Efficient Single-Pass Index Construction for Text Databases

Steffen Heinz Justin Zobel*
School of Computer Science and Information Technology, RMIT University
GPO Box 2476V, Melbourne 3001, Australia

Abstract

Efficient construction of inverted indexes is essential to provision of search over large collections of text data. In this paper, we review the principal approaches to inversion, analyse their theoretical cost, and present experimental results. We identify the drawbacks of existing inversion approaches and propose a single-pass inversion method that, in contrast to previous approaches, does not require the complete vocabulary of the indexed collection in main memory, can operate within limited resources, and does not sacrifice speed with high temporary storage requirements. We show that the performance of the single-pass approach can be improved by constructing inverted files in segments, reducing the cost of disk accesses during inversion of large volumes of data.

Keywords Inverted files, index construction, text databases

*Justin Zobel is the contact author for all correspondence, at jz@cs.rmit.edu.au.

1 Introduction

The size and growth rate of today's text collections bring new challenges for index construction. Building an index for a large text collection may involve the management of millions of distinct words, and of billions of occurrences of words in the text.

Most approaches to inversion are well suited to index a text collection in the megabyte range. For a small collection, the vocabulary and all postings (that is, representations of word occurrences) can be accumulated in main memory allowing fast index construction within the capabilities of a standard workstation. However, for large volumes of text, only a few techniques are practical. Some avoid high memory requirements by storing postings temporarily on disk, hence trading main memory with temporary disk space at the expense of costly random disk accesses that penalise severely their performance. More advanced approaches bound temporary storage requirements, but require an abundance of main memory. On the other hand the performance of standard workstations has increased dramatically over the last decades. It could be inferred from these improvements that index construction could eventually be performed completely in main memory. However, the volume of data to be indexed may well grow with our capacity to store it. For example, a small text collection of documents drawn from the Web might contain 100 Gb of data. An efficient representation of the index might be 20 Gb; the distinct index terms alone may require 200-400 Mb of memory. Since the size of such databases is growing rapidly, it is reasonable to assume that in-memory indexing remains impractical for large collections in the future even if the performance of standard workstations continuous to improve at the current rate.

The topic of this paper is the efficient index construction for text collections. The fastest approach previously described in the literature is sort-based inversion (Moffat & Bell 1995, Witten, Moffat & Bell 1999). It uses a single pass over the input, in which uncompressed postings are accumulated in main memory, sorted, compressed, and written out to disk in runs that are later merged to obtain the final index. An alternative is a two-pass method (Fox & Lee 1991, Moffat & Bell 1995), where temporary space requirements are small due to application of efficient compression schemes but the entire input must be processed twice, at significant cost. Both approaches are scalable, that is, can operate within fixed memory limits, but both require that the complete vocabulary of the collection be kept permanently in main memory, which may be not possible if resources are limited since the vocabulary of a collection increases approximately linearly with collection size (Williams & Zobel 2001).

We investigate in this paper an inversion approach that has several advantages over these techniques: it performs only one pass over the input; does not require that the complete vocabulary of the indexed text collection be held in main memory; can operate within limited volumes of memory; and does not need large amounts of temporary disk space. It is faster than existing approaches, as shown by our analysis based on a computational model as well as our experiments on reference text collections. The approach is essentially a combination of standard methods described previously—in-memory construction of runs that are saved to disk and then merged—but we show that, with careful design and key innovations, it significantly outperforms the alternative methods. We have constructed a word-level inverted file for a 20 Gb collection of Web documents on a Pentium III with 700 MHz using not more than 256 Mb of main memory in around 155 minutes,

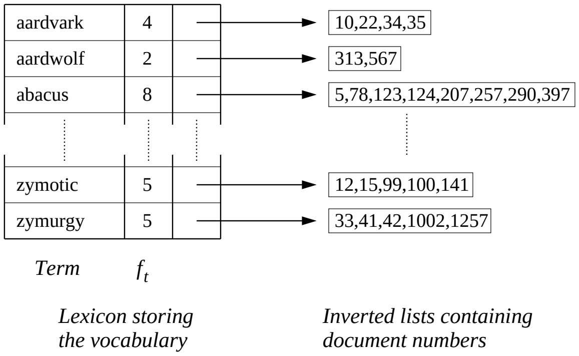


Figure 1: *Structure of a simple inverted file. The lexicon maintains for each index term a reference to an inverted list on disk. An inverted list contains document numbers in ascending order.*

which is approximately 8 minutes per gigabyte of text. Document-level inverted files can be built at a rate of less than 5 minutes per gigabyte of text.

2 Inverted files

An inverted index (Frakes & Baeza-Yates 1992, Salton & McGill 1983, Witten et al. 1999), also known as *postings file*, or simply *inverted file*, maintains the set of distinct terms of a text collection in a lexicon. The lexicon keeps for each index term a pointer to an inverted list on disk, which comprises a list of postings that describe all of the term's occurrences in the collection.

For many ranking applications, a *document-level* inverted file offers the best trade-off between evaluation efficiency and index size (Witten et al. 1999). In a document-level inverted file, the inverted list for each term t is of postings of the form $(d, f_{d,t})$ where d is a document number and $f_{d,t}$ is the document frequency, that is, the number of occurrences of t in d. The postings are usually stored in an ascending order by their document numbers. Hence, an inverted list of a document-level inverted file for a term t is of the form

$$\langle f_t; (d_1, f_{d_1,t}), (d_2, f_{d_2,t}), \cdots, (d_{f_t}, f_{d_{f_t},t})\rangle$$

with $d_i < d_j$ for $1 \leq i < j \leq f_t$, where f_t is the number of documents containing t. Figure 1 gives an overview of the structure of a document-level inverted file.

A fine-grained *word-level* inverted file additionally includes in each posting the ordinal word positions of a term in a document in ascending order to facilitate query evaluation. Hence, the form of an inverted list of a word-level inverted file for a term t is

$$\langle f_t; (d_1, f_{d_1,t}, l_{t,d_1,1}, \cdots, l_{t,d_1,f_{d_1,t}}), \cdots (d_{f_t}, f_{d_{f_t},t}, l_{t,d_{f_t},1}, \cdots, l_{t,d_{f_t},f_{d_{f_t},t}})\rangle$$

where $l_{t,d,i}$ are the ordinal word positions of t in document d and $l_{t,d,i} < l_{t,d,j}$ for $1 \leq i < j \leq f_{d,t}$.

Inverted lists are usually stored contiguously on disk in a compressed format, with each list individually accessible. Compression reduces the storage required for an inverted file and can also reduce the time required to evaluate a query (Moffat & Zobel 1996). The main insight that leads to efficient compression of inverted lists is that the document numbers and word positions in inverted

Table 1: *List of symbols used in the analysis of the inversion approaches and performance parameters.*

Parameter	Symbol	Value
Main memory size (Mb)	M	256
Average disk seek time (sec)	t_s	9×10^{-3}
Disk transfer time (sec/byte)	t_t	5×10^{-8}
Time to parse one term (sec)	t_p	8×10^{-7}
Time to compress a byte (sec)	t_c	5×10^{-7}
Time to lookup a term in the lexicon (sec)	t_l	6×10^{-7}
Time to compare and swap two 12 byte records (sec)	t_w	1×10^{-7}

lists are ascending sequences of integers. It is then necessary to store only the differences, or gaps, between document numbers, rather than the numbers themselves, which can be efficiently represented by Elias codes (Elias 1975) or Golomb codes (Golomb 1966). In practice, good compression of document-sorted inverted files is achieved by representing the gap-coded document numbers, or *d-gaps*, in Golomb codes, and the within-document frequencies $f_{d,t}$ and the word-position gaps in Elias codes (Bell, Moffat & Witten 1994, Moffat & Bell 1995, Witten et al. 1999). In recent work it has been shown that byte-wise coding schemes result in slight losses of compression efficiency, but greatly reduce decoding time (Scholer, Williams, Yiannis & Zobel 2002).

