have been proposed, although only few are scalable in practice [Witten et al. 1999]. We compare ourselves against the state-of-the-art inverted index construction proposed in Heinz and Zobel [2003] (referred to as the *single-pass approach*) which improves the well known *sort-based* approach, which is considered one of the most efficient approaches described in the literature [Heinz and Zobel 2003; Witten et al. 1999].

Both approaches are *in-memory*, as the inversion is done in main memory, and *single-pass*, as only one parsing pass over the uncompressed data is required (note that our definition of single-pass is stricter since we allow only a small number of word occurrences to be touched twice; see Section 1). Two-pass approaches on the other hand are slow but memory efficient, since the number of postings per indexed word is known. Hence the sizes of all in-memory and on-disk vectors can be easily calculated, and effective compression schemes can be applied [Heinz and Zobel 2003; Witten et al. 1999]. Disk-based approaches can invert collections of any size but suffer from excessive cost since traversal of on-disk linked lists of nonadjacent postings is required [Harman and Candela 1990; Rogers et al. 1995]. In-memory approaches, on the other hand, maintain an in-memory data structure for posting accumulation, either a linked list or a dynamically growing array. However, since text collections are typically much larger than the available main memory, the index is split into smaller runs, each of which is in-memory inverted and written out to disk. The final index is obtained by merging the on-disk runs through a multiway merge [Heinz and Zobel 2003; Moffat and Bell 1995]. An extensive amount of collected work on inversion approaches can be found in Witten et al. [1999].

The sort-based approach consists of the following basic steps: (i) a word to word-id mapping is maintained through hashing and the available memory is filled with postings that come from the incoming parsing stream; (ii) when the main memory limit is reached, the postings are sorted, compressed and written to disk as a new run; (iii) after the whole collection has been processed a multi-way merge is performed to obtain the final index. The merge can be performed either in-situ, for additional index permuting cost [Moffat and Bell 1995; Witten et al. 1999], or with a temporary file using roughly twice as much space as the size of the index.

The single-pass approach from Heinz and Zobel [2003] modifies steps (i) and (ii) as follows. First, instead of sorting postings, a dynamic array that accumulates the postings from the posting stream in compressed format is assigned to each index word; and second, words instead of word-ids are included in the runs and thus no word to word-id mapping is required. The advantage of the modified version is that sorting of a large amount of in-memory postings is avoided and that the vocabulary can be flushed to disk and the memory freed whenever a new run is written out to disk. The reported improvements range from 15% up to 20%.

In addition, Heinz and Zobel [2003] propose an interesting idea to reduce the merge cost that is similar to our construction algorithm: the index is partitioned into b buckets so that each bucket is responsible for a nonoverlapping range of lexicographically

[alg, alz)	D9002 algorithm 5 0.8	D9002 algorithmica 9 0.8	D9002 algorithmic 12 0.8	D9003 algae 54 0.9	D9004 algorithm 4 0.2

Fig. 2. Positional HYB index. A HYB block consists of a word range formed by two consecutive block boundaries. Each postings is a quadruple consisting of a doc-id (first row), a word-id (second row), a position-id (third row) and a score.

adjacent index words. The ranges of words are determined by building an in-memory inverted index for a snapshot of documents from the collections and then splitting the accumulated vocabulary into ranges so that each range corresponds to a set of posting lists of roughly the same memory size. Once the memory is exhausted a block is flushed to disk as a run. Hence each bucket is an index on its own small enough to fit in memory so that it can be independently in-memory merged. The improvement in the running time is, however, only marginal. The reason for this is that the improvement aims at reducing the number of disk seeks during the merging, which not the bottleneck in the state of the art inverted index construction (see Section 7.3.2).

2.2. The HYB Index

The HYB index does not maintain an explicit word-level vocabulary but rather a vocabulary of word ranges. The number of word ranges is typically a few thousand times less than the number of distinct words. Each word range corresponds to a *block* of postings (HYB block) that is analogous to an inverted list. The word ranges are defined by a sequence of *block boundary words* $w_0, \ldots, w_k$ such that block i contains all words in the range $(w_{i-1}, w_i]$, where w_0 is some word lexicographically smaller than all words in the collection. A posting in a HYB block is a document-id, word-id, position, score quadruple. In each HYB block, the postings are sorted by document-id and position (not by word-id). Both documents and words have contiguous ids. Word-ids are assigned to words in lexicographical order; this is key for the fast processing of prefix queries with HYB. For an example of a block that corresponds to the word range (alg, alz] see Figure 2. In this example, the first list entry says that the word "algorithm" occurs in a document with ID D9002 at position 5 with a score of 0.8.

One of the key results from Bast and Weber [2006] is that if these blocks are of roughly equal volume $\varepsilon \cdot N$, where N is the number of documents and ε is some constant, then HYB can be stored in space $1 + \varepsilon$ times that of INV.

The HYB index allows efficient computation of *autocompletion* queries. Formally, an autocompletion query is a pair (D, W), where W is a range of words (all possible completions of the last query word) and D is a set of documents (the hits of the preceding part of the query). A result of an autocompletion query is a subset $W' \subseteq W$ of words that occur in D, as well the subset $D' \subseteq D$ of their matching documents. Both D' and W' should be computed in ranked order. Less formally, imagine a user of a search engine typing a query. Then with every letter being typed, in time less than it takes to type a single letter, we would like a display of completions of the last query word that would lead to good hits. At the same time the best hits for any of these completions should be displayed. For example, promising completions for the query `index con` might be `index construction`, `index configuration`, etc., but not, for example, `index constitution`, assuming that, although `constitution` by itself is a frequent word, the query `index constitution` leads to only few good hits.[1]

Processing an autocompletion query with the HYB index goes in three simple steps as follows.

[1]See `http://search.mpi-inf.mpg.de` for a collection of online demos.

(1) Find each block B that correspond to a (precomputed) word range that overlaps with the last part of the query (it is not hard to prove that there is only one such block in average);
(2) Intersect each B with D to compute W'_B and D'_B;
(3) Compute D' by a multiway merge and compute W' by a simple linear-time sort into W buckets.

Another key result from Bast and Weber [2006] is that if the blocks are chosen with sufficiently large volume, then the average autocompletion query processing time is linear in the number of documents containing it. For proofs and more details (e.g., document and word ranking) the reader should refer there.

3. ALGORITHM OVERVIEW

To be able to construct a HYB index we must know the block boundaries in advance so that at parsing time we can determine the block to which a given word-id belongs. This could be trivially done by a full pass over the data, counting the frequency of each word and then computing the prefix sums. Inspired by parallel sorting algorithms, in Section 4 we show how to compute good estimates of the block boundaries by sampling only a logarithmic number of random passages in the given document collection.

Given the sequence of block boundaries, a straightforward approach to building the HYB index with a single pass would be as follows. Maintain a dynamically growing in-memory data structure of postings for each block, and for each word parsed, append the corresponding posting to the array of the block to which it belongs. When all words have been parsed, compress the blocks one after the other, and write them to disk. As is the case with the inverted index construction, the first obvious problem is that the size of the in-memory data structures will at some point exceed the total available memory. An obvious solution is to process the collection in parts by imposing a limit on their in-memory size. Once the collection is fully scanned, the fragmented parts (partial blocks) should be merged to produce the final index. This simple approach has the following efficiency issues.

The first efficiency issue is index merging since a full additional pass over the compressed data is required. We show that unlike INV construction, for the HYB construction we can avoid index merging by appending the in-memory blocks to their corresponding positions on disk in-place. This process is explained in more detail in Section 6.

A second efficiency issue not considered in the previous work of Heinz and Zobel [2003] is the cache efficiency of the posting accumulation. Namely, even though well approximated by Zipfian distribution and hence with a good locality of reference, we show that a significant fraction of the postings impose cache misses when appended to their inverted lists (HYB blocks). This is due to the fact that the number of inverted lists is typically much larger than the number of lines of the L1 data cache. In Section 5.1 we propose a multilevel posting accumulation scheme with better locality of access that can significantly improve the in-memory inversion performance for both, HYB and INV.

A third efficiency issue is the compression of word-ids. Unlike INV, HYB requires storing the word-ids as a part of each posting. However, the word-ids cannot be gap-encoded as they come in random order and have to be entropy-encoded instead. Near entropy-optimal compression could be, for example, achieved by arithmetic encoding [Witten et al. 1999]. In Section 5.2 we propose a much faster two-pass compression scheme that yields only a slight loss in the compression ratio. Furthermore, storing word-ids requires a permanent in-memory word to word-id mapping. Section 5 makes the simplifying assumption that the vocabulary fits in main memory. In Section 7.5 we propose a refinement of our basic algorithm that addresses this issue.

Table II.
Standard deviation of the block sizes computed by a full pass given as a percentage of the ideal block size n/k, where $k = 2000$ for all three collections

	DBLP	Wikipedia	TREC GOV2
Standard deviation (%)	12.7%	1.6%	6.4%

In Section 7 we experimentally compare our HYB index construction algorithm against the state-of-the art inverted index construction algorithm from Heinz and Zobel [2003]. As a reference, we also compare ourselves to the very fast and well-engineered Zettair system. Finally in Section 8 we show how the HYB index can support efficient index update operations.

4. COMPUTING BLOCK BOUNDARIES

We already mentioned that the block boundaries of the HYB index can be easily computed by scanning the whole collection. Having the word frequencies, the boundaries are the words from the vocabulary that split the collection into unions of posting lists with roughly the same number of occurrences. In this section we show how to compute the block boundaries by only a logarithmic number of accesses to the given collection. The basic idea is related to the idea of splitter selection in the parallel sorting literature (Samplesort, Grama et al. [2003]). Let n be the collection size in total number of occurrences and let k be the number of HYB blocks. The sampling lemma below shows how to compute block boundaries from a sample of word occurrences so that the resulting blocks are of size less than $a \cdot n/k$ with high probability, where $a > 1$ is an arbitrary constant and n/k is the ideal block size.

4.1. Sampling Lemma

LEMMA 4.1. *Pick $s \cdot k$ numbers from the range $1..n$ uniformly at random and independently from each other. Sort these numbers and consider the k integers $x_1, \ldots, x_k$ whose rank in the sorted sequence is a multiple of s. Let $B_1, \ldots, B_k$ be the block sizes induced by splitting the range $1 \ldots n$ according to $x_1, \ldots, x_k$. Let $B_{max} = \max\{B_1, \ldots, B_k\}$. Then*

$$Pr(B_{max} > a \cdot n/k) \leq n \cdot exp(-s \cdot K) \tag{1}$$

where $K \approx a - ln(a) - 1$.

PROOF. For brevity the proof is given in the appendix. We note that there is an alternative proof of the lemma based on Chernoff bounds with a looser upper bound that for the same failure probability requires close to twice as many sampled words. □

Example 4.2. Let $k = 2000$, $a = 1.5$, $n = 10^{10}$ (a dataset with 10 billion occurrences) and a failure probability of 10^{-10}. Then a sufficiently large s so that $Pr(B_{max} < 1.5 \cdot n/k) \leq 10^{-10}$ is 512, which means that a sample of 0.01% of the full collection is enough. Moreover, increasing the dataset by 10 times requires increasing s by roughly 30 (or 6% which is 60,000 more occurrences in total).

