Fast Construction and Maintenance of the HYB Index

Hannah Bast · Marjan Celikik

Received: date / Accepted: date

Abstract We show that a HYB index can be constructed twice as fast as an ordinary inverted index. As shown in a series of recent works, the HYB index 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, synonym search, error-tolerant search etc. HYB can be viewed as a “half-inverted index” and one of the main reasons for our speed improvement is that for a half-inversion we can omit the expensive index merging phase required for an ordinary inverted index. The algorithm has been carefully engineered, with special attention paid to cache-efficiency and disk cost. We compared our algorithm to the state-of-the-art index construction from Zettair. We also show that our HYB index construction naturally supports very fast dynamic index updates with running time that is independent of the size of the collection.

1 Introduction

The inverted index 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 (20% - 60%, or more, depending on whether positional information is considered or not (Witten et al, 1999; Zobel and Moffat, 2006)), it can be constructed fast, and it enables very fast full-text search. Its one major shortcoming is that only basic keyword queries (find all documents that contain all or some of the given query words) are supported efficiently.
In Bast and Weber (2006), an alternative index data structure, called HYB, was proposed that was shown to be as compressible as the inverted index (INV), but provides efficient support for a certain kind of prefix search (see Sect. 1.2 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: faceted search, query expansion with a large number of synonyms, error-tolerant search and semantic search. It should be noted that prefix search and all these advanced types of queries are notoriously hard for INV.

1.1 Our Contribution

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 post-processing an ordinary inverted index, yielding a total index construction for HYB that is about twice as much as that of INV.

In this paper we show that with careful algorithm engineering HYB can actually be constructed twice as fast as INV. This is remarkable in two respects. First, because HYB is 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, leaving little room for further improvement of the state of the art (see Sect. 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 Sect. 1.2 below. Such a HYB block can be several MBs large and processing it takes time on the order of tens of milliseconds, whereas INV can process very short lists by an order of magnitude faster. This price is justified, however, because basic keyword queries usually contain one or more frequent words, and in that case query times for INV are on the same order as those for HYB. For a more detailed discussion of this issue, we refer to Bast and Weber (2006).

As we will see, our index construction is truly single-pass in the sense that every posting is read only once from disk in the whole process. The best construction algorithms for INV are claimed to be single-pass, too, but that is only true in the sense that they make a single pass over the original text data, whereas later passes of already (inverted and) compressed versions of that data are not counted as additional passes.

1.2 The HYB Index

We briefly recapitulate from Bast and Weber (2006) what is necessary to know about the HYB index for this paper. 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. A posting for HYB is a quadruple of document id, word-id, position, and score. HYB then consists of so-called blocks of postings, sorted by document id and position (not by word-id). The blocks 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 smaller than all words in the collection. 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 $\varepsilon$ is some constant, then HYB
can be stored in space $1 + \varepsilon$ times that of INV. Here is an example of a block that corresponds to the word range (abl, abt):

<table>
<thead>
<tr>
<th>abl - abt</th>
<th>(doc ids)</th>
<th>(word-ids)</th>
<th>(positions)</th>
<th>(scores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abl - abt</td>
<td>D401</td>
<td>ablaze</td>
<td>5</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>D1701</td>
<td>abroad</td>
<td>3</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abnormal</td>
<td>12</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abnormality</td>
<td>54</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abscess</td>
<td>4</td>
<td>0.1</td>
</tr>
</tbody>
</table>

In this example, the first list entry says that the word “ablaze” occurs in a document with id D401 at position 5 with a score of 0.3. The basic queries that the HYB index can efficiently compute are so called context-sensitive autocompletion queries, informally defined as follows. 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. For a formal definition of the problem the reader should refer to Bast and Weber (2006).

1.3 Overview of Our Construction Algorithm

Our algorithm poses three major challenges. The first challenge is to compute the block boundaries, 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 Sect. 3 we show how to compute very good estimates of the optimal block boundaries by sampling only a logarithmic number of random passages in the given document collection.

The second challenge is a truly single-pass and cache-efficient construction of the index that does not require index merging. We will show that this, even though not pointed out in previous work, is efficiency bottleneck of the inverted index construction. This is dealt with in Sect. 4.

Note that the word-ids are part of the postings and have to be stored in the index. Unlike the inverted index construction, this requires a permanent in-memory word to word-id mapping. Section 4 makes the simplifying assumption that the vocabulary fits in main memory. In Sect. 6 we propose a refinement of our basic algorithm that addresses this issue.

In Sect. 5 we experimentally compare our construction against the very fast and well engineered state-of-the-art inverted index construction of Zettair (Heinz and Zobel, 2003), which will be described in more detail in the next section.

Finally in Sect. 7 we show that the HYB index naturally supports very efficient index updates that, unlike the inverted index, perform independently of the index size.