There are several proposed approaches for inverted file construction (Fox & Lee 1991, Harman & Candela 1990, Moffat & Bell 1995, Rogers, Candela & Harman 1995). A recent survey on inversion approaches (Witten et al. 1999) shows that only two of them are scalable: one, a two-pass in-memory approach by Fox & Lee (1991), which has been further improved by Moffat & Bell (1995); and the other, a single-pass sort-based inversion approach proposed by Moffat & Bell (1995). These approaches are applicable to text collections of any size and work within limited volumes of main memory and temporary disk space. In the next section, we discuss both schemes in detail and briefly review simple approaches that are only applicable to small- and medium-sized collections due to their high resource requirements. However, before we proceed to our review, we describe a computational model for index construction.

Computational model

In order to estimate the running times of inversion approaches, we use the computational model as shown in Table 1. The model describes the computational costs of basic operations performed during inverted file construction on a single workstation with a local disk. Similar models have been used elsewhere (Frakes & Baeza-Yates 1992, Moffat & Bell 1995, Witten et al. 1999).

The parameter values are based on the performance of a Pentium III 700 MHz with 512 Mb of main memory running Linux. We measured on this machine, for example, the virtual running time to parse a collection into terms and derived an average cost estimation t_p. We also empirically determined the average cost t_l to lookup a term in a lexicon that is implemented as a fast variant of a standard hash table with 2^{20} slots (Zobel, Heinz & Williams 2001). The time costs for t_c and t_w were estimated in a similar way. The disk-related parameters are based on the vendor's

Table 2: *Statistics of three reference text collections. The inverted file sizes are based on a compression scheme where d-gaps are Golomb-coded and within-document frequencies $f_{d,t}$ as well as word positions are represented in gamma-codes.*

	Symbol	Bible	Web V	Web XX
Size (Mb)	S	4.4	5,120	20,480
Distinct terms	n	13,569	2,964,428	6,837,589
Term occurrences ($\times 10^6$)	C	0.8	323.9	1,261.5
Documents	N	31,101	928,113	3,560,951
Postings ($d, f_{d,t}$) ($\times 10^6$)	P	0.6	140.6	543.2
Avg. number of term occurrences per document	C_{avg}	26	348	354
Avg. number of index terms per document	W_{avg}	21	151	153
Percentage of words that occur once only	H	33	45	44
Size of compressed document-level inverted file (Mb)	I	0.64	180	697
Size of compressed word-level inverted file (Mb)	I_w	1.27	672	2,605

specification of a standard 40 Gb hard disk with a 2 Mb buffer and 7,200 rpm spindle speed.

The aim of the computational model is to derive cost formulae for inversion approaches and to calculate their running times on reference collections. However, the model is not precise enough to calculate exact running times, since caching effects, costs for memory allocation and freeing, and other work done by the operating system are not taken into account. Nevertheless, these performance estimates allow comparison of inversion approaches.

We use three reference collections with the statistics shown in Table 2 to determine the resource requirements of inversion approaches. The first collection, of 4.4 Mb, is the King James version of the Bible, where each verse is taken as a document; book names, chapter numbers, and verse numbers are removed. This collection was chosen because it is readily available and can be meaningfully divided into a reasonable number of documents. The other two reference collections are drawn from the large Web track in the TREC project (Harman 1995), and are typical of the text indexed by current search engines. The smaller collection *Web V* is 5 Gb and the larger one *Web XX* is 20 Gb; both are collections of Web documents in several languages. The Web collections have large vocabularies, consisting of almost three million and seven million distinct terms, respectively. These collections feature the skewed distribution of terms that is typical for natural language text. For example, *Web V* has around 324 million term occurrences, of which just around 3 millions are distinct terms. The most frequent terms in the collection are “the”, “of”, and “and” respectively; the word “the” occurs in on average one in twenty-seven occurrences, almost twice as often as the second most frequent term. On the other hand, around 45% of all distinct terms occur only once.

We derived these statistics by parsing the collections into index terms without applying stemming, stopping, nor case-folding. We took as a term every sequence of characters, up to a maximum of 64, that contained no more than two digits and did not start with a digit. We imposed these restrictions to prevent long lists of page numbers or other enumerations from being included in the index. However, we note that such numbers may well be indexed in practical systems. We also excluded during parsing any XML or HTML tags, special characters, and punctuation.

3 Existing approaches to inversion

Simple in-memory inversion

Consider a document-level inverted file for a text collection. Viewing the inverted file not as a collection of inverted lists but as an inversion matrix is the main idea of a simple inversion approach. Each column in the matrix represents a document and each row an inverted list. The entries of the matrix represent postings. Hence, assuming a cell of the matrix at position (m, n) consists of the frequency $f_{m,n}$ of the nth term of the collection in document m.

The inversion matrix is constructed by two passes over the text. In the first pass, the set of index terms and the number of documents in the collection are determined. Then an empty matrix is allocated that has one row for each distinct term and a column for each document. In the second pass over the collection, the documents are processed in turn and the matrix is filled column by column. For each term parsed from the current document d, the corresponding row r is found and the document frequency counter in the matrix cell at position (d, r) is incremented. After all documents have been processed, we traverse the matrix in inverted order. Row by row, we obtain an inverted list by compressing the row entries and write the inverted list for the corresponding term to disk.

If we allow only 2 bytes to store a document frequency (which is enough for the small Bible collection since no index term occurs in a document more than $2^{16} - 1$ times), the inversion matrix for the 4.4 Mb Bible collection would occupy more than 800 Mb of main memory. The larger Web collections need 4 bytes for a document frequency, so more than 10 Tb of main memory would be required for the inversion matrix of collection *Web V* and almost 89 Tb for collection *Web XX*. However, since the distribution of terms in text is skewed, rows are only sparsely populated on average. For example, a document of collection *Web V* contains on average 151 distinct terms, accounting for less than 1% of the vocabulary size.

Space savings can be achieved by storing a row of the matrix as a linked list that grows dynamically, yielding a list-based in-memory inversion approach (Bertino, Ooi, Sacks-Davis, Tan, Zobel, Shidlovsky & Catania 1997, Moffat & Bell 1995, Witten et al. 1999). Each list node represents a $(d, f_{d,t})$ posting and comprises 12 bytes since, apart from the 8 bytes of the posting, a 4-byte pointer that refers to the next list node is also kept. With the list representation of a matrix row less than 8 Mb are needed for the Bible. However, around 1.6 Gb is required for collection *Web V*, and around 6 Gb for collection *Web XX*. In practical systems, word positions are essential, and thus even more space would be required.

Where maintaining the linked lists requires more main memory than is available, it is tempting to rely on virtual memory and to let the operating system map the lists to disk. Accesses to the lists respectively rows are, however, in random order. Thus, if only every tenth access requires a random disk access, taking around 9 milliseconds, more than 35 hours would be needed for random disk accesses during the inversion of collection *Web V*, for example.

We assume that the input to the inversion approaches consists of a stream of $(t, d, f_{d,t})$ postings or $(t, d, f_{d,t} : l_1, \cdots, l_{f_{d,t}})$ in case word positions are included. These postings are delivered by a process that parses the collection document by document into index terms. For each document, its index terms, their word positions and within-document frequencies are gathered. We have identified efficient data structures for per-document vocabulary accumulation in previous work (Heinz & Zobel 2002).