We note that for the above calculations and discussion, we assume that the list of all occurrences (considered sorted by word) can be split into blocks at arbitrary positions. However, in practice we do not put the word boundaries in the middle of a run of occurrences of the same word. This gives blocks of size at most those predicted by the lemma, except for very frequent words, which get a block on their own, and which in principle can become arbitrarily large, depending on the frequency of the word(s). Nevertheless, it is desirable in practice that very frequent prefixes such as `pro`, `com`, `the`, etc. get blocks on their own. Table II shows the standard deviation of these real block sizes from the ideal n/k.

We also note that although sampling causes some word occurrences to be visited twice, it does not violate our definition of single-pass since the number of drawn samples is very small compared to the total number of postings in the collection (about 2% in all our experiments).

4.2. Query Time

The following lemma says that the blocks obtained via sampling give a query time on the same order of magnitude as we would obtain it for the ideal blocks.

LEMMA 4.3. *Assume block boundaries are computed as in Lemma 4.1. Then, for an arbitrary given query, the query time is $O(q \cdot n/k)$ with high probability, where q is the number of blocks that contain a word/prefix from the query.*

PROOF. By Lemma 4.1 the maximal size of a block is $O(n/k)$ with high probability. The lemma then follows from the observation that each query word requires a scan of each block containing that query word. □

In practice, blocks are quite large (there are only a few thousand blocks in all our experiments). Therefore, typically only one block needs to be processed per query word and q is just the number of query words.

Also note that the above lemma holds for block boundaries that split the blocks at arbitrary positions. Again, for the actual block boundaries, a query word that lies in a block containing a very frequent word will require larger processing time, depending on the size of that block. In practice, we either accept these larger processing times, or we remove very frequent words (stop words) from the index.

4.3. IO-Efficient Sampling

Lemma 4.1 asks for a random sample of postings with each posting drawn uniformly and independently of other postings. The straightforward algorithm for drawing such a sample would require one random disk seek per posting. This can be prohibitively expensive if the sample size is large: to pick a sample of 0.01% of all postings in the previous example would take around 10,000 secs, in that time we could scan 500GB of data. We therefore use a variant of random sampling, namely, *cluster sampling* [Cochran 1977], where for each random access we read a whole passage of text from disk and draw a random sample of postings from this passage. Let P be the number of sample passages (number of affordable disk seeks) and p be the probability of drawing a certain occurrence within a passage. The passage size at each access then must be equal to $s \cdot k/(P \cdot p)$ occurrences. Table III shows the running time of random and area sampling for different sample sizes. Obviously, the main cost in area sampling comes from the number of passages and not from the size of the sample.

The assumption behind cluster sampling is that the passages have approximately equal distributions. As this is not always the case in practice, the price paid is a larger sampling error [Cochran 1977]. The sampling error in our case depends on P and p. A good trade-off between the sampling error and the sampling time was achieved for $P = 5000$ and $p = 0.1$. Smaller values of p did not seem to involve a significantly better approximation of the block boundaries. Table IV shows that computing the block boundaries using cluster sampling with these values of the parameters results in block sizes reasonably close to those computed by a full pass. We measure the deviation in two ways: (*i*) by the standard deviation of all block sizes from the ideal block size; and (*ii*) by Kullback-Leibler divergence, considering the block sizes obtained by both sampling methods as histograms. The intuition here is that if area sampling is biased and significantly deviates from uniform random sampling, then the resulting "distribution" of block sizes computed by sampling will significantly deviate from the

Table III.
Running time of random disk sampling compared to area disk sampling for different sample sizes and number of passages (accesses). The ratio between the passage size and the number of samples drawn is fixed to 0.1

Number of samples	Number of passages	Area sampling time	Random sampling time
10^4	1000 5000 10000	4.3 sec. 19.7 sec. 40.1 sec.	42.1 sec.
10^5	1000 5000 10000	4.6 sec. 21.5 sec. 40.4 sec.	302.9 sec.
10^6	1000 5000 10000	7.1 sec. 27.1 sec. 47.5 sec.	886.7 sec.

Table IV.
Standard deviation (given as percentage of the mean block size) calculated over all HYB block sizes when computed with sampling (random and area, defined in Section 4.3) and with a full pass. The last column shows the Kullback-Leibler divergence when the block size are considered as probability distributions (see last paragraph of Section 4.3). All numbers are averages of 100 experiments carried out on 1GB dataset with 2000 word boundaries (HYB blocks)

Sampling	Sample Size	Standard deviation (%)	Kullback-Leibler div.
Full dataset	100%	12.7%	0
Random	0.06% 0.6% 1.2%	15.1% 13.0% 12.8%	1.52 1.45 1.37
Area	0.06% 0.6% 1.2%	15.2% 12.9% 12.7%	1.43 1.45 1.46

"distribution" of block sizes computed by a full pass. We observed essentially the same results on all of our test collections.

5. IN-MEMORY HYB INVERSION

The posting accumulation from Heinz and Zobel [2003] works by assigning a single dynamic array to each inverted list. The postings coming from the parser are then stored by using on-the-fly compression. Dynamic arrays (also dynamic bit-vectors or doubling vectors) are in fact the standard, as well as the optimal, choice for dynamically growing data structures for index construction [Zobel and Moffat 2006]. Note that any array data structure could serve as a bit vector, for example `vector<int>` in C++/STL. Buttcher and Clarke [2005] observed that by maintaining the array's dynamic growth with a special function, almost as 75% additional space can be saved as when the array size is known in advance.

5.1. Multilevel Posting Accumulation

A weakness of the above scheme is its cache inefficiency since the number of inverted lists is usually in the order of millions while the number of L1 data cache lines is in the order of hundreds. Even though the distribution of postings is Zipfian and exhibits a good locality of access, we experimentally show that a significant fraction of the postings will impose cache misses when appended to their lists. As we will see later, this badly hurts the efficiency of the in-memory part of the index construction. Interestingly, to the best of our knowledge, this issue has not been studied in previous work. Note that

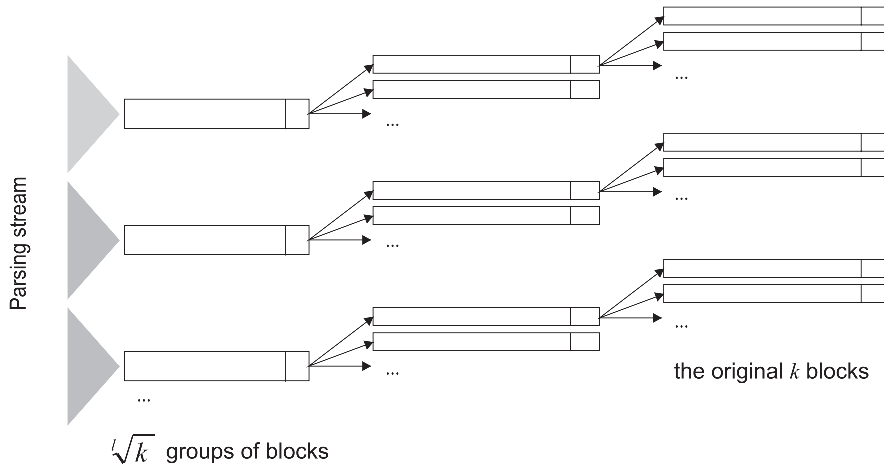

Fig. 3. Multilevel posting accumulation.

since the number of HYB blocks is much smaller than the number of inverted lists, the cache inefficiency of HYB posting accumulation is less severe.

We propose a multilevel posting accumulation scheme by grouping consecutive HYB blocks in groups so that the total number of groups is close to the number of L1 data cache lines. The postings from each group are, in a tree-like fashion, assigned to the next level of groups until each group is comprised of a single block. The idea is to assign each group to only a small number of groups at the next level of groups so that at each assignment of postings there are virtually no cache misses. If l levels of assignment are required to achieve this, then the number of groups per level should be $\sqrt[l]{k}$. To illustrate this by example, say we have 2000 blocks and 50 cache-lines. Then $l = 2$ is sufficient since $\sqrt{2000} < 50$. At the first level of assignment a posting that belongs to block i will be assigned to group $\lfloor i/\sqrt{2000} \rfloor$. At the second level of assignment, the postings coming from group j will be assigned to the blocks ranging from $j \cdot \lfloor \sqrt{2000} \rfloor$ to $(j+1) \cdot \lfloor \sqrt{2000} \rfloor$. Figure 3 gives an overview of this process. The price paid is moving each posting more than once. Note, however, that a cache hit can be several times faster than a cache miss.

In the following we give a theoretical evidence that under certain assumptions two-level posting accumulation is significantly faster than one-level posting accumulation.

LEMMA 5.1. *Consider a fully associative cache of size c and HYB blocks of equal size. Let a cache miss take m times longer than a cache hit. Two-level posting accumulation is then faster by a factor of $m/2 - g$, where g is usually a small function that depends on k.*

PROOF. The relatively simple proof is based on calculation of the total number of cache misses according to the assumptions and it is given in the appendix. □

Example 5.2. Consider an 8KB cache with 64B cache lines. Assume that an L1 cache hit requires 2 cycles while 8 cycles are required for a cache miss (which would still be an L2 cache hit due to the 2-level cache hierarchy). Then according to the given model, the two-level posting accumulation is roughly 2 times faster when $1{,}000 \leq k \leq 10{,}000$.

Indeed, the best result in practice was achieved for two-level posting accumulation. The left side of Figure 4 shows the number of cache-misses of one and two-level HYB

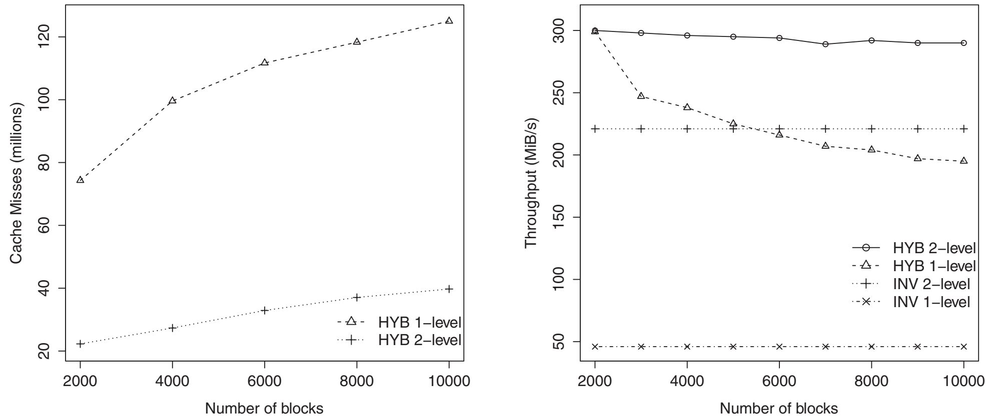

Fig. 4. Left: Total number of cache misses for 1 and 2-level HYB posting accumulation for total of 300,000,000 accesses (64K, 2-way associative cache, with 64B cache lines). Right: Throughput (given in MiB/s) of HYB and INV posting accumulation approaches used to in-memory invert 100 million occurrences for $\alpha = 1.0$ (defined in Section 5.1). Each posting is assumed to have 4 bytes.