---

1 Refer to http://search.mpi-inf.mpg.de/ for a collection of demos.
2 Related Work

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 as 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 pass over the uncompressed data is required (note that our definition of single-pass is stricter). 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 non-adjacent postings is required (Harman and C, 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 multi-way merge (Fox and Lee, 1991; 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 a new run is written to disk; (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 bit-vector that accumulates the postings from the posting stream 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 effort to reduce the merge cost: the index is partitioned into b buckets so that each bucket is responsible for a non-overlapping range of lexicographically adjacent index words. 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 (see Sect. 3.2).
3 Computing Block Boundaries

In this section we show how to compute the block boundaries of the HYB index using 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 (\( a > 1 \) is an arbitrary constant and \( n/k \) is the ideal block size).

Note, however, that sampling from disk is inefficient as one random access per word occurrence is required. We provide an alternative proof of the lemma with a tighter upper bound that for the same failure probability permits close to half as many sampled words than that known in the literature. As an alternative, in Sect. 3.3 we propose a faster disk sampling for the price of a slightly increased sampling error.

3.1 Sampling Lemma

Lemma 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 \( 1...n \) according to \( x_1, \ldots, x_k \). Let \( b_{\text{max}} = \max\{b_1, \ldots, b_k\} \). Then

\[
\Pr(b_{\text{max}} > a \cdot n/k) \leq n \cdot \exp(-s \cdot K)
\]

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

Proof Call the \( s \cdot k \) random numbers picked in the beginning splitters. Let the maximum block size be \( b_{\text{max}} \). We consider the event that \( b_{\text{max}} \) is larger than some \( b \). Then there must be a sub-range of size \( b \) that contains strictly less than \( s \) splitters. There are \( n - b + 1 \leq n \) such sub-ranges in total which means that \( \Pr(b_{\text{max}} > b) \leq n \cdot p \), where \( p \) is the probability that a fixed sub-range of size \( b \) contain less than \( s \) splitters. This probability is equal to \( \sum_{i=0}^{s-1} \binom{s}{i} \left( \frac{b}{n} \right)^i \left( 1 - \frac{b}{n} \right)^{s-k-i} \). We will derive an upper bound on the probability \( p \) 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 \( \binom{s}{i} \leq (es)^i \) and \( 1 - x < \exp(-x) \); by simple transformations we obtain

\[
p(s) \leq \exp \left( 1 + \ln(a) - a \cdot \frac{k-1}{k} \right) \cdot s
\]

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 \( p \leq \hat{s} \cdot p(s) \) provided that \( p(s) \geq p(s-1) \geq \ldots \geq p(0) \). For the binominal 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 2 in the latter inequality we obtain the following upper bound on \( p \):

\[
p \leq \exp \left( \left( 1 + \frac{\ln(s)}{s} + \ln(a) - a \cdot \frac{k-1}{k} \right) \cdot s \right)
\]
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.

**Example:** Let \( k = 2000, a = 1.5, n = 10^{10} \) (i.e. 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 6%.

frequencies in the collection.

It is unavoidable that common words with frequencies larger than the ideal block size \( n/k \) will force their containing blocks to grow larger. On the other hand it is desirable that very frequent prefixes such as `pro`, `com`, `the` etc. get blocks of their own which is automatically taken into account.

### 3.2 Query Time

We will first compute the average query time for HYB with perfect block boundaries. Let \( b_i \) be the block size of the \( i \)-th block and let \( n \) be the total number of occurrences. Consider a random query. The probability of picking a query that spans in block \( i \) is \( b_i/n \), where \( b_i \) is the size of block \( i \). Since the processing time for a query that spans in block \( i \) is proportional to \( b_i \), the expected query processing time is proportional to \( \frac{1}{n} \sum_i b_i^2 \). Hence, if ideally \( b_i = n/k \), the average query processing time is also proportional to \( n/k \). The following lemma proves that with block boundaries computed according to Lemma 1, the average query time for HYB remains on the same order as with the original, perfect block boundaries.

**Lemma 2** The expected HYB query processing time \( \frac{1}{n} \cdot E \left[ \sum_i b_i^2 \right] \) is proportional to \( n/k \) with high probability when the block sizes \( b_i \) correspond to blocks with boundaries computed according to Lemma 1.

**Proof** Assume that the probability \( \Pr(b_{\max} > a \cdot n/k) \) is bounded by some \( p \) and let \( b_{\max} \) be the maximum block size. Then

\[
\frac{1}{n} \cdot E \left[ \sum_i b_i^2 \right] = \frac{1}{n} \cdot E \left[ \sum_i b_i^2 \mid b_{\max} \leq a \cdot n/k \right] \cdot (1 - p) + \frac{1}{n} \cdot E \left[ \sum_i b_i^2 \mid b_{\max} > a \cdot n/k \right] \cdot p \\
= O \left( \frac{n}{k} \right) + O(k \cdot n) \cdot p
\]

By plugging in the bound from Lemma 1, the second term becomes \( n^2 \cdot \exp(-s \cdot K) = \exp(-s \cdot K + 2 \ln n) \), which is equal to \( o(1) \) for \( s > (2/K) \cdot \ln n \) (in practice this is satisfied for any collection size).
Table 1 Mean and standard deviation calculated over all HYB blocks sizes when computed with sampling (random and area, defined in Sect. 3.3) 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 Sect. 3.3). All numbers are averages of 100 experiments carried out on 1 GB dataset with about 2000 word boundaries (HYB blocks).