Disk-based inversion

The main idea of the single-pass inversion approach by Harman & Candela (1990) is to accumulate postings in a posting file on disk and not in main memory. Postings that belong to the same term are linked together in the file, which consists of multiple linked lists, one for each index term. After all postings have been accumulated, a new file is allocated on disk to accommodate the final inverted lists and the lists are processed in lexicographical order. Each list is traversed, then the retrieved postings are compressed and written to disk as a compressed inverted list.

For efficiency reasons, it is necessary to maintain for each index term the file address of its previous posting in an in-memory lexicon. For each posting $(t, d, f_{d,t})$ from the parsing stream, the lexicon is queried for term t. The file address p of the terms' previous posting is retrieved and a new posting $(d, f_{d,t}, p)$ is appended to the posting file. Then the lexicon entry for t is updated. However, the postings for a term are not stored adjacently in the posting file and the lists are not stored in lexicographical order, as is required in the final inverted file. Hence, after all documents have been processed, a post-processing step is needed in which all lists in the posting file are traversed.

The post-processing step starts by allocating a second file on disk, then a lexicon traversal outputs the index terms in lexicographical order. Since for each index term the file address of its last posting is maintained in the lexicon and each posting is linked to its predecessor, the postings for a term are retrieved in reverse order and accumulated. Then, the inverted list for the term is computed and appended to the second file. The approach can straightforwardly be generalised to include word positions in the postings (Rogers et al. 1995).

Compared to the simple in-memory inversion schemes, the disk-based approach uses only one pass over the collection, but still requires that a lexicon be kept in main memory. If the lexicon is implemented as splay tree, each tree node uses 4 bytes for storing the file address (and even more bytes if the posting file exceeds 2 Gb), at least 2 bytes for f_t, 8 bytes for the two references to other tree nodes, and 4 bytes for a term pointer. Assuming that storing a term needs on average a further 8 bytes, the lexicon occupies around 74 Mb for collection *Web V* and 170 Mb for collection *Web XX*.

On the other hand, the savings in main memory come at the expense of large volumes of temporary disk space. A posting occupies 4 bytes for the document number, 2 bytes for the $f_{d,t}$ components, and 4 bytes for the reference to the previous posting of the term. Thus, a posting comprises 10 bytes and the posting files for collection *Web V* and collection *Web XX* exceed 1.3 Gb and 5 Gb. This space cannot be released until the end of the post-processing step as it is not possible to write inverted lists back into the posting file. Additional temporary space is needed if word positions are indexed as well, adding say 6 bytes per posting on average. More seriously, accesses to the posting file are in random order during the post-processing step. Due to the large number of random disk accesses required, the running times reported are not practical.

For example, index construction for a 276 Mb collection of Wall Street Journal articles is reported to have taken more than 2 days (Rogers et al. 1995); the Sparcstation 2 used in that instance is outperformed by current workstations, but disk characteristics, the main bottleneck, have only moderately improved since the work was undertaken.

To improve performance, Rogers et al. (1995) propose to reduce the number of random disk accesses by employing several posting files each of which is small enough to be fully mapped into main memory during the post-processing step. Each posting file corresponds to a disjunctive range of terms and accumulates only postings of terms that fall within that range. The term ranges are chosen to produce evenly-sized posting files. However, the division into separate posting files increases disk activity during the pass over the collection and the keeping of data on disk reduces opportunities for optimising disk accesses. Reducing the size of the posting file by applying compression techniques to the postings is also difficult, since each node must be individually accessible during the post-processing step.

In terms of the performance figures and parameters given in Tables 1 and 2, the predicted inversion time of the disk-based inversion approach is

$$\begin{aligned} T = \quad & St_t + C(t_p + t_l) + 10Pt_t + && \text{(read, parse, lookup lexicon, write postings)} \\ & Pt_s/v + 10Pt_t + && \text{(traverse lists)} \\ & I(t_c + t_t) && \text{(compress, write out inverted file)} \end{aligned} \qquad (1)$$

seconds. In the event that the size of the posting file exceeds available main memory size, that is, $M < 10P - L$, a complete mapping of the posting file to disk is impossible. We therefore denote the number of postings that are retrieved within one random disk access with v.

For inversion of collection *Web V*, around 9 minutes are spent reading and parsing the collection into index terms. A further 3 minutes are required for looking-up the lexicon. Writing the postings to disk and writing out the final inverted lists takes together less than 4 minutes. The major component of the overall running time is the traversal time for the posting file when mapping is impossible. For example, if only half of the posting file can be mapped into memory and hence $v = 2$, around 7 days are spent seeking postings on disk yielding a total predicted running time of around 7.3 days to construct a document-level inverted file for collection *Web V* and 28.3 days for collection *Web XX*. For small collections that can be fully mapped into memory, however, the disk-based inversion approach is suitable, since no random disk accesses are required during the traversal of the lists. The posting file for the Bible, for example, comprises less than 6 Mb and the predicted running time for the inversion is around 3 seconds. Including word positions in the index leads to larger posting files but the overall increase in predicted running time due to coding word positions is small.

Two-pass in-memory inversion

A drawback of the disk-based inversion approach is that large volumes of temporary space are needed during index construction. Fox & Lee (1991) propose an in-memory inversion approach that uses compression to limit temporary disk space usage. The approach is discussed in detail elsewhere (Frakes & Baeza-Yates 1992) and improvements are proposed by Moffat & Bell (1995) to make the approach applicable to collections of any size. This approach uses two passes over the input and we refer to it as the *two-pass* approach in this paper.

In the first pass, the vocabulary and statistics of the text collection are gathered. The number of documents N, the number of distinct index terms n, and the within-collection frequency f_t for each index term in the collection are determined and the vocabulary is stored in an in-memory lexicon. Prior to the second pass over the collection, an in-memory bitvector is allocated to store postings. The vector is divided into n partitions with one partition per index term; since the number of postings per term was determined during the first pass over the collection, the boundaries of the individual partitions and the overall size of the in-memory bitvector are easily pre-calculated based on the gathered statistics. During the second pass, the partitions are filled up with corresponding postings.

In comparison to the disk-based inversion approach, space is saved because postings that belong to the same index term do not have to be linked together, since they are adjacent in a partition. However, without compression, storing the postings for collection *Web V* and collection *Web XX* would require around 1 Gb or 4 Gb of main memory respectively, which is too large in practice.

A further reduction in main memory usage is possible when the first pass over the text collection also determines the maximum number of bits that are needed to store the postings for each term directly as a compressed inverted list in a partition. The exact number of bits required depends on the actual document numbers but these cannot be gathered during the first pass. Since a partition is not easily extended if the compressed document numbers do not fit into the pre-allocated space, a strict upper bound must be calculated. For Golomb coding, choosing parameter $b = 2^{\lfloor \log_2((N-f_t)/f_t) \rfloor}$ leads to an upper bound on the size of the codes (Witten et al. 1999) and guarantees that inverted lists do not overflow their corresponding pre-allocated partitions. In practice, the compression leakage is reported to be around 5% of the compressed inverted file size (Witten et al. 1999).