Table V.
Throughput given in MiB/s for one-level posting accumulation, two-level posting accumulation and sorting-based inversion on the inverted index (INV) for different values of the α parameter of the Zipfian distribution of word occurrences

$\alpha = 1.0$		$\alpha = 1.1$		$\alpha = 1.2$	
sorting:	50MiB/s	*sorting*:	60MiB/s	*sorting*:	73MiB/s
$l = 1$:	46MiB/s	$l = 1$:	84MiB/s	$l = 1$:	150MiB/s
$l = 2$:	221MiB/s	$l = 2$:	295MiB/s	$l = 2$:	314MiB/s

posting accumulation on a 2-way associative, 64KB L1 data cache with 64B cache lines, computed using the `valgrind` tool.[2] The right side of the figure shows a comparison of the efficiency of one and two level posting accumulation for both, INV and HYB. The efficiency of one-level HYB posting accumulation is highly affected by the number of blocks and in fact can approach that of INV for a large number of blocks and small cache sizes. The efficiency of the two-level posting accumulation, on the other hand, is only slightly affected.

The number of cache misses in the posting accumulation for INV depends on both, the number of inverted lists and the value of the α parameter of the Zipfian distribution of word occurrences. According to Breslau et al. [1999] the asymptotic cache hit ratio of a cache with c cache lines is given by $H_{c,\alpha}/H_{n,\alpha} \approx (c^{1-\alpha} - 1)/(n^{1-\alpha} - 1)$ where $H_{n,\alpha} = \sum_{i=1}^{n} 1/i^{\alpha}$ is the generalized harmonic number of order n of α and n is the number of inverted lists. However, after grouping the inverted lists in groups, the distribution of occurrences in the groups is not Zipfian anymore and thus the total cost is hard to approximate. Still, the results given in Table V show that the improvement on INV is in fact more dramatic than that of HYB when two-level posting accumulation is used and varies from a factor of 2 for $\alpha = 1.2$ to a factor of 5 for $\alpha = 1.0$. One caveat here is that two-level posting accumulation is not easy to apply on INV, since all distinct words are not known in advance. Another caveat is that the actual improvement depends on the amount of processor cache available.

[2] `http://valgrind.org`

5.2. Fast Compression of HYB Blocks

Consider the following example of a HYB block that corresponds to the word range [5,133):

3	3	3	5	5	5	5	5	6	6	6	6	8	8	8
1	4	5	2	5	9	11	15	7	8	12	15	7	12	15
7	122	5	113	7	24	7	122	88	93	7	113	122	5	93

The first row corresponds to doc-ids, the second to positions and the third to word-ids (each word in the block is replaced by its word-id). One of the differences to INV is that the word-ids are part of the postings and have to be compressed just like the doc-ids and the positions. The latter means that a fast compression of word-ids is required. But unlike the doc-ids and the positions, which come in ascending order and could be compressed by gap and universal encoding (just as done for INV), the word-ids come in random order and have to be entropy encoded. One problem with entropy-optimal encoding like Huffman or arithmetic encoding is their speed. Gap combined with universal encoding, on the other hand, is a fast coding scheme which involves simple word-level arithmetic operations [Moffat and Zobel 1996].

The compression scheme that we propose is based on the assumption that the word-ids have near-Zipfian distribution. Given this, it is not hard to show that universal encoding that assigns $\sim \log x$ bits to a word of rank x obtained by sorting the word-ids in order of descending frequency, is near entropy optimal as well. To see this simply observe that if there are k distinct words in a block and if the relative frequency of the i-th most frequent word is Ω/i, then the per-item entropy is equal to

$$\sum_{i=1}^{k} \frac{\Omega}{i} \cdot \log_2 \frac{i}{\Omega},$$

while the average per-item-space consumption would be equal to

$$c \cdot \sum_{i=1}^{k} \frac{\Omega}{i} \cdot \log_2 i \qquad (2)$$

for some (small) constant factor $c > 1$. We refer to this scheme as *Zipf compression*. The scheme requires three passes: a pass to obtain the frequency counts (linear in the number of word occurrences), a pass to sort the distinct word-ids with respect to their frequency counts (super-linear in the number of distinct word-ids) and a third pass where each word-id is replaced by its rank. Assume that instead of sorting, the rank of each word is approximated by a move-to-front (MTF) transform over the input which can be computed more efficiently in a single linear scan of the list. In a move-to-front transform, the next element is always appended at the beginning of a list with the intuition that frequent elements are likely to end up being not far from the front of the list and thus obtain smaller ranks. The following lemma shows that this gives us an efficient compression algorithm at the price of a small loss in the compression ratio.

LEMMA 5.3. *Given a Zipfian distribution of the input with independent consecutive elements, the ranks obtained by a MTF-transform have expected values that are not much larger than the true ranks (in particular, the ranks are less than a factor of* $1/\Omega$ *away from the true ranks, that is, more skewed distribution results in ranks that are closer to the true ranks).*

Table VI.
Rate given in MiB/s of our MTF zipf compression (without the proposed refinement at the end of the section) compared to zipf compression with sorting and gap encoding with Elias-gamma code for compressing positions (taken as a reference)

Compression	Zipf with sorting	Zipf with MTF	Gap + Elias gamma coding
Rate	135MiB/s	343MiB/s	155MiB/s

PROOF. Let X_r be a random variable defined as the rank of a word w obtained by a MTF-transform over the input. Observe that the value of X_r is equal to the number of words in front of the first occurrence of w, ignoring duplicates (counting from the end of the original input list). Let j be the true rank of w (obtained by sorting) and let X_j be a random variable defined as the number of words (including duplicate words) in front of the first occurrence of w. Let $p = \Omega/j$ be the probability of observing w, where $\Omega < 1$ is a normalization constant that depends on the distribution. Observe that X_j has a geometric distribution with expectation $E[X_j] = 1/p = j/\Omega$. This gives us an upper bound on the expected value of X_r as $X_r \leq X_j$. □

According to Lemma 5.3, the average per-item-space consumption is now less than

$$c \cdot \sum_{i=1}^{k} \frac{\Omega}{i} \cdot \log_2 \frac{i}{\Omega},$$

which is only slightly larger than the value given in Equation (2) (note that $\Omega < 1$). The loss in compression ratio due to the MTF transform on our test collections in practice was not far from what is predicted by our model: less than 1% more total space required when indexing Wikipedia and about 1% more total space required when indexing TREC GOV2 compared to when the ranks were obtain with sorting the word-ids. Regarding the speed, our word-id compression scheme is twice as fast as coding word positions with gaps and Elias gamma code (see Table VI).

The given lemma does not ensure that the few most frequent word-ids that make most of the input get their true ranks with high probability. We propose the following procedure as one possible refinement that is somewhat less efficient and harder to implement. We pick a small random subset of all word-ids with size large enough so that the f most frequent words are contained with high probability (f is small here, for instance 3 or 4). The probability that the i-th most frequent word-id is in a random subset of size s is equal to

$$1 - \left(1 - \frac{\Omega}{i}\right)^s \approx 1 - \exp\left(-\frac{\Omega}{i} \cdot s\right). \tag{3}$$

To ensure that all of the f words are in the subset with high probability, we should pick s such that $S \gg f/\Omega$ (e.g. for $S > 5 \cdot f/\Omega$, Equation (3) has value that is already larger than 0.99). Say $W_1, \ldots, W_{f_1}$, where $f_1 \geq f$, are the distinct words from the subset. We reserve the first f_1 ranks and allocate an array $Count[1..f_1]$ (initially set to 0). We maintain a mapping $g : x \mapsto i$, for $x = W_i$ and $g : x \mapsto -1$ for $x \neq W_i$ so that we can count the frequencies of $W_1, \ldots, W_{f_1}$ by increasing $Count[g(W_i)]$ as we go through the list and compute the MTF transform. At the end we assign the ranks of $W_1, \ldots, W_{f_1}$ computed by sorting their frequencies and keep the ranks obtained by the MTF transform for the rest of the word-ids. The latter can be done by maintaining another mapping $r : x \mapsto$ rank of x for $x = W_i$ and $r : x \mapsto -1$ for $x \neq W_i$. We note that all computations required for this algorithm are done in-memory and no posting is required to be revisited on disk.

5.3. Algorithm Engineering of the In-Memory Inversion

Our algorithm uses two-level posting accumulation and assigns dynamically growing arrays or bit-vectors to each group of HYB blocks. The doc-ids and the positions are compressed on the fly while the word-ids are accumulated without compression and compressed right before being written to disk.[3] Elias-gamma code is used to compress the doc-gaps, position-gaps and word-frequencies, and Zipf compression from Section 5.2 to compress the word-ids. We note that the groups of HYB blocks are still well compressed because, as shown by Bast and Weber [2006], the HYB index with positional information has smaller empirical entropy[4] than that of the inverted index for any choice of block size. Once the memory limit of the in-memory HYB blocks has been reached, the groups of HYB blocks are processed one by one by decompressing each group and restoring and (re)compressing the individual HYB blocks. To encode the document and position gaps in the HYB groups we use variable-byte encoding due to its speed.

To maintain the word-to-word-id mapping, a fast hash-table based vocabulary [Heinz and Zobel 2002] is employed (an alternative to permanent vocabulary is discussed in Section 7.5). To determine the corresponding HYB block for each posting, a data structure is employed that computes the HYB block efficiently as follows. For each 3-gram, the interval in the lexicographically sorted list of word boundaries that contain that 3-gram as prefix is precomputed and stored. Whenever a new posting comes, its prefix is looked up and if the resulting interval contains more than one word, a binary search is performed to find the true word boundaries (the intervals usually contain small number of boundary words, for example the most frequent 3-letter prefix in the 2000 boundary words of the GOV2 collection was `int` with 11 occurrences. All other 3-letter prefixes typically occurred less than 3 times). The newly computed ID of the HYB block is then stored in the word's vocabulary entry so that the computation is done only once per each distinct word.

The single-pass approach by Heinz and Zobel [2003] must sort the index words whenever a new compressed run is being written out to disk, as otherwise merging of the temporary inverted lists would not be possible. Note that while the HYB construction does not require word sorting, the word boundaries are already precomputed in fixed (lexicographic) order. Moreover, the postings that correspond to each HYB block are already in doc-id order and do not require sorting either.

6. ON-DISK HYB INVERSION

As we have discussed in the related work section, index construction usually operates in two passes. In the first pass each collection is divided into n parts, where the size of each part is defined by the available main memory. Then a partial in-memory index is created, which, once the memory limit is reached, is transfered contiguously to disk. The price for contiguous disk access in the first pass is paid in the second pass where the on-disk inversion takes place. At this point the partial indexes must be merged together through n-way merge to obtain the final index. In the following we will show that sacrificing contiguous disk access in order to avoid index merging usually pays off on the HYB index.

An advantage of the HYB index is that the HYB blocks are of relatively similar sizes and reasonably large in number. This allows us to construct the HYB index in a single pass as follows. In an initial step we compute an estimate for each block size by running an in-memory version of our algorithm on an already sampled set of documents (see

[3]The word-ids must be stored before compression since they are entropy-encoded.

[4]The empirical entropy defined in Bast and Weber [2006] estimates the inherent space complexity of the HYB and the inverted index independently of the specialties of a particular compression scheme.