<table>
<thead>
<tr>
<th>Sampling</th>
<th>Sample Size</th>
<th>Mean</th>
<th>Standard dev.</th>
<th>Kullback-Leibler div.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full dataset</td>
<td>100%</td>
<td>80,300</td>
<td>9,922</td>
<td>0</td>
</tr>
<tr>
<td>Random</td>
<td>0.06%</td>
<td>82,283</td>
<td>11,844</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td>0.6%</td>
<td>80,533</td>
<td>10,164</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>1.2%</td>
<td>80,061</td>
<td>9,989</td>
<td>1.37</td>
</tr>
<tr>
<td>Area</td>
<td>0.06%</td>
<td>79,886</td>
<td>11,880</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>0.6%</td>
<td>79,461</td>
<td>10,109</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>1.2%</td>
<td>79,318</td>
<td>9,956</td>
<td>1.46</td>
</tr>
</tbody>
</table>

3.3 IO-Efficient Sampling

Lemma 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 access per posting. This can be prohibitively expensive: to pick a sample of 0.01% of all postings in the example above would take around 10,000 secs, in that time we could scan 500 GB 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.

Note that the assumption behind cluster sampling is that the clusters or passages have approximately equal distributions. As this is not always the case in practice, the price paid is a larger sampling error (Cochran, 1977). In our case this error depends on the ratio between the size of the passage and the number of postings drawn from it. A good trade-off between the sampling error and the sampling time was achieved for a ratio of 0.1. By a smaller ratio, assuming that the total number of sampled postings does not change, we obtain a slightly better approximation of the block boundaries but with rapidly increasing sampling time.

Table 1 shows that computing the block boundaries using cluster sampling results in block sizes close to the ideal ones. 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 will significantly deviates from the "uniform distribution" i.e. the perfect block sizes. We observed essentially the same results on all of our test collections.

4 Block Building

In this section we will show how to efficiently build the HYB index while comparing it to the state-of-the-art inversion approaches described in Sect. 2. We will make a number of simplifying assumptions, which will be addressed in subsequent sections.
Given the sequence of block boundaries, a straightforward approach to building the HYB blocks with a single-pass would be as follows. Maintain a dynamically growing in-memory data structure of postings for each block (e.g. linked lists), 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 blocks in runs by imposing a limit on their in-memory size. However, unlike for the inverted index construction, our algorithm avoids a merge of the temporary runs by writing the partial in-memory blocks to their corresponding positions on the fly, without any fragmentation. This process is explained in more detail in Sect. 4.1 and Sect. 4.4.

An engineering issue not considered in the previous work of Heinz and Zobel (2003) is the cache efficiency of the in-memory inversion. Namely, even though well approximated by a Zipfian distribution and hence with a good locality of reference, a significant fraction of the word occurrences will 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 L1-cache lines. We point out in Sect. 4.2 and experimentally confirm in Sect. 5.2 that this can significantly affect the inversion performance.

Since the word-ids are part of the postings, they have to be compressed as well. Note that the word-ids cannot be gap-encoded as they come in random order and have to be entropy-encoded instead. Near entropy-optimal but inefficient compression could be, for example, achieved by arithmetic encoding (Witten et al, 1999). In Sect. 4.3 we propose a fast two-pass compression scheme that yields only a slight loss in the compression ratio.

### 4.1 Posting Accumulation

Dynamic arrays are the standard, as well as the optimal, choice for dynamically growing data structures for index construction (Zobel and Moffat, 2006). Butcher and Clarke (2005) observed that by maintaining the array’s dynamic growth, almost as much additional space can be saved as when the array size is known in advance. Once a dynamic in-memory array has been assigned to each HYB block, postings from the posting stream are accumulated and compressed on the fly with an Elias-gamma code for the doc-gaps, position-gaps and word-frequencies, and Zipf compression (see Sect. 4.3) for the word-ids.

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 Sect. 6). Since assigning lexicographic word-ids on the fly is not easy, a non-lexicographic to lexicographic word-id permutation (obtained by sorting the vocabulary) is stored at the end of the construction. To determine the corresponding HYB block for each posting, a fast in-memory data structure is employed that computes the HYB block and then stores the information in the word’s vocabulary entry so that the computation is done only once per distinct word.

The single-pass approach by Heinz and Zobel (2003) sorts the index words whenever a new compressed run is being written out to disk as otherwise merging of the
Fig. 1 Throughput (given in MiB/s) of HYB and inverted index posting accumulation approaches (defined in Sect. 4) to in-memory invert a run of 100 million occurrences.

temporary inverted lists at a later stage 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. This means that our overall algorithm requires almost no sorting.