The predicted running time to build an inverted file with the two-pass approach is, in terms of the parameters used in Tables 1 and in 2,

$$\begin{aligned} T = \quad & St_t + C(t_p + t_l) + && \text{(first pass to gather statistics)} \\ & St_t + C(t_p + t_l) + \bar{I}t_c + && \text{(second pass, in- memory compression)} \\ & \bar{I}t_c + I(t_c + t_t) && \text{(decompress, recompress, write out inverted file)} \end{aligned} \tag{2}$$

seconds, where $\bar{I} \approx 1.05\ I$. For constructing a word-level inverted file for a collection, the compression leakage is likely to be much less than 5% when the same non-parameterised coding scheme for the word positions is used in the in-memory vector as well as in the final inverted file.

The assumption underlying this approach is that main memory is large enough to accommodate the pre-allocated vector of size $\bar{I}$ together with the lexicon of size L and hence $M \geq L + \bar{I}$. For $M =$ 256 Mb, we assume that the lexicon and the vector for the two smaller reference collections can be accommodated in main memory. The first pass over collection *Web V* takes 12 minutes. During the second pass, the collection is read again taking an additional 12 minutes. The in-memory compression of the postings and the decompression of the vector to obtain the inverted lists requires a little over 3 minutes. Writing the inverted lists to disk and encoding them on the fly requires around 2 minutes. The total predicted running time to construct a document-level inverted file for collection *Web V* is 29 minutes, of which the bulk of the time is spent parsing the collection. For the inversion of the Bible collection, we calculate a predicted running time of less than 4 seconds. However, due to the in-memory vector size of around 730 Mb that would be needed, collection *Web XX* cannot be indexed within 256 Mb of main memory.

A way to improve the scalability of the two-pass approach is to pre-allocate the partitioned vector not in main memory but on disk and to subdivide the text into several loads (Moffat & Bell 1995, Witten et al. 1999). The size of each load is chosen to be small enough to invert in main memory as described above. When main memory is exhausted, the current load is merged into the disk vector. The disk vector is read sequentially, that is block by block, into main memory and the blocks are written back in-place after new entries from the current load have been added. After all loads have been processed, the unused space between inverted lists is compacted in a final sequential pass over the file. In the last step, each inverted list is also decompressed and recompressed using the final compression scheme of the inverted file.

For this text-partitioned scheme, more information has to be gathered during the first pass over the text collection than is needed for the simple two-pass approach. For each load, the number of documents and within-load frequencies of the terms are determined. Based on these statistics, the in-memory vectors of the loads are divided into partitions and the partition boundaries are calculated. The statistics of each load are stored compressed in a description file on disk requiring only a few megabytes each.

The predicted running time to build an inverted file with the improved two-pass approach in terms of the parameters in Tables 1 and 2 is

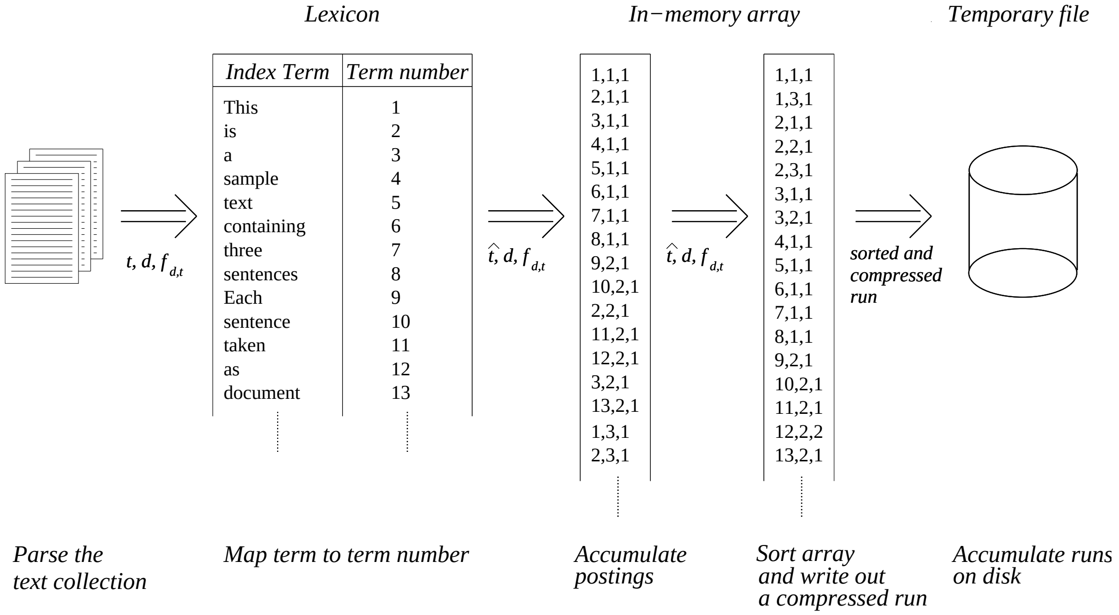
$$\begin{aligned} T = \quad & St_t + C(t_p + t_l) + && \text{(first pass to gather statistics)} \\ & St_t + C(t_p + t_l) + 3\bar{I}t_c + 2l\bar{I}(t_s/b + t_t) + && \text{(second pass, invert in-place)} \\ & (I + \bar{I})(t_s/b + t_t + t_c) && \text{(compact inverted file)} \end{aligned} \qquad (3)$$

seconds, where l denotes the number of loads into which the text is divided and b denotes the size of the data blocks that are transferred within one disk access. The number of loads is computed by $l = \lceil \bar{I}/(M - L) \rceil$.

In order to estimate the running time to build a document-level inverted file for collection *Web XX*, we make the simplifying assumption that main memory is used only to invert each load, thus ignoring memory consumed by the lexicon. Three loads are processed for collection *Web XX*. The first pass over the collection takes a little over 47 minutes. The second pass also requires around 47 minutes to read, parse, and lookup the terms. Each load is processed in around 11 minutes. Compacting the inverted file in the last step of the algorithm takes 17 minutes, assuming that an input buffer of at least $b = 64$ Kb is available. Thus the total predicted inversion time is around 144 minutes for collection *Web XX*.

Sort-based in-situ inversion

Moffat & Bell (1995) propose a sort-based inversion approach to inverted file construction that is applicable to text collections of any size. This single-pass approach is intended to operate with limited amounts of main memory and does not require large temporary disk space. In this paper, it



Index Term	Term number
This	1
is	2
a	3
sample	4
text	5
containing	6
three	7
sentences	8
Each	9
sentence	10
taken	11
as	12
document	13

Figure 2: *Constructing sorted and compressed runs during the sort-based approach.*

is referred to as the *sort-based* approach. It operates as follows. First, the available main memory is filled with postings from the incoming parsing stream. Each time a main memory threshold is reached, the postings are sorted and compressed, and this *run* is written to a temporary file. These two steps as depicted in Figure 2 are repeated as long as there are postings delivered from the parsing process. Then, in the last step, the runs on disk are combined by a multi-way merge and the final inverted lists are computed. A careful *in-situ* merge allows writing of the inverted lists gathered during the merge back into the same file. In memory, efficient coding of the term components in the postings is achieved by replacing the string representation of a term t by a unique term number $\hat{t}$; thus the lexicon maintains for each index term a unique term number. The assignment of integers to terms is not based on the lexicographical order of the terms but on their order of first appearance. The form of the postings is $(\hat{t}, d, f_{d,t})$, and each is 12 bytes.

When main memory is full, prior to processing postings from the next document, the postings are sorted in ascending order of their term number as the primary key and then ascending by their document number as the secondary key. The run can then be compressed as it is written to disk using gap coding. Note that the number of postings accumulated in a run is not fixed since the main memory is shared between the array and the lexicon, which grows with the number of distinct terms parsed from the collection.