Table VII.
Average random and equidistant seek time over 2000 disk seeks for different file sizes (the distance in the second row is 1/2000-th of the file size)

File size	100MB	1GB	10GB	100GB
Average random seek time	1.5ms	5.2ms	11.1ms	14.5ms
Average equidistant seek time	0.8ms	4.7ms	5.6ms	6.2ms

Section 4.1). Next, we create the index file and reserve enough disk space for each block. Once the available memory is used up, we process the HYB blocks one by one, seeking to the disk location of the current partial block and writing it in-place. The index construction finishes once the entire collection has been scanned. Note that we do not pay the full cost for each disk seek since the seek time depends on the track distance [Popovici et al. 2003]. Our algorithm makes jumps with comparably similar distances compared to the size of the index, and hence keeps the track distance small e.g. about 10MB in average for a compressed index of size 20GB. Table VII shows this empirically. A subtlety here is a slight inherent nonlinearity of the disk cost, as the seek time slightly increases with the index (collection) size (which is reflected in our results). This behavior, however, does not affect the running times seriously on collections of the order of TREC GOV2.

The one potential problem of this approach are the blocks that require more than the estimated space. A straightforward solution is to simply leave enough free space after each HYB block on disk. We will see later that this is in fact necessary for fast index update operations when the collection is dynamic (see Section 8). Another simple solution is to use a separate file for each HYB block. This solution will, however, increase the construction cost as well as the block reading cost at query time since the operating system usually does not guarantee that the physical blocks are indeed laid out contiguously on disk. A third solution that does not affect the indexing performance significantly is a procedure that we call *space propagation*, described in detail in Section 6.3.

Note that in-place writing is impossible to achieve on the inverted index, first, because most of the inverted lists are short and their size cannot be reasonably well estimated and second, because the number of inverted lists is in the order of millions, making in-place list writing not worthwhile.

6.1. Disk Cost of In-Place and Merge-Based Inversion

In the following we show and experimentally confirm in Section 7, that in-place HYB index construction is more efficient than the standard paradigm of two or three passes over the data depending on whether the merge is in-situ or not. Let C and U be the collection size in compressed and uncompressed format respectively. Let R_x be the cost of sequentially reading x bytes, W_x be the cost of sequentially writing x bytes, T_s be the disk seek time and B be the size of the in-memory buffer used for in-memory posting accumulation as well as index merging (reducing the number of disk seeks by allocating a part of the buffer to each on-disk run). The total disk cost for merge-based index construction is then roughly equal to

$$R_u + \left(R_c + W_c + \left(\frac{C}{B}\right)^2 \cdot T_s\right) + (R_c + W_c) + W_c \quad (4)$$

$$= R_u + 3 \cdot W_c + 2 \cdot R_c + \left(\frac{C}{B}\right)^2 \cdot T_s. \quad (5)$$

The second and the third terms in the left-hand side correspond to the cost to merge and permute the merged file. The last term in the right-hand side corresponds to the disk seek cost, where C/B is the total number of runs. If twice more disk space than the size of the index file is allowed (the merging is not in-situ), then the third term should be skipped (note that our approach requires almost no additional disk space; see Section 7.4). Let k be the total number of HYB blocks and B be the buffer size for in-memory posting accumulation. The total disk cost for the in-place HYB construction is then equal to

$$R_u + \left(W_c + k \cdot \frac{C}{B} \cdot T_s \right), \tag{6}$$

since the collection should be read and written in compressed format only once. The second term in the brackets corresponds to the total seek time since each block requires a single disk seek for each run. Even though the number of disk seeks is increased by a factor of $k \cdot B/C$, the disk transfer cost $2 \cdot (W_c + R_c)$ is usually the dominating part in Equation (4).

Example 6.1. Consider a 50GB collection with 20GB of compressed (positional) index. Let the disk throughput be 50MB/s and let a single disk seek take 5ms. Let the memory limit in both cases be 1024MB (20 runs). For $k = 2000$ the disk costs for INV and HYB are 3072 and 1633 secs, respectively. Furthermore, the disk seeks take around 14% of the total disk cost of the HYB construction.

One way to reduce the number of disk seeks required for in-place index construction is by either allowing a larger memory buffer or by using a better in-memory compression (e.g., Golomb code instead of Elias-gamma). In the next section we propose a procedure called *partial block flushing* that can additionally reduce the number of disk seeks.

6.2. Partial Block Flushing

To make room for further incoming postings, our basic in-place index construction flushes each HYB block to disk once the memory limit has been reached. Hence $C/B \cdot k$ disk seeks in total or k disk seeks per run are required. In this section we show how the total number of disk seeks required can be reduced by postponing the flushing of a certain set of blocks while flushing the rest. To achieve this, we use the observation that those blocks that correspond to stop-words are significantly larger than the rest of the blocks. For example, typically around 20% of the largest HYB blocks take half of the total memory. The idea of partial flushing has been already used in Büttcher and Clarke [2008] for speeding up index update operations. In this section we provide an analysis of the effectiveness of the procedure for a given discrepancy between the block sizes.

Our partial block flushing works as follows. All blocks are initially divided into a set of large and a set of small blocks. Let L_1 be the number of large blocks and L_2 the number of small blocks, where $L_1 + L_2 = k$. Once the memory limit has been reached, all large blocks are transfered to disk and the memory freed while all small blocks are left untouched. Whenever the size of the small blocks reach a fraction $T < 1$ of the memory, all blocks are transfered to disk and the entire memory is freed. To analyze the number of disk seeks required, assume that the small blocks occupy a fraction $f \leq T$ of the total memory and let $m = k/L_1$. Then it is not hard to show the following

LEMMA 6.2. *Partial flushing reduces the total number of disk seeks required for constructing a HYB index by at least a factor of*

$$\frac{1}{2} \cdot \frac{m + f \cdot m}{1 + f \cdot m}.$$

PROOF. The relatively straightforward proof can be found in the Appendix. □

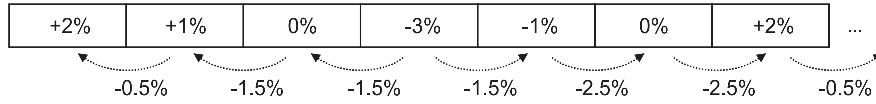

Fig. 5. Space propagation. The rectangles corresponds to blocks, where the percentage inside the block is the estimation error for that block ("+" indicates overestimation and "−" indicates underestimation). Below is the propagated space of the blocks.

Clearly the advantage of partial flushing increases when the number of large blocks is small (large m) and when the small blocks take only little memory (small f).

Example 6.3. If $f = 10\%$ (the short blocks occupy 10% of the memory) and the number of large blocks is 10% of the number of the small blocks, the number of disk seeks will be reduced by a factor of $4.95 \cdot L_2/1.6 \cdot L_2 \approx 3$.

The gap between the large and the small blocks in practice is less extreme. The improvement that we achieved due to partial flushing on our test collections ranges from around 25% (for $k = 2000$) to around 40% (for $k = 10000$) less disk seeks on TREC GOV2 and Wikipedia and from around 31% to around 50% less disk seeks on DBLP.

6.3. Avoiding Block Collisions by Space Propagation

We already mentioned the problem that blocks with initially underestimated sizes will overflow and overwrite the neighboring blocks. Note that due to estimation errors, half of the block sizes in average will be underestimated. Also note that once the block writing starts, further movement of the blocks is not possible. A solution to this problem that does not affect the query and the indexing performance is to allow the underestimated blocks to make use of the space of the overestimated blocks without splitting them in two parts. In the followings we give a description of the procedure and provide theoretical evidence that it fails only with small probability given that certain assumptions are satisfied.

The idea is to shift the unused space of the overestimated blocks towards the underestimated ones by permitting an underestimated block to borrow space from its neighbor. If the lending block is not large enough to fit its own data plus the additional data of the borrower, it becomes a borrower, too (of the next neighbor). Hence, a cascade of blocks that borrow space from their neighbors can be formed. The cascade terminates once a large enough block to amortize for the propagated space request is encountered (note that the propagated space along the cascade of blocks can increase or decrease; see Figure 5).

To determine if a certain block will overflow or not, after a significant fraction of the data has been processed, each block size is reestimated every time a new run is being written to disk. After each reestimation the newly estimated block sizes are compared to those that have been initially estimated. If the estimated block size is less than the space allocated for that block, then the left or the right boundary of the block is moved towards the next block and the block is extended for the size of the current run. Note that this does not require actual moving of blocks on disk, but merely writing the new run before or after the block.

The space propagation at certain block fails if both neighbors of that block have free space that is less than the required. Note that instead of writing the runs at the beginning of the block, the runs are written in the middle of the allocated space and alternated from left to right so that both sides of the block grow equally fast. This requires defragmenting the blocks once they are read from disk by decompressing each run separately. Also note that even though the blocks are of roughly the same size, there is a possibility that a certain block is too small for the space propagation demand

of its larger neighbors. To address this, all blocks are initially sorted with respect to their estimated size. This increases the chance that neighboring blocks are of roughly the same size and will not interrupt the propagation.

Observation 6.4. *Under certain assumptions, the event of one or more space propagation failures occurs with small probability. The assumptions as well as the calculation of this probability is given in the appendix.*

There are two options in case of space propagation failure: the affected block could be split into two parts by writing the remaining part of the block to a different location; or the affected block could be entirely relocated at the end of the index file, thus providing extra free space for its neighbors. The second alternative requires touching a small number of on-disk postings for the second time. This event, however, occurs with small probability. Furthermore, we limit the number of block relocations to a small constant (e.g., 5 blocks) which amounts to less than 1% of the total number of postings.

7. EXPERIMENTS

We compare an in-place as well as a merge-based version of our HYB construction algorithm against our own implementation of the state-of-the-art inverted index construction from Heinz and Zobel [2003], (including the cache-friendly two-level posting accumulation from Section 5.1 as an additional optimization). As a reference, we also compare against the authors' implementation of the same algorithm that comes as a part of the Zettair[5] search engine. The index construction of Zettair essentially implements the ideas from Heinz and Zobel [2003] but slightly varies from the original single-pass approach by using log-10 partitioning instead of a single n-way merging. According to a large study of Middleton and Baeza-Yates [2007], Zettair's index construction is indeed the fastest on the open source market to date, with performance close to twice as fast as the second fastest option.

7.1. Experimental Setup

Our implementation of the HYB index construction, as described in Sections 4 and 5, writes all blocks to a single file, with an array of block offsets at the end of the file. The vocabulary is compressed with zlib and stored in a separate file on disk. All our code is written in C++ and compiled with GCC 4.1.2 with the -O6 flag. We used the latest (0.9.3) stable version of Zettair.

The experiments were performed on two different machines with different hardware specifications (in particular different disk speeds). The first machine (with standard disk) had 16GB of RAM (with contents flushed before every execution), 4 dual-core AMD Opteron SE @ 2220MHz processors with 1MB cache (we used only one core at a time), operating in 32-bit mode and running Debian 4.1.1-19 with a standard SATA (Hitachi Deskstar) hard disk with 7200RPM, sequential read rate of 60MiB/s, sequential write rate of 75MiB/s and average access time of 12.9ms. The second machine (with fast disk) had 4 quad-core AMD Opteron 2.8GHz processors with 8MB cache, operating in 64-bit mode on a fast RAID file system with sequential read/write rate of up to 600MiB/s.

We investigate the impact of hard disk speed on index merging and in-place block writing from Section 6 separately by carrying out experiments on two additional hard disks: a relatively fast (Seagate Savvio) hard disk with 10000RPM, sequential read/write rate of around 150MB/s and average reported access time of 4.6ms (HDD1); and a slow Samsung hard disk, with 7200RPM, sequential read/write rate of 50MB/s and average access time of 20ms (HDD2).