4.2 Cache Efficiency

We already mentioned that the straightforward posting accumulation by Heinz and Zobel (2003) is cache-inefficient since a significant fraction of the postings impose cache misses when appended to their HYB blocks. To address this issue, we propose a multi-level posting accumulation scheme by grouping consecutive HYB blocks in groups so that the total number of groups is close to the number of cache lines of the L1-cache. The postings from each group are recursively assigned to the next level of groups until each group is comprised of a single HYB block. The number of cache misses will be minimized if the number of groups per level is equal, i.e. $\sqrt{k}$ ($l$ is the number of levels). We note that the block groups will still be well compressible (for in-memory posting accumulation) as shown by Bast and Weber (2006), because the HYB index with positional information has smaller empirical entropy than that of the inverted index for any choice of block size.

The price paid for the above procedure is multiple copies of each posting. Note, however, that a cache hit can be 5 to 100 times faster than a cache miss. Let us for the sake of simplicity consider a fully associative cache of size $c$ and HYB blocks of equal size. Then regardless of the replacement policy, the expected numbers of cache misses
is \( l \cdot n \cdot (1 - c/\sqrt{k})^+ \). Obviously, for certain \( l \) we could reduce the number of cache misses to 0, however ideally one should minimize

\[
\text{argmax}_l \cdot \left( 1 - \frac{c}{\sqrt{k}} \right)^+ \cdot T_{\text{miss}} + \frac{c}{\sqrt{k}} \cdot T_{\text{hit}}
\]

where \( T_{\text{miss}} \) and \( T_{\text{hit}} \) respectively are the costs for a cache miss and a cache hit. The best results in practice were achieved for \( l = 2 \) (two-level posting accumulation). 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. The number of cache misses for \( l = 2 \) will be less than the number of cache misses for \( l = 1 \) if \((k - c)/(2\sqrt{k} \cdot (\sqrt{k} - c)^+) > 1\). This inequality is almost always satisfied if \( k < 4 \cdot c^2 \), with the number of cache misses many times smaller for \( \sqrt{k} \sim c \) and essentially 0 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 8 KB.

**Example:** Let’s consider an 8 KB cache with 64 B cache lines \((c = 128 \text{ cache lines in total, } \sqrt{k} \text{ is usually less than } 100)\). 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 reasoning above, the posting accumulation cost for \( l = 2 \) is roughly 2 times smaller for \( 1,000 \leq k \leq 10,000 \).

Figure 1 shows that the efficiency of the simple HYB block posting accumulation approaches that of the inverted index when the number of HYB blocks is large. The efficiency of the two-level posting accumulation, on the other hand, is not affected. This is due to the fact that the number of block groups remains smaller than the number of cache lines. Note that the above model, though simplified, matches Fig. 1.

4.3 Fast Word-id Compression

Our compression scheme is based on the assumption that the input (collection of word-ids) has a near Zipfian distribution. Given this, it is not hard to show that universal encoding (Moffat and Zobel, 1996) that assigns \( \sim \log x \) bits for a word of tank \( x \) obtained by sorting the word-ids in order of descending frequency, is near entropy optimal as well\(^3\). We refer to this scheme as Zipf compression. An obvious drawback here is that a full sort of a large number of word-ids is required to obtain the ranks. Assume that instead of sorting, the rank of each word is obtained by a MTF (move-to-front) transform over the input, with the intuition that frequent word-ids are likely to end up near the front of the list and thus get smaller ranks. The following lemma shows that this will give us an efficient compression algorithm for the price of a very small loss in the compression ratio.

**Lemma 3** Given a Zipfian distribution of the input, the ranks obtained by a MTF-transform have expected values that are not much larger than the true ranks (determined by the skewness of the Zipfian distribution of the input).

\(^2\) \((x)^+ = x \text{ if } x \geq 0 \text{ and } 0 \text{ otherwise.}\)

\(^3\) 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 term is \( \Omega/i \), then the per-item entropy is \( \sum_{i=1}^{k} \Omega/i \cdot \log_2(i/\Omega) \) and the average per-item-space consumption would be close to \( \sum_{i=1}^{k} \Omega/i \cdot \log_2 i \).
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 $X_r \leq X_j$.

Note that to additionally insure that the most frequent words always get the lowest
ranks, one could sort only a small subset of the input words (which with high probability
will contain the most frequent words) and then reserve the obtained ranks for these
words.

The loss in compression ratio on our two test collections in practice was surprisingly
small: less than 1% on Wikipedia and about 1% on TREC Terabyte (see Sect. 5.1).
We note that this compression scheme is almost as fast as coding with gaps and Elias
gamma code.

4.4 In-place Block Writing

Once the memory limit for the in-memory HYB blocks has been reached, each group of
HYB blocks is decompressed and each individual HYB block is restored and optionally
re-compressed (this time the doc-ids are compressed with Golomb code). Instead of
writing the partial HYB blocks contiguously to disk for a later merge, we exploit the
fact that the HYB blocks are of roughly equal size and are reasonably large in number
to write them on the fly (in-place), in a single pass. We show below and experimentally
confirm in Sect. 5, that this 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.

Since writing the blocks in-place requires knowing the block sizes in compressed
format, we first compute an initial estimate for each block size by running an in-
memory version of our algorithm on a random sample of documents. To prevent a
miss-estimated block from overflowing and overwriting the next block, a procedure
called space-propagation is proposed in Sect. 4.5.