When all the documents have been processed, the K runs that have been accumulated on disk are merged to obtain the final inverted list for each index term. A K-way merge requires only one merging pass over the runs. To avoid excessive disk costs, an in-memory input buffer is assigned to each of the K runs. A request for the next compressed inverted list from a run checks the corresponding input buffer first. If the data in it has been consumed, the next chunk of data from the run is read from disk into the input buffer and processed. Thus, filling a buffer requires only one random access to disk followed by several sequential accesses, depending on the block size of the disk and the input buffer size u. The optimal size u of an input buffer depends on the available main memory, the number of runs, the size of inverted lists, and on the disk characteristics such as block size and transfer rate.

During merging, we have to find repeatedly among the K runs the smallest temporary inverted lists, that is, the list with the smallest term number that starts with the lowest document number. To find efficiently the smallest list, a priority queue such as a heap is maintained to retrieve the smallest list at a cost of $\lceil \log_2 K \rceil$ comparisons and swaps (Sedgewick 1998). The lists are then decompressed, recompressed, and merged into the final inverted list for the current term number.

The merged lists can straightforwardly be written to a new file. However, the sort-based approach would then need temporary disk space more than twice the size of the final inverted file. Moffat & Bell (1995) describe a technique to write the final inverted lists back in-place into the temporary file. At the start of the K-way merge, each of the input buffers is loaded with a block from the temporary file. Thus, the first K output blocks that are accumulated during the merging process can be written back into these blocks. At that stage, at least one of the K runs must have already read its second block into its input buffer, making one more page available. Due to the more efficient coding scheme used in the final inverted lists and because of omission of $\hat{t}$-gaps, the output stream during merging is likely to be smaller than the input stream. In rare cases, an output block might not find a vacant block on disk due to switching from a delta-code or gamma-code to Golomb coding of the final inverted lists. To resolve this problem, an output block is simply appended at the end of the temporary file.

After all runs have been merged, the block table is used to permute the blocks to ensure that the physical order of the inverted lists corresponds to their logical order of the term numbers. The in-place permutation process requires an in-memory buffer of two blocks of b bytes each. Each block is read and written exactly once without any additional coding.

The predicted running time of the sort-based approach to construct a document-level inverted file is, in terms of the parameters in Tables 1 and 2,

$$\begin{aligned} T = \quad & St_t + C(t_p + t_l) + && \text{(read, parse)} \\ & K(1.2k \log_2 k)t_w + \bar{I}(t_t + t_c) + && \text{(sort, compress, write runs)} \\ & \bar{P}\lceil \log_2 K \rceil t_w + (\bar{I} + I)(t_s/u + t_t + t_c) + && \text{(merge, recompress)} \\ & 2\bar{I}(t_s/b + t_t) && \text{(permute)} \end{aligned} \tag{4}$$

seconds, where K is the number of runs that are processed, k is the number of postings that can be kept in main memory, u is the size of the input buffer assigned to each run during merging, and the block size during the merging phase is b. We calculate $k = (M - L)/12$, where L denotes the size of the lexicon, and derive $K = \lceil P/k \rceil$. The size of the temporary inverted file is denoted by $\bar{I}$ and $\bar{P}$ is the number of tuples that are inserted into the heap.

Based on the results of preliminary experiments, we assume here that $\bar{I} \approx 1.2\ I$, since the runs are compressed with a less efficient non-parameterised coding scheme compared to the coding scheme used in the final inverted file. However, larger runs due to having more memory yield smaller $\bar{I}$ since the $\hat{t}$-gaps are likely to become smaller on average. It is difficult to calculate $\bar{P}$ exactly, since it depends on M and the characteristics of the particular text collection. For a skewed distribution of terms, for example, it is likely that frequent terms occur more than one run. In that case, fewer than P tuples are inserted into the heap. However, each distinct term number appears only once in each of the K runs. Thus, we have the upper bound $\bar{P} \leq nK$.

According to Formula 4, the predicted running time for collection *Web XX* is around 105 minutes, ignoring as before memory requirements for the lexicon. The predicted time is approximately 47 minutes for reading, parsing, and querying the lexicon and a little over 27 minutes for sorting the 25 runs. Compressing and writing out the runs account for around 9 minutes. Managing the heap takes about 2 minutes. Around 18 minutes are spent on merging the temporary inverted lists including decompression and recompression time for each list. The block permutation at the end requires less than 2 minutes. Similar relative cost figures are calculated for the other reference collections. The predicted running time for collection *Web V* is about 28 minutes and the inversion of the Bible is performed hypothetically in less than 5 seconds.

As with the other approaches, the inclusion of word positions during index construction leads to only small modifications to the algorithm. The main modification is to include in each posting a reference to an array of word positions, enlarging a posting by a 4-byte pointer p. Hence, the form of a posting becomes $(\hat{t}, d, f_{d,t}, p)$ using 16 bytes. The array that is referred to by the pointer p contains $f_{d,t}$ word positions and comprises additionally $4f_{d,t}$ bytes. The word positions are compressed when a run is written out to disk.

If a posting contains only one word position, space is saved by storing the single word position in the memory space that is allocated for the pointer p. In preliminary experiments, we calculated the average byte size of a posting plus its corresponding array of word positions. We parsed collection *Web V* into $(\hat{t}, d, f_{d,t}, p)$ postings and accumulated them in main memory. For postings where $f_{d,t} > 1$, we additionally allocated an array of $4f_{d,t}$ bytes to store the word positions of term t in document d. The accumulation of postings stopped when a memory threshold of 100 Mb was reached. Over 30 runs with pair-wise disjunctive sets of documents parsed from the collection *Web V*, we observed that 1.61 array slots were used per posting on average with a standard deviation in the runs of 0.21. We conclude that the average size of a posting and its associated array of word positions is around 22 bytes. We therefore expect that the number of runs generated by the sort-based approach would increase by approximately 80% if we consider building a word-level inverted file instead of a document-level inverted file for a fixed amount of main memory. Hence, by calculating $k = (M - L)/22$ and setting $I = I_w$, we can apply Formula 4 to calculate the predicted running time to build a word-level inverted file with the sort-based approach.

Comparison of inversion approaches

The simple inversion approaches accumulate postings in main memory but use no compression. They are suitable for small collections when main memory is large enough to accumulate all postings in the form of an inversion matrix or dynamic linked lists.

The disk-based approach by Harman & Candela (1990) needs merely to maintain a lexicon in main memory since postings accumulate on disk. However, retrieval of the postings in the second stage is a performance bottleneck, as there are too many random disk accesses. Efficient compression cannot be applied to the postings, so temporary disk space is typically 50%–100% of collection size. Thus, the disk-based approach is not scalable and not suitable for large collections.

The sort-based approach and the two-pass approaches write accumulated index data in a compressed format to disk, yielding low temporary space overheads. They can both operate within limited amounts of main memory for postings but require the entire vocabulary of the collection in memory. In the sort-based approach, an in-memory array is used to accumulate the postings. When memory runs out, the sorting step ensures that postings that belong to the same term are adjacent. Mapping terms to term numbers has two advantages. First, sorting requires only integer comparisons. Second, main memory is utilised more efficiently, since using a 4-byte integer for a term number in each postings is more space-efficient than storing the string representations. We observed in experiments on the Web collections that the size of the compressed runs is typically around 15%-20% larger than the size of the final inverted file if 100 Mb of main memory is used to accumulate postings. The leakage is caused by the different coding schemes that are used in the compressed runs and the final inverted file. We also noticed that, with increase in the size of the runs, the leakage becomes smaller.