[5] http://www.seg.rmit.edu.au/zettair/

Table VIII.
Summary of the Three Test Collections Used in Our Experiments

Collection	Raw size	Index size	Documents	Occurrences	Distinct words
DBLP	1.1GB	0.3GB	0.3 millions	0.15 billions	1.2 millions
Wikipedia	21GB	7.2GB	9.3 millions	3.2 billions	29 millions
TREC GOV2	426GB	46GB	25 millions	23 billions	57 millions

For both algorithms, we imposed the same memory limit for the in-memory inversion. We used the `--big-and-fast` option on Zettair which allows 500MB of memory. According to Heinz and Zobel [2003] a higher memory limit does not significantly impact the performance of their construction algorithm. We did not take into account the additional memory usage of the vocabulary and other auxiliary data structures. We picked the HYB block sizes as $N/5$, where N here is the number of documents in the collection (see Bast and Weber [2006]).

To ensure that both parsers produce an equal number of word occurrences, we replaced each nonalphanumeric character in the collections by a single space.

7.2. Test Collections

Our experiments were carried out on three collections (Table VIII):

(1) *DBLP*: a selection of 31,211 computer science articles with a raw size of 1.3GB, 157 million word occurrences (5,030 occurrences per document in average) and 1.3 million distinct words. Our goal was to investigate the impact of the construction algorithm on collections that are small and can be fully inverted in main memory.

(2) *WIKIPEDIA*: a dump of the English Wikipedia, with a raw size of 21GB, 9,326,911 documents, 3.2 billion word occurrences (343 occurrences per document in average), and a vocabulary of 29 million distinct words.

(3) *TREC GOV2*: the TREC GOV2 (or TREC Terabyte) corpus with a raw size of 426GB, 25,204,103 documents, 23 billion word occurrences (912 occurrences per document in average), and a vocabulary of 57 million distinct words. To get a better picture on the scalability of the algorithms we ran the experiments on subsets of size 25%, 50% and 100% of the full collection.

7.3. Index Construction Time

Table IX shows the index construction running times on two of our test collections and two different machines. Compared to our own INV construction, the HYB construction is faster by a factor of 1.7 (on Wikipedia) to a factor of 2 (on GOV2). Compared to Zettair, we are faster by a factor of 1.7–2 (on GOV2) to more than a factor of 2 (on Wikipedia). The merge-based version of our HYB construction algorithm is slower than the in-place version by a factor of 1.2 (on Wikipedia) to a factor of 1.3 (on GOV2). The difference on the DBLP collection was small and it is not shown. We point out that the advantage on a particular system depends on the ratio of disk and CPU speed. As reflected in Table IX, the advantage on systems with high disk bandwidth is less dramatic since our improvement over inverted index construction is slightly larger with respect to disk time than with respect to CPU time.

7.3.1. Posting Accumulation. Table X shows a breakdown of the HYB construction time by 5 major subroutines. Only about 16% of the time it takes to construct our index is spent on posting accumulation or in-memory inversion. As suggested in Section 5.1, for the inverted index construction this cost is more than twice as much. This suggests that

Table IX.
Elapsed index construction time given in minutes to construct a word-level index with our HYB builder and Zettair on Wikipedia and the TREC GOV2 collection

Machine I (standard disk)				
	Wikipedia	25% GOV2	50% of GOV2	100% of GOV2
HYB (in-place)	23 min	43 min	75 min	156 min
HYB (merge)	27 min	54 min	101 min	209 min
INV	52 min	103 min	172 min	327 min
Zettair	50 min	92.3 min	146 min	288 min
Machine II (fast disk)				
	Wikipedia	25% GOV2	50% of GOV2	100% of GOV2
HYB (in-place)	11 min	21 min	36 min	70 min
HYB (merge)	13 min	27 min	45 min	90 min
INV	19 min	34 min	57 min	146 min
Zettair	23 min	35 min	70 min	127 min

Table X.
Breakdown of the total elapsed index construction time by 5 subroutines on the TREC GOV2 collection

Parsing & Hashing	Accumulation	Compression	Disk I/O	Sampling
32%	16%	20%	29%	3%

the cache inefficiency seriously affects the running time of the in-memory inversion of the single-pass approach. For whatever reason, posting accumulation cost is not reported in the elapsed-time figures of Heinz and Zobel [2003].

7.3.2. In-Place vs Merge-Based Index Construction. Index merging for the HYB index is simple to implement and requires no priority queue to find the next chunk of data to be written to disk. It works by allocating a buffer to each on-disk run and writing the partial HYB blocks in-order by iterating the buffers one by one. Whenever empty, each buffer is refilled with the next few partial blocks from the corresponding run.

Figure 6 compares the disk costs of merge-based versus in-place index construction on two different hard disks. In-place block writing almost always outperforms index merging with the difference between the two being larger on the faster hard disk where a disk seek is less expensive. Index merging is faster on the slower disk only when the available main memory is very low.

While merge-based (INV and HYB) index construction spends about 20%–30% of the total time on writing and merging the index, only about 5% of the time on the faster hard disk and about 15% of the time on the slower hard disk is spent on in-place block writing. The reason for this is that merge-based construction has to fully read and write the temporary index file more than once. We note that the actual bottleneck of index merging is not caused by the disk seeks (in our experiments the disk seeks took about 1% of the HYB merging time and below 15% of the INV merging time) but is rather the result of reading and writing large amounts of data from disk. As a consequence, the index partitioning improvement proposed in Heinz and Zobel [2003], which aims to reduce the number of disk seeks by performing an in-memory index merge, has been only marginally faster in practice.

In Section 6 we saw that the disk seek cost of our construction algorithm depends on the number of blocks as well as on the total number of runs which in turn depends on the main memory limit. Table XI shows the increase in running time of the HYB index construction for varying number of HYB blocks when two different memory limits are used. An increase by a factor of 10 in the number of blocks increased the total running

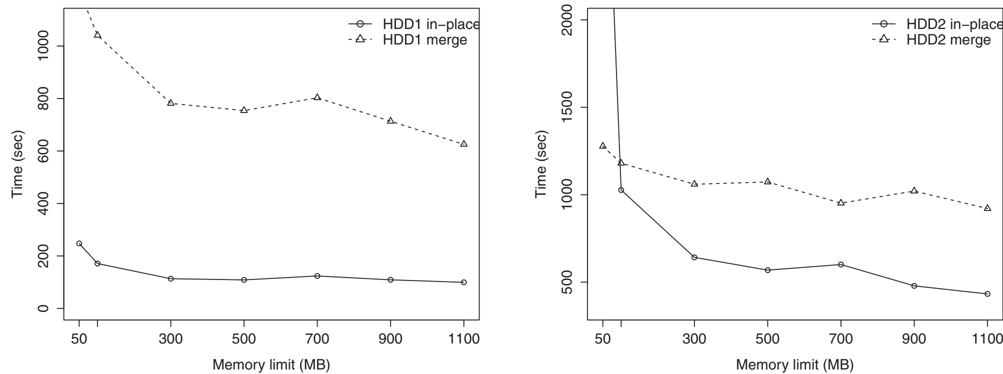

Fig. 6. Disk cost of in-place and merge-based HYB index construction on the GOV2 collection, when a fast (left) and a slow hard disk is used (right) (see Section 7.1).

Table XI.
Increase in HYB construction time given in percents for varying number of HYB blocks and two different memory limits. The experiment was carried out on the GOV2 collection

Mem. Limit	2000 blocks	4000 blocks	6000 blocks	8000 blocks	10000 blocks
512MB	0%	+2.9%	+9.2%	+13.3%	+21.5%
1024MB	0%	+1.6%	+5.8%	+6.5%	+8.2%

time by about 8% when the memory limit was set to 1024MB and by about 21% when the memory limit was set to 512MB.

7.3.3. Parsing and Hashing. The most expensive phase is the parsing and hashing of words which takes roughly one third of the total cost. We believe that there is not much room for improvement here since a hash table with move-to-front chains and a bitwise hash function has been the fastest practical data structure for in-memory vocabulary accumulation for some time, conditioned on the assumption that the words do not need to be maintained in sort order [Zobel et al. 2001]. An alternative fast and practical data structure that maintains the sorted order of the words is the *burst trie* [Heinz and Zobel 2002].

7.4. Temporary Disk Space

The additional temporary disk space usage during the HYB index construction (other than that required for the final index) is very small and comes from the overflowed blocks. The total size of the overflowed space depends on the quality of the initial estimation of the block sizes (see Section 6.3). The additional peak disk space usage in our experiments varied from 1% of the size of the final index file on small to medium-size collections, up to 3% on the full TREC GOV2 (up to 3% overhead).

Our inverted index construction writes the index in a separate file by merging the temporary index file. The total additional space usage is hence 100% or twice the index size. The reported additional temporary disk space usage in Heinz and Zobel [2003] is about 26% for document-level inverted indexes and about 8% for word-level inverted indexes. However, we found out that during the parsing and merging phase the peak additional space usage of Zettair on the TREC GOV2 was about 65% and 43% respectively of the size of the final index file. We note that in a setting where the input data is very large and streamed, using significantly more space than the final index

Table XII.
Peak Temporary Disk Space Usage Compared to the Size of the Final Index

	Zettair	HYB construction	INV construction
Wikipedia	163% (12GB)	101% (8GB)	200% (16GB)
TREC GOV2	165% (82GB)	103% (46GB)	200% (108GB)

Table XIII.
Increase in the HYB construction time on one quarter of the TREC GOV2 collection when different in-memory vocabulary size limits are used (see Section 7.5). The total number of distinct words was roughly 28.5 millions and the construction (without vocabulary limits) took 43 mins

Vocabulary size limit in %	50%	33%	25%	20%
Large Hash-keys	+7%	+10%	+12%	+11%
On-disk vocabulary	+14%	+20%	+22%	+23%

might be undesirable. Table XII gives a summary of the peak disk space usage (note that the total and not the additional disk usage is shown).

7.5. Refinements

Our basic algorithm assumes that the entire vocabulary fits into main memory. Given that the vocabulary of TREC GOV2 takes roughly 600MB, this is a realistic assumption. Still, for the case where one wants to get rid of the assumption, we propose the following refinements.

7.5.1. Large Hash Keys. Instead of permanently storing word–word-id pairs, we can compute the word-ids with an additional hash function, making it possible to flush the vocabulary to disk once its size approaches the memory limit. At the end of the inversion the sorted vocabulary fragments[6] are merged into a final vocabulary (without fully reading them in memory). To avoid word-id collisions one can easily show that by providing a universal family of hash-functions with large enough hash keys, the expected number of collisions can be kept below 1.

Let v be the vocabulary size and r be the size of the range of the hash function. Let X_i be the indicator random variable that is 1 iff the i-th word collides with an earlier word, and 0 otherwise. Since $Pr(X_i = 1) = (i-1)/r$, the expected number of such collisions is equal to

$$\sum_{i=1}^{v} Pr(X_i = 1) = \sum_{i=1}^{v} \frac{i-1}{r} = \frac{v(v-1)}{2 \cdot r}.$$

Hence, $v(v-1)/2r < 1$ as long as $v < (1 + \sqrt{8 \cdot r})/2$ which is roughly equal to 6.074 billions for $r = 2^{64}$ (the TREC GOV2 collection has roughly 50 million distinct terms).

The running time overhead imposed by this procedure amounts to only about 10% of the total index construction time in practice (see Table XIII). The size of the compressed word-ids does not significantly increase since the word-id compression algorithm works by compressing the word ranks instead of directly compressing the word-ids. The unique word-ids are only stored in the codebook (rank to word-id mapping), which due to the Zipfian distribution of the word-ids has small size (see Section 5.2).