Note that in-place writing is impossible to achieve on the inverted index, first,
because the size of short inverted lists cannot be sufficiently well estimated and second,
because the number of inverted lists is in the order of millions.

In the following we specify and compare the construction disk cost of the inverted
index to that of the HYB index. Let $R_u$ be the cost of sequentially reading the uncom-
pressed collection, $R_c$ be the cost of sequentially reading the compressed temporary
file, $W_c$ be the cost of sequentially writing the temporary compressed file, $n$ be the
total number of runs for both algorithms and $t_s$ be the average disk seek time.
To reduce the number of disk seeks, a buffer of size $B$ is allocated to each on-disk run.
The total disk cost for the inverted index construction is then roughly equal to

\[
R_u + W_c + \left( R_c + W_c + \max \left\{ \frac{W_c}{B}, n \right\} \cdot t_s \right) + (R_c + W_c)
\]

\footnote{Note that the authors interchangeably use \textit{disk seek time} and \textit{disk access time}. In both cases we mean the full disk access time}
where the second and the third terms correspond to the cost to merge and permute the merged file. If twice more disk space than the size of the index file is allowed (merge is not *in-situ*), then the third term should be skipped (note that our approach does not require additional disk space, see Sect. 5.3). The total disk cost for the HYB construction is equal only to

\[ R_u + (W_c + n \cdot k \cdot t_s) \]

The second term in the brackets corresponds to the total seek time since each block requires a single disk seek for each run. Note that the above formula is pessimistic since it ignores the fact that the disk seek time depends on the track distance (Popovici et al., 2003). Our algorithm makes equidistant jumps from one block to the adjacent block and hence keeps this distance small e.g. not larger than 5 MB for a 10 GB collection. Table 2 shows this empirically. A subtlety here is a slight inherent non-linearity of the disk cost (dominated by disk seeks) as the seek time \( t_s \) slightly increases with the collection size (reflected in our results, Table 3). This behavior, however, does not affect the running times seriously even on collections of the order of TREC Terabyte.

**Example:** Consider a 50 GB collection with 20 GB of compressed (positional) index. Let the disk reading and writing throughput be 50 MB/s and let a single disk seek takes 5 ms. Let the memory limit in both cases be 512 MB (= 40 runs). For \( k \sim 2000 \) the disk costs for INV and HYB respectively are 3072 and 1833 secs.

### 4.5 Space Propagation

We already mentioned the problem that blocks with initially underestimated sizes will overflow and overwrite the neighboring blocks. Note that no matter how good the initial estimation is, roughly half of the blocks will have sizes that are underestimated. Also note that once the block writing starts, further movement of the blocks is not possible. A straight-forward solution is to write the overflowed data at the end of the index file for the price of an additional disk access at query time. Another easy solution is to have a separate file for each HYB block. However, the implementation of this solution requires a significant increase of disk access time during construction.

An alternative solution 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. Below we give a description of the procedure and provide theoretical evidence that it fails 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 lender block is not large enough to fit its own data and the data of

---

**Table 2** Average random and equidistant seek time over 2000 disk seeks

<table>
<thead>
<tr>
<th>File size</th>
<th>100 MB</th>
<th>1 GB</th>
<th>10 GB</th>
<th>100 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average random seek time</td>
<td>1.5 ms</td>
<td>5.2 ms</td>
<td>11.1 ms</td>
<td>14.5 ms</td>
</tr>
<tr>
<td>Average equidistant seek time</td>
<td>0.8 ms</td>
<td>4.7 ms</td>
<td>5.6 ms</td>
<td>6.2 ms</td>
</tr>
</tbody>
</table>
the borrower in the same time, it becomes a borrower of the next neighbor. Hence, a cascade of blocks that borrow space can be formed that terminates when the space request is amortized.

After a significant fraction of the data has been processed (e.g. 70% - 80%), the block sizes are reestimated whenever a new run is being written to disk in order to determine if a certain block will overflow. After each re-estimation 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 i.e. the block is extended for the size of the current run.

The space propagation of a certain block fails if both neighbors of that block have free space that is less than the required space. 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. 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 size. This increases the chance that neighboring blocks are of roughly the same size and will not interrupt the propagation.

4.5.1 Theoretical Evidence

To provide theoretical evidence that supports this procedure we propose the following model. Let $\epsilon$ be the estimation error of each block size given in percents. Let us assume that the initial estimated size of each block varies from its true size with Gaussian error with mean $\mu = 0$ and variance $\sigma^2$. Let the correct block size be $B$ and let us assume that $1 - r$ percent of the block size (equal to e.g. 70% - 80%) is enough to provide a reliable estimation for the whole block. We also assume that there are enough runs so that the propagation “converges”. We will compute an upper bound on the probability that at least one space propagation failure 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 maximum borrowable space (from both of its neighbors) is insufficient to fit its data, i.e. if $B \cdot \epsilon < -2 \cdot r$. Note that by adding another block, if isolated, the probability of the latter 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 $B \cdot \epsilon_1 + B \cdot \epsilon_2 < -2 \cdot r$ (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 group bound we obtain

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

(4)