A disadvantage of the sort-based approach is that it constructs inverted files that do not support range queries well, due to the order in which the inverted lists are stored in the final inverted file. If the lists, however, were stored according to the lexicographical order of the terms, as it is done by the two-pass approach, only one random access is needed to locate the first inverted list and the subsequent lists can be retrieved by sequential accesses.

A smaller compression leakage is reported for the two-pass approach (Witten et al. 1999). In contrast to the sort-based approach, postings are stored compressed as soon as they are accumulated in main memory. However, the compression leakage comes at the cost of an additional first pass over the collection. Performing a pass over a collection has three major cost factors: transfer costs, costs for decompressing in case the collection is compressed, and parsing costs. Temporary disk space can be traded against parsing time by storing the output of the first parsing process and to use the parsed version of the text in subsequent passes. Hence, in that case, only transfer costs and compression costs are incurred during subsequent passes.

The predicted running times of the approaches are presented in Table 3 together with the predictions for our single-pass in-memory inversion approach, which we discuss in the following section. As shown, the sort-based approach outperforms the two-pass approach in terms of predicted running times. Whereas the difference between the two approaches with regard to their predicted running time is only small for constructing document-level inverted files, the margin increases when we compare their performance in constructing word-level inverted files. The advantage of the two-pass approach is that it requires less temporary disk space than the sort-based approach during the construction of document-level inverted files. However, the opposite is true if we consider building word-level inverted files, where the two-pass approach needs slightly more temporary disk space because of the descriptions files of the loads.

The sort-based approach and the two-pass approach allow a trade-off between main memory and running time. Greater availability of main memory, for example, yields fewer runs and loads. It is therefore interesting to consider the performance of inversion approaches when the available main memory size is varied. Even for large volumes of main memory, the sort-based approach

Table 3: *Predicted running times in minutes of inversion approaches to construct inverted files on the sample Web collections with the statistics as shown in Table 2. The predicted size in megabytes of the peak amount of temporary disk space is given in parentheses next to the predicted running time of each run. The volumes of disk space do not include the final inverted file sizes. The times are calculated in terms of the parameters given in Table 1 and Table 2. 256 Mb of main memory is assumed to be available.*

	Web V	Web XX
Constructing a document-level inverted file		
Disk-based approach	10,561 (1,341)	40,803 (5,180)
Two-pass approach	29 (10)	144 (43)
Sort-based approach	28 (66)	105 (140)
Single-pass in-memory approach	23 (59)	91 (230)
Constructing a word-level inverted file		
Disk-based approach	10,567 (2,145)	40,826 (8,288)
Two-pass approach	73 (41)	396 (158)
Sort-based approach	41 (34)	162 (130)
Single-pass in-memory approach	37 (61)	143 (234)

outperforms the two-pass approach by a margin of around 40% as shown on the left hand side of Figure 3. For constructing a word-level inverted file, the performance gap is much wider. When main memory is limited, the predicted running time of the two-pass approach increases dramatically as shown on the right hand side of Figure 3, whereas the running times of the sort-based approach do not vary much, and increase only slightly for very limited volumes of main memory. Note that Figure 3 also depicts the performance of the single-pass in-memory inversion approach that we propose later.

It is also interesting in that context to consider the performance of the algorithms for asymptotically large text collections. If we assume, for the sake of simplicity, that S, C, and I grow as P grows, and L remains constant, then the predicted running time of two-pass approach is limited by $C(P^2/M + P)$ whereas the sort-based approach is bound by $C(P^2/M^2 + P/M \log P)$ with small constant factors preceding the P^2 factors. For $P \rightarrow \infty$, the sort-based approach is more efficient, although sorting is required. Due to its scalability, its good asymptotic behaviour in theory, and its good performance in practice, the sort-based approach is used in several approaches to distributed inverted file construction (Melnik, Raghavan, Yang & Garcia-Molina 2001, Ribeiro-Neto, de Moura, Neubert & Ziviani 1999).

However, the sort-based approach requires, as do the other approaches that we have reviewed, the maintenance of an in-memory lexicon. Since the number of index terms increases with collection size more or less linearly (Williams & Zobel 2001), the size of the lexicon increases with the number of documents processed. Thus, if main memory is fixed and storage space for the lexicon increases, less memory becomes available for accumulating postings in main memory. Given a skewed distribution of text, however, a large number of index terms are never accessed again after they have been inserted into the lexicon. Approaches such as the sort-based approach or the two-pass approach suffer from this inefficient memory use, since with the increasing number

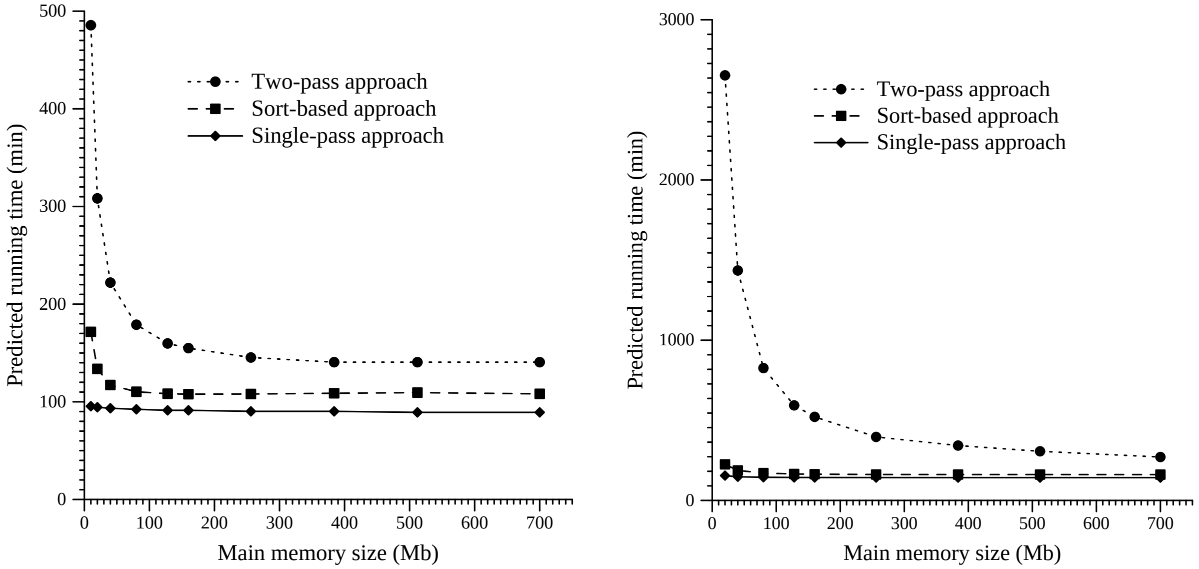


Figure 3: *Predicted running times in minutes to construct a document-level inverted file on the left and a word-level inverted file on the right hand side for collection* Web XX *with varying main memory sizes, for the inversion approaches.*

of runs or loads, overall running time increases too. In the next section, we propose an inversion approach that does not rely on maintaining the vocabulary of the text collection in main memory.

Witten et al. (1999) also review several inversion approaches including the sort-based approach and the two-pass approach. Based on a similar computational model, Witten et al. (1999) predict that the sort-based approach outperforms the two-pass approach in practice, confirming our theoretical results. However, the focus there is solely on the construction of document-level inverted files and in contrast to our work word-level inversions are not discussed nor are empirical results presented. A further difference to the work by Witten et al. (1999) is that we have considered the performance of the inversion approaches on a range of reference collections for varying volumes of available main memory.