7.5.2. On-Disk Vocabulary. If one wants to avoid 64-bit word-ids, an alternative approach to limit the size of the vocabulary is to keep a part of the vocabulary on disk in the following way. Once the limit of the vocabulary size is reached, a small number

[6]The vocabulary is sorted before flushing it to disk.

of (word, word-id, hash-value) triples that correspond to rare words is flushed and appended to a temporary file on disk so that the vocabulary size is slightly below the limit. Whenever a new run is to be compressed and written out to disk, each word is read from the file and checked against the hash-table. If the word is in the hash-table this means that an already flushed word has occurred again, inevitably having a different word-id. To prevent multiple word-ids for a single word, the new word-id (that is already in the vocabulary) is replaced with the initial. To address the problem of incorrect word-id assignments in the current in-memory run, instead of word-ids we store pointers to the word-ids that are stored in the vocabulary, without using compression.

The overhead in the index construction time here is about twice as large as that in the previous approach and it comes from frequently reading from disk a gradually growing fraction of the vocabulary.

7.5.3. Multiple Cores. Another way to address the vocabulary size problem is to extend our algorithm to work in a multiprocessor environment as follows. Let C be the number of processor cores. The first easy-to-implement approach simply assigns a group of k/C HYB blocks to each core, where one of the cores also does the parsing. Hence the hashing, compression and posting accumulation part of the inversion (which in total accounts for more than 1/2 of the total time) will benefit from a speed-up by a factor of up to C. Because each of the cores is responsible for only a certain range of words their vocabulary sizes remain reasonably small to fit in main memory.

A second approach that is orthogonal to the first and exhibits more parallelism, makes use of the observation that most of the words in large vocabularies are rare. If the TREC GOV2 collection is split into 16 equal-sized segments, the total number of words in each of the 16 vocabularies will sum up to only twice the size of the vocabulary of the original collection. Instead of a group of HYB blocks, each core is now responsible for a fraction of all documents, without a central core. To prevent multiple word-ids, the cores should use a common hash-function to generate the word-ids (as in Section 7.5.1).

If a single index is desired, then each core must write its run in the same index file. Since this can happen at any given point in time, either the run writing should be synchronized (the first core is always the first to write, the second core always the second, etc.) or additional header data for the order of runs should be included in the blocks. The time needed for the whole in-memory part of the inversion, which in total accounts for more than 2/3 of the total index time, will be reduced by a factor of up to C.

In case of multiple indexes, each core has its own index and one of the cores in addition is in charge for answer-set merging across the partial query results.

8. INDEX MAINTENANCE

So far it has been shown that our HYB index construction is significantly faster than the state-of-the-art inverted index construction on static collections. In many search applications, however, one has to deal with dynamically growing collections such as news search, search in digital libraries and literature search (e.g., DBLP), where new documents arrive at a high rate and the search engine must frequently update its index. In such a setting efficient index maintenance is essential to ensure that the new content is searchable for the user at all times.

In the following we restrict ourselves to document insertions. The goal of all update strategies is to store the incoming and the existing posting lists in a contiguous fashion. Indexes that are stored on disk with discontiguities require additional disk seeks at query time which can easily become a bottleneck in the query processing routines.

8.1. Index Maintenance for INV

In a nutshell, all index update strategies work by amortizing the update cost over a series of updates by maintaining a temporary in-memory index for the new documents.

When the main memory is exhausted, the in-memory posting lists are combined with the existing index according to the update strategy and memory is freed [Zobel and Moffat 2006]. Two main strategies for index maintenance were proposed and experimentally compared in Lester et al. [2004]: *merge-based* and *in-place* strategies. Hybrid update strategies combine the advantages of the two strategies and are the fastest in practice [Büttcher and Clarke 2008].

8.1.1. Remerge. The merge-based strategy (also remerge) works by employing a sequential merge of the new and the existing on-disk posting lists as follows. Whenever the in-memory index runs out of buffer space, all lists are processed in ascending order using the hash value of the corresponding index word as the sorting key. For each in-memory and on-disk posting list, if the index word of the in-memory posting list has a hash value greater than that of the on-disk list, the in-memory posting list is written to the new index (and viceversa). If the hash values are equal, then the in-memory posting list is written after the on-disk posting list. Hence, a full copy of the existing indexes is made. The remerge strategy is easy to implement, but it suffers from the limitation that on every update the entire on-disk index has to be processed.

Assume that the number of postings in the update is equal to a constant fraction c of the full collection. Let N be the total number of postings and M be the number of postings that can fit in memory (in compressed format). Then the cost of remerge in number of read/written postings is equal to

$$\frac{cN}{M} \cdot (M + 2 \cdot N) = \Theta\left(\frac{N^2}{M}\right), \tag{7}$$

where $0 < c \leq 1$, since each time M posting should be read from disk and the entire index should be read, remerged, and written. Obviously the cost is quadratic in the number of postings.

8.1.2. In-Place. The in-place strategy does not suffer from the shortcomings of remerge. The idea is to allocate some free space whenever a posting list is transfered to disk. The new in-memory posting list is then appended at the end of the corresponding on-disk list on the fly, without the need to read the entire list from disk. In case of insufficient disk space, the entire posting list is moved to a location with enough space, again with free space allocated at the end of the list. This strategy in practice has worse performance than remerge due to the following limitations. First, since a document can contain a large number of distinct words a large number of random disk seeks is required for each update. This is very wasteful for short lists where a disk seek exceeds the time needed to append the new list. Second, there is no obvious disk space allocation strategy when relocating the posting list so that index fragmentation is minimized.

If the free space is allocated by doubling the current list size at each relocation, then the cost in number of read/written postings for a list of size l is at most

$$2 \cdot \sum_{i=0}^{\infty} \frac{l}{2^i} = 4 \cdot l, \tag{8}$$

that is, the update complexity is linear in the number of postings. However, we should also include the disk seek cost which is equal to $\log_2 l$ disk seeks per posting list. Since typically millions of postings lists have to be updated, the disk seek time becomes the dominating part of the total update time.

8.1.3. Hybrid Index Update Operations. The hybrid update strategies try to combine the advantages of the two basic merging strategies. One of the observations in Büttcher and Clarke [2008] is that the total disk overhead associated with index update operations

Table XIV.
Statistics for the growth of the HYB block sizes on Wikipedia with documents sorted w.r.t their entry dates. The first half of the collection has been indexed and the other half incrementally added to the index. 2x below means that for each block it has been allocated twice more than the estimated space

Percentage of added docs.	10%	20%	30%	40%	50%
Average block increase	10%	21%	31%	41%	51%
Standard deviation	3%	6%	10%	13%	17%
Blocks requiring more space (2x)	0	0	0	3	9

is not optimized by exclusively applying any of the two strategies alone. For example, in-place is more appropriate for long lists since the overhead of random disk accesses is not worthwhile for lists that are short. Following this idea, given a certain length threshold, a list that becomes longer than the threshold is marked as long, and all other as short. The long list are then updated via in-place and the short lists via the remerge strategy. Büttcher and Clarke [2008] show how to compute an optimal threshold value, achieving an improvement of about 74% compared to remerge. The reported disk complexity in number of transfered postings is between $\Theta(N^{1.6})$ and $\Theta(N^{1.7})$.

8.1.4. Geometric Partitioning. Lester et al. [2008] propose another technique to improve the performance of remerge, called *geometric partitioning*. The insight of the technique goes as follows. Since the cost of merging two indexes is proportional to the sum of their sizes, instead of one large index, a hierarchy of indexes is maintained, with each index in the hierarchy having capacity that is r times larger than the prev-ious index. The in-memory index is always merged with the smallest index in the hierarchy and whenever an index runs out of capacity it is merged into the index at the next level. The disadvantage of this technique is that each of the indexes has to be queried independently and the partial results merged. Another disadvantage is more complex index construction and maintenance.

8.2. Index Maintenance for HYB

Index update for the HYB index is based on the observation that it is plausible that the blocks should grow at reasonably similar rates up to a certain point in the future. One weaknesses of the in-place strategy (as well as the motivation behind hybrid index maintenance) is its wastefulness on short lists. Observe, however, that this weakness vanishes when in-place is applied to the HYB index where all blocks are of a roughly similar size (or larger when they correspond to word ranges of frequent words). For the second limitation of in-place (no obvious preallocation strategy), recall again that each block size is already estimated and known in advance. If the blocks retain similar sizes in the future, then a sufficient allocation strategy would be to double the block sizes. Table XIV provides experimental evidence to support this claim. In this experiment we sorted all documents from the Wikipedia collection by their entry date, constructed a HYB index on the first half of the collection, and incrementally added documents from the second part of the collection. Our update procedure in greater detail goes as follows.

—We allocate, say twice more than the estimated space for each block. Once the memory for the in-memory index is exhausted, all the affected on-disk blocks are updated by appending their in-memory counterparts directly at the end of each block, similarly as in Lester et al. [2004];
—We maintain a separate table with information about the location of each block;
—Whenever a block runs out of free space it is split into two parts, assigning the second part of the block to one of the blocks with more than enough space (for example, the 20 smallest blocks in the experiment from Table XIV required less than 10% of the

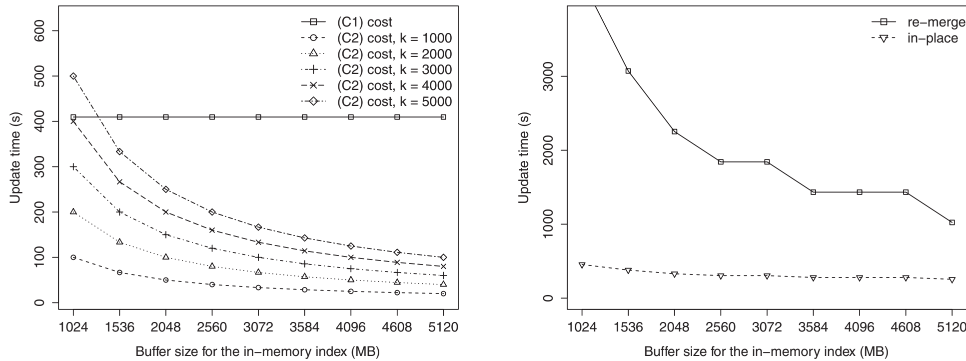

Fig. 7. Left: The (C1) and (C2) index update costs on the HYB index for different sizes of the in-memory index and different number of HYB blocks for a disk throughput of 50MB/s and disk seek time of 5ms (see Table VII). The assumed collection size is 20GB (in compressed format) and the update size is 10GB. Right: Update running times (given in seconds) of the two update strategies for different buffer sizes of the in-memory index. The update size is 10GB and the index size is 20GB with $k = 5000$ HYB blocks.

allocated space). Alternatively, if block splitting is undesired, the affected block is relocated to a location with enough free space (possibly at the end of the index);

—If the number of blocks requiring more space reaches a threshold (e.g., 10 blocks), then the entire index is rebuilt and the block boundaries are recomputed by collecting as many postings for each block as required so that the blocks contain roughly the same number of postings (note that the total number of occurrences as well as the total number of blocks is known in advance).