The question now is whether the joint failure probability for a group of $n$ blocks ($p_n$) increases when $n(\leq k)$ grows. The joint propagation failure probability for a group of
The value of the $z(n)$ argument of the space propagation failure probability with $r = 70\%$ (defined in Sect. 4.5) for two directional propagation and 3 block space blow-up factors. (note that $p(-4) \sim 7.7 \cdot 10^{-9}$, $p(-5) \sim 7.6 \cdot 10^{-13}$, $p(-6) \sim 1.0 \cdot 10^{-17}$, $p(-7) \sim 2.0 \cdot 10^{-23}$)

Fig. 2 Value of the $z(n)$ argument of the space propagation failure probability with $r = 70\%$ (defined in Sect. 4.5) for two directional propagation and 3 block space blow-up factors. (note that $p(-4) \sim 7.7 \cdot 10^{-9}$, $p(-5) \sim 7.6 \cdot 10^{-13}$, $p(-6) \sim 1.0 \cdot 10^{-17}$, $p(-7) \sim 2.0 \cdot 10^{-23}$)

According to the assumptions, $\sum_{i=1}^{n} \epsilon_i$ has Gaussian distribution with mean $n \cdot \mu$ and variance $n \cdot \sigma^2$, which means that $p_n$ can be written as $p_n(z) = 1/2(1 + \text{erf}(z(n)))$ where $\text{erf}()$ is the Gaussian error function and $z(n) = (-2r - n\mu)/(\sigma\sqrt{2n})$. Note that $p_n(z)$ is a strictly increasing function of $z(n)$, meaning that $p(z)$’s behavior is solely determined by $z(n)$. Obviously if $\mu = 0$, then $z(n)$ strictly increases with $n$, resulting in large values of $p_n(z)$ (e.g. 1/2 for $n = 2000$). However, if $\mu > 0$ (a small blow-up in the block sizes), then $z(n)$ reaches a maximum for $n = 2 \cdot r/\mu$ and then starts to decrease, resulting in extremely small values of $p_n(z)$ for large $n$ (e.g. 2000). Figure 2 plots the $z(n)$ value against the number of blocks $n$ with different space blow-up factors.

5 Experiments

We compare the performance of our HYB index construction algorithm to a state-of-the-art inverted index construction algorithm, namely the one implemented as part of Zettair5, which essentially implements the ideas from Heinz and Zobel (2003) (slightly

5 http://www.seg.rmit.edu.au/zettair/
varying from the original single-pass approach by using log-10 partitioning). According to the large study of Middleton and Baeza-Yates (2007), Zettair’s index construction is indeed the fastest on the open source market to date.

Our implementation of the HYB index builder, as described in Sect. 3 and 4, 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 -O3 flag. All experiments were performed on a machine with 16 GB of RAM (with contents flushed before every execution), 4 dual-core AMD Opteron 2.8 GHz processors (we used only one core at a time), operating in 32 bit mode and running Debian 4.1.1-19 with a standard ATA (Hitachi Deskstar) hard drive with 7200 RPM and reported average access time of 12.9 ms. We used the latest (0.9.3) version of Zettair.

5.1 Test Collections

Our experiments were carried out on two collections:

**Wikipedia:** a dump of the English Wikipedia, with a raw size of 12.7 GB, 2,874,500 million documents, 795 million word occurrences, and a vocabulary of 8 million distinct words.

**TREC Terabyte:** the TREC GOV2 corpus with a raw size of 426 GB, 25,204,103 documents, 23 billion word occurrences, and a vocabulary of 57 million distinct words.

To get a better picture on the scalability of both algorithms we ran the experiments on subsets of size 25%, 50% and 100% of the full collection. To ensure that both parsers produce the same sequence of words we replaced each sequence of non-word characters in the collections by a single space.
Table 3  Elapsed index construction time in minutes to construct a word-level index with our HYB builder and Zettair on all our test collections.

<table>
<thead>
<tr>
<th></th>
<th>Wikipedia 25%</th>
<th>TREC Terabyte 50%</th>
<th>TREC Terabyte 100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>HYB</td>
<td>4.7 min</td>
<td>75.1 min</td>
<td>155.8 min</td>
</tr>
<tr>
<td>Zettair</td>
<td>7.3 min</td>
<td>92.3 min</td>
<td>288.9 min</td>
</tr>
</tbody>
</table>

5.2 Index Construction Time

Compared to Zettair, our HYB index builder is faster by a factor of 1.6 (on Wikipedia) to 2.1 (on Terabyte). For both algorithms, we imposed a memory limit of 500 MB for the in-memory posting accumulation (we used the `--big-and-fast` option on Zettair) without taking into account the additional memory usage of the vocabulary and other auxiliary data structures. In all experiments we picked the HYB block sizes as \( \frac{N}{5} \), where \( N \) here is the number of documents in the collection (see Bast and Weber (2006)).

Table 5.3 shows a break-down of the running time of our HYB builder. The most expensive phase is the parsing and looking-up of word-ids 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 bit-wise 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).