4 Single-pass in-memory inversion

The single-pass in-memory approach that we propose uses only one pass over the collections, while requiring only small amounts of temporary disk space due to an efficient in-memory compression scheme. Although it shares features with other approaches that we have discussed, there has not been any publication of a thorough discussion nor of any experimental results regarding this approach to date. We refer to it as the *single-pass* approach in this paper.

The basic idea of the single-pass approach is to assign to each index term in the lexicon a dynamic in-memory bitvector that accumulates their corresponding postings in a compressed format. Since no statistics of the index terms are known, only non-parameterised coding schemes can be employed. We use a coding scheme identical to that employed for the compression of sorted runs in the sort-based approach, that is, Elias codes represent the d-gaps, word-position gaps, and document frequencies. The gap coding of document numbers requires that the last-inserted document number is known when the next posting is inserted into a bitvector, so we keep for each term the last document number that has been inserted into it as an uncompressed integer. In contrast to the compression scheme used in the sort-based approach, no representation of a term is stored in a bitvector and no mapping of terms to term numbers takes place.

With the use of these bitvectors, the main steps of the single-pass approach are as follows. An empty temporary file is allocated on disk. For each posting delivered from the parsing stream, a search for its corresponding term is made in the lexicon. If the search is unsuccessful, the term is inserted into the lexicon and the corresponding bitvector is initialised. The posting is inserted into the bitvector and compressed on the fly. This process is iterated as long as main memory is available. When main memory is used up, the index terms and their bitvectors are processed in lexicographical term order. Each index term is appended to a temporary file on disk and front coding is used to represent the terms efficiently. Each bitvector is padded, that is, aligned to the start of the next byte, and appended to the temporary file. The lexicon is freed and the process repeats until all documents have been processed. When all documents have been processed, the compressed runs are merged to obtain the final inverted file. A similar approach was used for the *CAFE* genomic retrieval system (Williams 1998).

Merging is as in the sort-based approach: that is, a multi-way merge is applied to the compressed runs to give the final document-sorted inverted file. We can employ the *in-situ* variant of the multi-way merge, so we have to permute disk blocks at the end of the merge. Since the inverted lists are merged in lexicographical order, the inverted file stores, after the permutation of blocks, all inverted lists in that order. Thus, range queries are supported efficiently since inverted lists that belong to a range of terms in lexicographical order are stored contiguously.

An advantage of the single-pass approach is that we avoid keeping index terms permanently in memory, in particular those that are never accessed after they have been inserted into the lexicon, since we flush out the lexicon when main memory is exhausted. On the other hand, assigning a bitvector to each index term in the lexicon increases the size of the lexicon. In our implementation, a bitvector requires 32 bytes to keep some associated counters and variables and, on initialisation of a bitvector, further bytes are dynamically allocated for its byte array. On an overflow of a byte array, the array size is dynamically doubled. Hence, for terms that occur only once during a run, the scheme uses more main memory to store the single posting of 12 bytes than the sort-based approach does. However, if a term occurs frequently during a run, storing its postings compressed in a bitvector is more efficient than accumulating the postings in an array. It is difficult to predict the space savings in practice, since we have to take into account main memory size and the skewed distribution of terms, but we surmise that the single-pass approach performs fewer runs than the sort-based approach.

A disadvantage of the single-pass approach is that we have to include index terms in the runs. However, since the terms are processed in lexicographical order, adjacent terms are likely to have a common prefix, and a front-coding scheme can be used to reduce storage space. For a large number of sorted terms, adjacent terms typically share a prefix of 3 to 5 characters and only 1 byte is needed to store that value. However, front-coding of a terms is less space-efficient than using coded term numbers as in the sort-based approach. Hence, transfer costs increase and more temporary disk space is used by the compressed runs compared to the runs of the sort-based approach.

We have to process the index terms in lexicographical order when we write a compressed run to disk, otherwise merging of temporary inverted lists is not possible. If we implement the lexicon with a sorted data structure, such as a binary search tree or a splay tree, an in-order traversal processes the terms and their bitvectors in lexicographical order and no sorting is needed. However, these tree structures are typically slower than hash tables, where term lookups are fast but there are sorting costs when a run is written out.

The predicted running time to construct an inverted file with the single-pass approach is, in terms of the parameters in Table 1 and 2,

$$\begin{aligned} T = \quad & St_t + C(t_p + \bar{t_l}) + && \text{(read, parse)} \\ & \bar{I}(t_c + t_t) + && \text{(in-memory compression, write)} \\ & \bar{P}\lceil \log_2 K \rceil t_w + (\bar{I} + I)(t_s/u + t_t + t_c) + && \text{(merge, recompress)} \\ & 2\bar{I}(t_s/b + t_t) && \text{(permute)} \end{aligned} \tag{5}$$

seconds, where $\bar{I}$ is the size of the temporary inverted file, K is the number of runs, u with $u < M/K$ is the size of each input buffer assigned to each run, and $\bar{P}$ denotes the number of tuples inserted into the heap. We calculate the number of runs as $K = \bar{I}/(M - \bar{L})$, where $\bar{L}$ denotes the size of a lexicon during a run. We assume as before that $\bar{P} \leq nK$ and $u = b = 64$ Kb. We further estimate a slower lexicon lookup time per term, that is, $\bar{t_l} = 1.5t_l$, since we assume that the lexicon is implemented as a data structure that keeps terms in lexicographical order, which offers slower lookup times compared to a fast hash table implementation.

Based on some preliminary experiments, we further assume that $\bar{I} \approx 1.33\ I$ for constructing a document-level inverted file and $\bar{I} \approx 1.09\ I_w$ in the case of considering the inclusion of word positions in the inverted file. The compression leakage is larger compared to that achieved by the sort-based and the two-pass approach, due to the front-coded terms included in the runs.

The predicted running time for the single-pass approach to build a document-level inverted file for collection *Web V* is around 23 minutes and around 91 minutes are required for the inversion of collection *Web XX*. The predicted running time for the document-level inversion of collection *Web XX* is as follows. Around 54 minutes are spent on reading, parsing, and looking up the index terms in the lexicon. The in-memory compression of postings in the bitvectors accounts for a further 9 minutes. Around 2 minutes are spent on comparing strings and performing swap operations in the string heap. Merging the inverted lists from the 4 runs including the decompression and recompression time takes approximately 20 minutes. The time taken to permute the blocks accounts for another 6 minutes.

Table 3 presents the predicted running times for the single-pass approach on the reference Web collections. The single-pass approach outperforms the other approaches in terms of speed for all main memory limits and its performance only slightly depends on the volumes of available memory, as shown in Figure 3.

The asymptotic costs of the single-pass approach for $P \to \infty$ under the simplistic assumption that S, C, and I grow as P grows, and L remains constant, are $C(P/M + P/M \log(P/M))$ with a small factor in front of P. Thus, the single-pass approach is also asymptotically more efficient than the sort-based and the two-pass approach for fixed M.

Implementation

To implement single-pass inversion efficiently, first we have to consider which in-memory data structure to use for the accumulation of index terms and their statistics. All practical inversion approaches require an in-memory lexicon to maintain index terms and their corresponding statistics such as frequency counters and postings. The overall performance of an inversion approach is therefore dependent on the performance of the underlying lexicon data structure. This is also evident in the formulae predicting running times; t_l, the time to lookup an index term in a lexicon, is a major cost factor. Apart from speed, the memory requirements of a lexicon also affect the overall performance of inversion approaches. In general, the more memory is used by the lexicon, the more runs have to be generated during the inversion of a collection.