Unlike for the inverted index, for an update that is a fraction of the whole collection, rebuilding the HYB index is not required. The frequency of the index rebuilding will depend, however, on the nature of the collection. The total cost of the in-place update strategy on the HYB index hence consists of the following components:

(C1) Appending new data from the temporary in-memory index to each block;
(C2) Seeking to the last position of every block;
(C3) Rebuilding the index (if a critical number of blocks run out of space).

The cost of (C1) is equal to $\frac{cN}{M} \cdot M = cN$ and it is the cost absolutely necessary to spend. The overhead cost (in number of disk seeks) that comes from (C2) is equal to $\frac{cN}{M} \cdot k$ where k is the number of blocks. k is typically thousand times smaller than the number of posting lists and unlike for the inverted index, this cost is usually not dominating. Figure 7 (left) compares the costs of (C1) and (C2) for different in-memory buffer sizes and different number of blocks. The cost of (C3) is equal to the cost to copy the entire index to a new location. By using space allocation by doubling and index rebuilding whenever the collection approaches 1.5 times of its original size, the number of times required to relocate the index for a cumulative update of total size $c \cdot N$ is equal to

$$\lfloor \log_{1.5}(c+1) \rfloor \approx \lfloor 1.7 \cdot \log_2(c+1) \rfloor,$$

compared to $\lfloor c \cdot N/M \rfloor$ rebuilds when using the remerge strategy. Nevertheless, our experience says that text collections usually do not grow very fast relatively to their size. For example, compared to last year the (English) Wikipedia is only twice as large. This would require one or two index rebuilds over that whole period.

Figure 7 (right) shows the running times of the remerge and the in-place update strategy when applied on the HYB index for different buffer sizes and update size

comparable to that of the full index. In practice the performance gap between the strategies is large due to the obvious drawback of the remerge strategy that even for small updates a full copy of the entire index is required.

Index copying and block collisions during construction and update could be avoided by maintaining a separate file for each HYB block[7] with the caveat that the files may not occupy contiguous space at all. The price paid would be increased index construction and query time.

9. CONCLUSIONS

We have shown that the HYB index can be constructed with both less I/O and less CPU time than an INV index. Our HYB construction is always faster, and for a typical ratio of CPU to disk transfer speed, it is twice as fast.

The main reasons for this speedup are that (i) HYB requires only a half-inversion of the data, while INV requires a full inversion, and that (ii) the HYB blocks can be constructed in-place, without the need for merging. Exploiting this potential is not trivial, however. One challenge is to compute good block boundaries without making an additional pass over the data. Another challenge is to accumulate postings in a cache-efficient manner; we have proposed a procedure called multilevel posting accumulation that can be used to improve the cache-efficiency of both, INV and HYB. Yet another challenge is to do the word-id compression (needed only for HYB and not for INV) efficiently. We have also shown that dynamic index updates are much easier to achieve efficiently for HYB than for INV.

It is worth noting that despite the fact that our construction algorithm practically requires only a single pass, it might be useful to regard it as a heavily optimized two-pass approach, as well, since the techniques used in this work could be useful as optimizations to other two-pass approaches.

Given the already known powerful querying capabilities supported by HYB, this makes HYB a very attractive alternative to INV in many practical search engine scenarios.

APPENDIX

A. PROOFS OMITTED FOR BREVITY

PROOF OF LEMMA 4.1. Call the $s \cdot k$ random numbers picked in the beginning splitters. Let the maximum block size be B_{max}. We consider the event that B_{max} is larger than some B. Then there must be a subrange of size B that contains strictly less that s splitters. There are $n - B + 1 \leq n$ such subranges in total which means that $Pr(B_{max} > B) \leq n \cdot p$, where p is the probability that a fixed subrange of size B contains less than s splitters. This probability is equal to

$$\sum_{i=0}^{s-1} \binom{sk}{i} (B/n)^i \,(1 - B/n)^{sk-i} .$$

We will derive an upper bound on the probability $p(s)$ that exactly s splitters fall into a fixed range of size b and from there derive Equation (1). After plugging $B = a \cdot n/k$ into $p(s)$ and applying the inequalities $\binom{sk}{s} \leq (ek)^s$ and $1 - x < \exp(-x)$; by simple transformations we obtain

$$p(s) \leq \exp\left(\left(1 + \ln(a) - a \cdot \frac{k-1}{k}\right) \cdot s\right), \tag{9}$$

[7]This is feasible since the number of HYB blocks is in the order of thousands.

which can be written as $\exp(-C \cdot s)$, for $C > 0$. This inequality is satisfied for all practical values of k (e.g., $k > 1000$), provided that $a > 1$. To complete the proof we will use the inequality $p \leq s \cdot p(s)$ provided that $p(s) \geq p(s-1) \geq \cdots \geq p(0)$. For the binomial distribution the latter holds if s is no larger than the mode of the distribution M as $p(s)$ is maximized when $s = M$. In our case this condition is satisfied as

$$M = \lfloor sk \cdot B/n \rfloor \geq sk \cdot a/k = s \cdot a,$$

which is larger than s if $a > 1$. By plugging in Equation (9) in the latter inequality we obtain $p \leq \exp\left(\left(1 + \ln(s)/s + \ln(a) - a \cdot (k-1)/k\right) \cdot s\right)$ which can be also written as $\exp(-K \cdot s)$. Again, $K > 0$ for small values of $a > 1$ and all practical values of k. This concludes the proof. □

Proof of Lemma 5.1. Under the assumptions given in Section 5.1, the expected numbers of cache misses for HYB posting accumulation is equal to

$$l \cdot n \cdot \left(1 - \frac{c}{\sqrt[l]{k}}\right), \tag{10}$$

assuming $c \leq \sqrt[l]{k}$. Obviously, by picking $l = \lceil \log k / \log c \rceil$ we could reduce the number of cache misses to 0, however ideally one should minimize the total cost by computing

$$\operatorname*{argmin}_l \, l \cdot \left(\left(1 - \frac{c}{\sqrt[l]{k}}\right) \cdot T_m + \frac{c}{\sqrt[l]{k}} \cdot T_h\right), \tag{11}$$

where T_m and T_h respectively are the costs for a cache miss and a cache hit.

In the following we compare the total posting accumulation cost for $l = 1$ and $l = 2$. The expected numbers of cache misses for $l = 1$ and $l = 2$ are $n \cdot (1 - c/k)$ and $2n \cdot (1 - c/\sqrt{k})$, respectively. Given that the ratio between the two is equal to

$$\frac{k - c}{2\sqrt{k}(\sqrt{k} - c)}, \tag{12}$$

the following three cases could take place when $l = 2$: first, the number of cache misses will be less for $k < 4 \cdot c^2$; second, the number of cache misses will be less by a large factor for $\sqrt{k} \sim c$; and third, the number of cache misses will be equal to zero for $\sqrt{k} \leq c$. The last scenario is realistic given that the number of blocks is typically less than 10,000 and that todays L1-caches are larger than 8KB. Namely, 8KB cache with 64B cache lines, has $c = 128$ cache lines in total, where $\sqrt{k}$ is usually less than 100. By assuming that $\sqrt{k} \leq c$, the total cost for $l = 2$ is simply equal to $2 \cdot n \cdot T_h$, while the ratio between the two costs is equal to

$$\frac{n\left(\left(1 - \frac{c}{k}\right) T_m + \frac{c}{k} \cdot T_h\right)}{2n \cdot T_h} = \frac{1}{2} \cdot \frac{T_m}{T_h} - \frac{c}{k} \cdot \frac{T_m - T_h}{2T_h}.$$

Say a cache hit is m times faster than a cache miss i.e. $T_m/T_h = m \geq 2$. Two-level posting accumulation is then faster by a factor of

$$\frac{m}{2} - g(k), \tag{13}$$

where $g(k) = \frac{c}{k} \cdot \frac{m-1}{2}$ is small and reaches its minimum for $c = \sqrt{k}$, resulting in a factor of $\frac{m}{2} - \frac{m-1}{2\sqrt{k}} \approx \frac{m}{2}$ speed-up. Indeed, the best result in practice was achieved for $l = 2$. □

PROOF OF LEMMA 6.2. Observe that by using partial flushing the total number of runs will increase since the memory buffer is not always fully emptied, however the average number of disk seeks per run will decrease since only the large blocks are flushed. The memory threshold T of the small blocks on average is reached once in every

$$\lfloor T/f \rfloor + 1$$

runs, at which point $L_1 + L_2$ disk seeks are required. Every other run requires only L_1 disk seeks. The average number of disk seeks per run from $L_1 + L_2$ hence reduces to

$$\left(1 - \frac{1}{\lfloor T/f \rfloor + 1}\right) \cdot L_1 + \frac{1}{\lfloor T/f \rfloor + 1} \cdot (L_1 + L_2) = L_1 + \frac{L_2}{1 + \lfloor T/f \rfloor}.$$

The total number of runs required will be increased by a factor of

$$\frac{\lfloor T/f \rfloor + 1}{\sum_{i=0}^{\lfloor T/f \rfloor} (1 - i \cdot f)} = \frac{\lfloor T/f \rfloor + 1}{(\lfloor T/f \rfloor + 1)\left(1 - f \frac{\lfloor T/f \rfloor}{2}\right)} = \frac{2}{2 - f \cdot (\lfloor T/f \rfloor)}.$$

The total number of disk seeks is the product of the average number of seeks per run and the total number of runs. Compared to the baseline, partial flushing reduces the required number of disk seeks at least by a factor of

$$\frac{2 - T}{2} \cdot \frac{1 + \lfloor T/f \rfloor}{1 + \frac{L_1}{k} \lfloor T/f \rfloor} \geq \frac{1}{2} \cdot \frac{fk + k}{fk + L_1} = \frac{1}{2} \cdot \frac{m + fm}{1 + fm},$$

where $m = k/L_1$. □

B. EVIDENCE FOR OBSERVATION 6.4

To provide some theoretical evidence that the block space propagation procedure from Section 6.3 fails with small probability, we consider the following model. Assume that the correct block size is B and that the estimated block size is $\hat{B}$. We define the estimation error of that block as

$$\varepsilon = \frac{\hat{B} - B}{B}$$

or the fraction for which the estimated block size varies from the true block size. Assume that the estimated size of each block varies around the true size with Gaussian error with mean 0, that is, $\varepsilon \sim N(\mu, \sigma^2)$ with $\mu = 0$. To provide evidence that this model is realistic, Figure 8 and Table XV show the empirical distribution of ε on two of our test collections. Furthermore, assume that $1 - r$ is a fraction of the full collection that is large enough to provide a reliable estimation for the true block sizes (e.g., 0.8 or 80%). Since the space propagation goes in discrete steps, we assume that there are enough of them for the propagation to "converge." We will compute an upper bound on the probability that at least one space propagation failures takes place.

Consider the event that a single block causes a space propagation failure on its own. This event will occur if the block size is underestimated to the degree that the

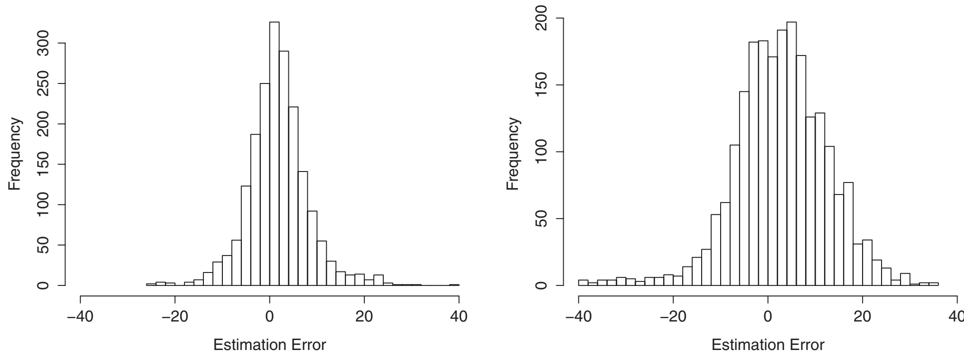

Fig. 8. Empirical distribution of the block size estimation error ε for DBLP (left) and Wikipedia (right) when 5% of all documents are sampled.