While Zettair spends one third of its time merging runs, our index builder spends only one fifth of its time on block writing. This is due to the fact that Zettair has to fully read and write the temporary index file more than once. We note that the actual bottleneck of the index merging is not caused by the disk seeks but is rather the result of reading and writing large amounts of data from disk. As a consequence, the index partitioning improvement proposed in the last section of Heinz and Zobel (2003) which aims to reduce the number of disk seeks was only marginally faster in practice.

Only about 16% of the time it takes to construct our index is spent on posting accumulation, that is, in-memory inversion. As suggested by Fig. 1, this cost in the inverted index construction is more than twice as much. The latter shows that cache misses seriously affect the efficiency of the in-memory inversion of the single-pass approach. For whatever reason, posting accumulation costs are not reported in the elapsed-time figures of Heinz and Zobel (2003).

In Sect. 4.4 we saw that the disk access cost of our algorithm depends on the number of blocks and the total number of runs which in turn depends on the main memory limit. Table 5 shows the running time of the HYB construction when the number of blocks increases and the amount of available memory is constant (in this experiment we used a memory limit of 200 MB). An increase by a factor of 10 in the number of blocks increased the total running time by about 35%. To achieve the same speed for larger numbers of blocks our algorithm requires more available memory.
Table 4 Break-down of the total elapsed index construction time in four major steps for the TREC Terabyte collection.

<table>
<thead>
<tr>
<th></th>
<th>Parse &amp; Look-up</th>
<th>Accumulate</th>
<th>Compress</th>
<th>Disk I/O</th>
<th>Sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32%</td>
<td>16%</td>
<td>20%</td>
<td>29%</td>
<td>3%</td>
</tr>
</tbody>
</table>

Table 5 Increase in HYB construction time given in percents for varying number of HYB blocks and constant memory limit of 200 MB. The experiment was carried out on one quarter of the full TREC Terabyte collection.

<table>
<thead>
<tr>
<th>No. blocks</th>
<th>Time increase</th>
<th>2 \cdot 10^3</th>
<th>3 \cdot 10^3</th>
<th>4 \cdot 10^3</th>
<th>5 \cdot 10^3</th>
<th>10 \cdot 10^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 \cdot 10^3</td>
<td>+0%</td>
<td>+10%</td>
<td>+16%</td>
<td>+24%</td>
<td>+34%</td>
<td></td>
</tr>
<tr>
<td>3 \cdot 10^3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 \cdot 10^3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 \cdot 10^3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 \cdot 10^3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3 Temporary Disk Space

The additional disk requirement of our HYB construction 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 Sect. 4.4). The peak disk space usage in our experiments varied from 100% of the size of the final index file on small to medium-size collections, up to 103% on the full TREC Terabyte (i.e. up to 3% overhead).

The reported additional temporary disk space usage in Heinz and Zobel (2003) is about 26% for document-level inverted indices and about 8% for word-level inverted indices. However, we found out that during the parsing and merging phase the peak space usage of Zettair on the TREC Terabyte was correspondingly about 165% and about 143% of the final size of the 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 might be undesirable.

6 Refinements

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

6.1 Large Hash Keys

Instead of permanently storing word - word-id pairs, compute the word-ids with an additional hash function, making it possible to flush to disk once the vocabulary size approaches the memory limit. At the end of the inversion the sorted vocabulary fractions\(^6\) 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 range size of the hash function.

\(^6\) The vocabulary is sorted before flushing it to disk
Table 6  Increase in the HYB construction time on one quarter of the TREC Terabyte collection when different in-memory vocabulary size limits are used (see Sect. 6). The total number of distinct words was roughly 28.5 millions and the construction (without vocabulary limits) took 43 mins.

<table>
<thead>
<tr>
<th>Vocabulary size limit in %</th>
<th>50%</th>
<th>33%</th>
<th>25%</th>
<th>20%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Hash-keys</td>
<td>+7%</td>
<td>+10%</td>
<td>+12%</td>
<td>+11%</td>
</tr>
<tr>
<td>On-disk vocabulary</td>
<td>+14%</td>
<td>+20%</td>
<td>+22%</td>
<td>+23%</td>
</tr>
</tbody>
</table>

(i.e. 64-bit hash keys). Let $X_i$ be the indicator random variable that the $i$-th word collides with an earlier word. Since $Pr(X_i = 1) = (i - 1)/r$, the expected number of coalitions 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 Terabyte collection has roughly 50 million distinct terms).

The overhead imposed by this procedure amounts to only about 10% of the total indexing time in practice (see Table 6).

6.2 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 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 its initial.

The overhead here is about twice larger than that in the previous approach and it comes from frequently reading a gradually growing fraction of the vocabulary from disk.

6.3 Multiple Cores

Our algorithm can be easily extended in a multi-core 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 in addition 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 $c$ times
speed-up. Because each of the cores is responsible for only a certain range of words their vocabulary sizes should not be a problem.

The second approach is orthogonal to the first and exhibits more parallelism: instead of a group of HYB blocks, each core is responsible for a fraction of the whole dataset. Hence, 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 $c$. To address the vocabulary size problem we make use of the fact that most of the words in large vocabularies are rare. For example, if the TREC Terabyte 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. To prevent multiple word-ids, each core should use an identical hash-function to generate the word-ids (as in Sect. 6.1). Note that additional header data for the order of runs in each block should to be included in the index since the runs will be written out of order.

We note that to the best of our knowledge, we are not aware of any previous work done on parallel inverted index construction.

7 Efficient Index Maintenance

So far it has been shown that our HYB index construction outperforms the fastest inverted index construction on static collections by a large margin. However, equally important is the setting where new documents arrive at a high rate and efficient index maintenance is required. Three main strategies for index maintenance were proposed and experimentally compared in Lester et al (2004). We will show that one of the update strategies that performed unsatisfactory on the inverted index, is well suited and outperforms the previously best known strategy on the HYB index.

All index update strategies in a nutshell 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 up (Zobel and Moffat, 2006). According to Lester et al (2004), re-merge is the fastest update strategy for the inverted index. It works by employing a sequential merge of the new and the existing on-disk posting lists as follows. 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 (and vice-versa), the in-memory posting list is written to the new index and an advance is made to the next index word. 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 indices is made to a new index.

The in-place strategy works as follows: once the main memory is exhausted, an attempt is made to append each in-memory posting list to the corresponding on-disk list on the fly. In case of insufficient disk space, the entire posting list is moved to a location with enough space, inevitably causing index fragmentation. This strategy has three inherent problems: first, since a document can contain a large number of distinct words a prohibitive number of random disk accesses is required for the update, second, there is no obvious strategy how to preallocate disk space during construction so that at later stage index fragmentation is minimized, and third, repetitive relocation of whole posting lists additionally imposes significant disk cost.
Interestingly, all of the above problems are solved when the in-place strategy is applied to the HYB index. The first problem is addressed by the fact that the number of HYB blocks is typically 1000 times smaller than the number of distinct words. The number of disk accesses, hence, should not be a problem. In fact, a dynamic update is identical to writing a new run in our HYB construction algorithm from Sect. 4.4. To address the second and the third problem, note that on the one hand, each HYB block size is already estimated and known in advance, and on the other, the block boundaries are computed such that all blocks are of roughly the same size (see Sect. 3). The latter implies that each block should grow about equally fast in the future and therefore allocating free disk space after each block is an adequate preallocation strategy. Note that one should keep in mind that the whole index will have to occasionally be copied to a new location if the blocks are running out of free space. This could be prevented by maintaining a separate file for each block for the price of a increased index construction time (see Sect. 4.4).

To compare the costs of the two merging strategies on the HYB index, let $U$ be the update size, $B$ the in-memory buffer size allocated for the in-memory index and $B_c$ be the cost to contiguously write $B$ bytes on disk. Let $R_c$ and $W_c$ be respectively the costs to read and write the compressed index file as they were defined in Sect. 4.4. Then the update cost of the re-merge strategy is roughly equal to

$$\frac{U}{B} (B_c + R_c + W_c)$$

---

7 We already argued in Sect. 4.4 that in-place block writing is more efficient than merging the entire HYB index.

8 All costs are given in seconds.
Let \( t_s \) be the cost for a single disk access and let \( k \) be the total number of blocks. The update cost of the in-place strategy is roughly equal to

\[
\frac{U}{B} (B_c + k \cdot t_s)
\]

In practice the gap between the above costs is enormous due to the obvious drawback of the re-merge strategy that even for small updates a full copy of the entire index is required. The in-place update strategy on the other hand is independent of the index size. Figure 4 shows the running times of the re-merge and the in-place update strategy for different buffer sizes and update size comparable to that of the full index (i.e. one half). Note that the in-place strategy is even faster for updates that are significantly smaller than the index size.

**Example:** Consider a HYB index with size of 20 GB. Let the disk throughput be 50 MB/s and let a single disk seek take 5 ms. Let the size of the update be equal to 2 GB and let the in-memory buffer size be 1 GB. For \( k \sim 2000 \), the update costs of the re-merge and the in-place strategy respectively are roughly 862 secs. and 62 secs.

8 Conclusions

We have carefully designed and engineered a construction algorithm for the alternative data structure of Bast and Weber (2006) called HYB index that has all the advantages of the inverted index but in addition supports a palette of advanced queries. We have shown that our construction algorithm is twice as fast as the fastest inverted index construction from Zettair. Moreover, our algorithm is simple to implement, supports very efficient index maintenance, and, unlike for the inverted index, does not require additional data structures for external sorting. Our approach is truly single-pass in that the bulk of the word occurrences are each touched only once, it is cache-efficient, does not require in-memory or external sorting and during construction uses no more space than the final compressed index.

References

Bast H, Weber I (2006) Type less, find more: fast autocompletion search with a succinct index. SIGIR
Butcher S, Clarke CLA (2005) Memory management strategies for single-pass index construction in text retrieval systems. Tech. rep., School of Computer Science, University of Waterloo, Canada
Cochran WG (1977) Sampling techniques, 3rd edition
Witten IH, Moffat A, Bell TC (1999) Managing gigabytes: Compressing and indexing documents and images