Table XV.
Percentage of block estimation errors within one, two and three standard deviations. According to the called 3-sigma rule for normal distribution, around 68% of the data should lie within one, about 95% within two and about 99.7% within three standard deviations

	$\leq \sigma$	$\leq 2\sigma$	$\leq 3\sigma$
DBLP	74.5%	94.7%	98.1%
Wikipedia	73.9%	95.1%	99.2%

maximum borrowable space (from both of its neighbors) is insufficient to fit its data:

$$B \cdot \varepsilon + 2 \cdot r < 0,$$

where ε is the estimation error of that block. Note that by adding another block, the probability of the latter isolated event does not change. However, even if both blocks do not cause a propagation failure on their own, they can still cause a joint propagation failure when their space demand is combined. This event will occur if

$$B \cdot \varepsilon_1 + B \cdot \varepsilon_2 + 2 \cdot r < 0,$$

where ε_1 and ε_2 are the estimation errors of the first and the second block, respectively (note that the blocks can borrow space from each other). Let $F_{i,j}$ for $i \leq j$ be the event that the group of blocks that starts at position i and ends at position j, causes a (combined) propagation failure. The probability of at least one propagation failure (or just a failure probability) is then equal to

$$Pr(\cup_{1 \leq i \leq j \leq k} F_{i,j}).$$

Let $p_{j-i} = Pr(F_{i,j})$. Since $Pr(F_{i_1,j_1}) = Pr(F_{i_2,j_2})$ for $j_2 - i_2 = j_1 - i_1$, by the union bound we obtain

$$Pr\left(\cup_{1 \leq i \leq j \leq k} F_{i,j}\right) \leq k \cdot p_1 + (k-1) \cdot p_2 + \cdots + 1 \cdot p_k. \quad (14)$$

The question now is whether the joint failure probability for a group of m blocks (p_m) increases when m ($\leq k$) grows. The joint propagation failure probability for a group of m blocks is equal to

$$p_m = Pr\left(\sum_{i=1}^{m} \varepsilon_i + 2 \cdot r < 0.\right) \quad (15)$$

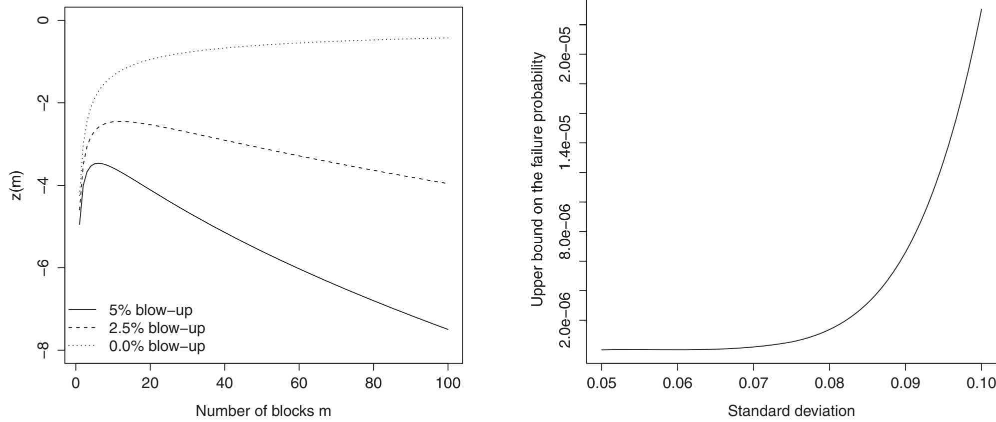

Fig. 9. Left: value of the $z(m)$ argument of the space propagation failure probability with $r = 70\%$ and 3 block space blow-up factors. (note that already $p(-4) \sim 7.7 \cdot 10^{-9}$). Right: Upper bound on the space propagation failure probability when the standard deviation (σ) grows from 5% to 10% (the space blow-up is set to be equal to the standard deviation).

According to the assumptions, $\sum_{i=1}^{m} \varepsilon_i$ has Gaussian distribution with mean $m \cdot \mu$ and variance $m \cdot \sigma^2$, which means that p_m can be written as

$$p_m(z) = \frac{1}{2\left(1 + \operatorname{erf}\left(z\left(m\right)\right)\right)},$$

where erf() is the Gaussian error function and

$$z(m) = \frac{-2r - m\mu}{\sigma\sqrt{2m}}. \tag{16}$$

Note that $p_m(z)$ is a strictly increasing function of $z(m)$. Obviously if $\mu = 0$, then $z(m)$ strictly increases with m, resulting in large values of $p_m(z)$ (e.g., 0.5 for $m = 2000$). However, if $\mu > 0$ (a small blow-up in the block sizes), then $z(m)$ reaches a maximum for $m = 2 \cdot r/\mu$ and then starts to decrease, resulting in extremely small values of $p_m(z)$ for large m (e.g., $m = 2000$). Figure 9 (left) plots the $z(m)$ value against the number of blocks m with three different space blow-up factors.

For reasonable values of the standard deviation of the block size estimation error σ, the upper bound on the failure probability given in Equation (14) remains small, for instances, for any given number of blocks, assuming $r = 30\%$, the failure probability ranges from $2.5 \cdot 10^{-9}$ to $2.3 \cdot 10^{-5}$ when $5\% \leq \sigma \leq 10\%$ as Figure 9 (right) shows.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their outstandingly painstaking, competent, and constructive comments.

REFERENCES

Arge, L., Samoladas, V., and Vitter, J. S. 1999. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the 18th Symposium on Principles of Database Systems (PODS'99)*. ACM, New York, NY, 346–357.

Bast, H. and Celikik, M. 2010. Efficient two-sided error-tolerant search. In *Proceedings of the 2nd International Workshop on Keyword Search on Structured Data (KEYS'10)*. ACM, 3.

Bast, H., Chitea, A., Suchanek, F., and Weber, I. 2007. Ester: Efficient search on text, entities, and relations. In *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR'07)*. ACM, New York, NY, 671–678.

Bast, H. and Weber, I. 2006. Type less, find more: fast autocompletion search with a succinct index. In *Proceedings of the 29th Annual International Conference on Research and Development in Information Retrieval (SIGIR'06)*. ACM, New York, NY, 364–371.

Bast, H. and Weber, I. 2007. The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR'07)*. G. Weikum, Ed. VLDB Endowment, 88–95.

Breslau, L., Cue, P., Cao, P., Fan, L., Phillips, G., and Shenker, S. 1999. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99)*. IEEE Computer Society Press, Los Alamitons, CA, 126–134.

Büttcher, S. and Clarke, C. L. 2008. Hybrid index maintenance for contiguous inverted lists. *Inform. Retrie. 11,* 3, 175–207.

Buttcher, S. and Clarke, C. L. A. 2005. Memory management strategies for single-pass index construction in text retrieval systems. Tech. rep., School of Computer Science, University of Waterloo, Canada.

Celikik, M. and Bast, H. 2009a. Fast error-tolerant search on very large texts. In *Proceedings of the Symposium of Applied Computing (SAC'09)*. ACM, New York, NY, 1724–1731.

Celikik, M. and Bast, H. 2009b. Fast single-pass construction of a half-inverted index. In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE'09)*. Springer-Verlag, 194–205.

Cochran, W. G. 1977. *Sampling Techniques,* 3rd Ed. John Wiley & Sons, Inc., New York, NY.

Crauser, A. and Ferragina, P. 2002. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica 32,* 1, 1–35.

Faloutsos, C. and Christodoulakis, S. 1984. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Inform. Syst. 2,* 4, 267–288.

Farach, M. 1997. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS'97)*. IEEE Computer Society, Los Alamitos, CA, 137.

Ferragina, P., Koudas, N., Muthukrishnan, S., and Srivastava, D. 2003. Two-dimensional substring indexing. *J. Comput. Syst. Sci. 66,* 4, 763–774.

Ferragina, P. and Manzini, G. 2005. Indexing compressed text. *J. ACM 52,* 4, 552–581.

Grama, A., Karypis, G., Kumar, V., and Gupta, A. 2003. *Introduction to Parallel Computing, 2nd Ed*. Addison Wesley, Boston, MA.

Harman, D. and Candela, G. 1990. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *J. Amer. Soc. Inform. Sci. 41*, 581–589.

Heinz, S. and Zobel, J. 2002. Performance of data structures for small sets of strings. *Austra. Comput. Sci. Comm. 24*, 87–94.

Heinz, S. and Zobel, J. 2003. Efficient single-pass index construction for text databases. *J. Am. Soc. Inform. Sci. Tech. 54*, 713–729.

Lester, N., Moffat, A., and Zobel, J. 2008. Efficient online index construction for text databases. *ACM Trans. Data. Syst. 33,* 3, 1–33.

Lester, N., Zobel, J., and Williams, H. E. 2004. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian Conference on Computer Science (ACSC'04)*. Vol. 56. Australian Computer Society, Inc., 15–23.

Manber, U. and Myers, G. 1990. Suffix arrays: a new method for on-line string searches. In *Proceedings of the 1st Annual Symposium on Discrete Algorithms (SODA'90)*. SIAM, Philadelphia, PA, 319–327.

Middleton, C. and Baeza-Yates, R. 2007. A comparison of open source search engines. Tech. rep. http://wrg.upf.edu/WRG/dctos/Middleton-Baeza.pdf.

Moffat, A. and Bell, T. A. H. 1995. In situ generation of compressed inverted files. *J. Am. Soc. Inform. Sci.Tech. 46*, 537–550.

Moffat, A. and Zobel, J. 1996. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inform. Syst. 14*, 349–379.

Popovici, F. I., Arpaci-dusseau, A. C., and Arpaci-dusseau, R. H. 2003. Robust, portable I/O scheduling with the disk mimic. In *Proceedings of the USENIX Annual Technical Conference*. IEEE, Los Alamitos, CA, 297–310.

Rogers, W., Candela, G., and Harman, D. 1995. Space and time improvements for indexing in information retrieval. In *Proceedings of 4th Annual Symposium on Document Analysis and Information Retrieval (SDAIR'95)*.

Ukkonen, E. 1995. On-line construction of suffix trees. *Algorithmica 14,* 3, 249–260.

Witten, I. H., Moffat, A., and Bell, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Diego, CA.

Zobel, J., Heinz, S., and Williams, H. E. 2001. In-memory hash tables for accumulating text vocabularies. *Inform. Process. Lett. 80*, 271–277.

Zobel, J. and Moffat, A. 2006. Inverted files for text search engines. *ACM Comput. Surv. 38*, 6.

Zobel, J., Moffat, A., and Ramamohanarao, K. 1998. Inverted files versus signature files for text indexing. *ACM Trans. Inform. Syst. 23,* 4, 453–490.

Received April 2010; revised November 2010, February 2011; accepted March 2011