In other work we explored and developed a variety of data structures that can be used for this task (Heinz & Zobel 2002, Heinz, Zobel & Williams 2002, Zobel et al. 2001). We found that the fastest approach is an appropriately implemented hash table; conventional tries are slower and use six times as much space, while trees are both slightly larger and several times slower. The only lexicographically-sorted structure that approaches the hash table in performance is one we developed in the course of this work, called the burst trie, a variant of trie in which leaves are small buckets or containers of strings that share a common prefix. By replacing a bucket with a trie node and a collection of child buckets whenever it becomes too large (say 100 strings), the burst trie can grow smoothly and provides fast access: common, short strings are entirely represented in the trie, while longer strings, accessed more rarely, are kept in a space-efficient structure. For a large collection of strings, a burst trie is no larger than a hash table and only about 25% slower. For small collections of strings, the burst trie is the fastest of the many structures we have tested.

In contrast to burst tries, hash tables do not maintain index terms in sort order. Hence, hashing requires an additional sorting step before a run can be compressed and written out to disk. Although hashing outperforms sorted data structures for overall vocabulary accumulation (Heinz et al. 2002), these intermediate sorting steps decrease the performance of hashing, as observed in our experiments on per-document vocabulary accumulation (Heinz & Zobel 2002). In particular, if only small volumes of main memory are available and hence sorting occurs frequently, a sorted data structure such as the burst trie might be more suitable. Intermittent splay trees, where the tree is only reorganised at every kth access for some small fixed k, are another option, however we anticipate that they are slower than hash tables and burst tries given the outcome of our previous experiments (Heinz et al. 2002).

The performance of the single-pass approach also depends on the data structure employed for the task of per-document accumulation of index terms and their statistics. Per-document data is merged into the lexicon whenever a document has been fully parsed and processed. The corresponding index term of each posting is searched for in the lexicon, and new index pointers are inserted into the term's bitvector. In our previous work (Heinz et al. 2002) we found that burst tries are the fastest data structure for per-document vocabulary accumulation, and we use this technique in all our experiments on inverted file construction.

When postings are delivered in lexicographical order, implementing a lexicon with a burst trie has another beneficial effect: a stable access path during the merge of per-document data into the lexicon. When the vocabulary of a document is sufficiently large, it is likely that subsequent lexicon accesses terminate in the same subtrie or even in the same container. Hence, nodes that have been visited previously are likely to be kept in the CPU cache where access is fast. In contrast, the access path of an intermittent splay tree is not stable due to the tree reorganisations. CPU caching is also less effective when a hash table implements a lexicon, since terms are randomly distributed over slots.

For the sort-based approach, a key issue is how to accumulate postings in memory. We use an in-memory array whose size depends on the available main memory size. When a document-level inverted file is constructed, each array slot keeps a $(t, d, f_{d,t})$ posting. However, when a word-level inverted file is constructed, a different approach is required to accumulate variable-size $(t, d, f_{d,t} : l_1, \cdots, l_{f_{d,t}})$ postings. In our implementation, we store in each slot an additional pointer to a dynamic array to keep the $f_{d,t}$ word positions of a posting. Such a scheme is likely to suffer from high costs of invoking the system calls used for memory allocation and freeing. To reduce these costs, we allocate space for the word positions block-wise and maintain the blocks with a size of 512 Kb in a linked list. Further costs are saved if a posting contains only a single word position. In that case, the word position is stored in the storage space of the pointer itself.

The question of how to implement the lexicon for the sort-based approach deserves careful consideration. The lexicon's purpose is to map terms to term numbers. Moffat & Bell (1995) suggest using a hash table as a fast-lookup lexicon, since maintaining the terms in sort order is not necessary. We employ a hash table with move-to-front in chains, a table size of 2^{20} slots, and a bit-wise hash function. According to the results of our previous experiments on vocabulary accumulation (Heinz et al. 2002, Zobel et al. 2001), this hashing scheme is the best choice.

5 Experiments

In our experiments, we explore the performance of the single-pass and the sort-based approaches for the task of inverted file construction, considering the construction of document-level and word-level inverted files. However, we do not apply *in-situ* merging in either approach. It should be noted that this does not affect their relative performance. The purpose of our experiments is to identify the most efficient variant of the single-pass approach and compare it with the sort-based approach in terms of speed and temporary disk space usage. A further goal is to explore how both approaches perform and scale when available main memory is limited.

The test data used in these experiments are three Web collections, *Web V* and *Web XX* with the statistics shown in Table 2, and a third web collection *Web X*. *Web X* is 10 Gb, contains 1,780,983 documents with about 631 million term occurrences of 3,604,125 distinct terms. The elapsed parsing time for collection *Web V* is 788 seconds, for *Web X* 1,544 seconds, and for *Web XX* 3,088 seconds. In the experiments we report elapsed running times, which include any time spent waiting for disk accesses. The time taken to parse the collections into terms is included as well. All programs were implemented in C. We use a Pentium III 700 MHz with 512 Mb of main memory running a standard version of the Linux operating system. Prior to each experimental run, the internal buffers of our machine were flushed to ensure a non-biased test environment. We report experimental results over single runs as we have observed almost no variance between times over multiple runs.

One aim of our empirical comparison is to test the scalability of inversion approaches, that is, how they perform when only a limited amount of main memory is available. We therefore must keep track during inversion of how much memory is consumed. In general, we do not take into account memory usage of small support structures such as buffers and temporary structures used for sorting, for example. For the single-pass approach, we count the total size of the in-memory data structure including bitvectors. If this size reaches a given memory limit, a run is generated, the lexicon is traversed and freed, and the process continues as described earlier.

It is less straightforward, however, to decide how to count the memory usage of the sort-based approach. This approach maintains a hash table lexicon for mapping terms to term numbers. In addition, memory is consumed by the dynamic array that accumulates postings. The three main cost factors are therefore the lexicon, the array, and the postings (including the dynamic arrays to store word positions). Instead of restricting the total memory usage for the lexicon, the array, and the postings, we only limit the amount of main memory for the postings and do not take into account the additional memory usage of temporary arrays and other structures. Hence, in the case of constructing a document-level inverted file, runs contain the same number of postings for a fixed memory limit. A similar approach to limit the total main memory for the sort-based approach is considered elsewhere (Moffat & Bell 1995). For constructing a word-level inverted file with the sort-based approach, we also take into account the memory requirements of the word positions. As a result, individual runs usually do not contain the same number of postings. Alternatively, we could additionally take into account the storage cost of the lexicon. However, the lexicon grows over time and in extreme cases, the lexicon alone may eventually approach the given memory limit leaving no memory left over for the postings. Choosing a specific memory limit for the lexicon may discriminate against this method, so we ignore its requirements but note that many more runs would be required in practice.

To ensure a fair comparison of the sort-based and the single-pass approach, we use the same coding scheme for the final document-level inverted files: d-gaps are Golomb-coded, whereas gamma-codes are used for an efficient representation of $f_{d,t}$ and f_t frequencies. However, the intermediate runs generated by both approaches use slightly different compression schemes due to the requirement to store either term numbers or terms, respectively.

To facilitate disk accesses during the inversions, we use an in-memory buffer to write runs to disk and read blocks of 512 Kb from runs during the merge phase in all experiments. For small main memory limits and large text collections, it may be necessary to use smaller buffer sizes to accommodate a buffer for each run in main memory. Our experiments with a main memory limit of 40 Mb use a buffer size of 512 Kb, which yield a better result than if smaller buffers had been used to stay strictly within the main memory limit.

Results

In the first set of experiments, we measure the elapsed time and temporary disk space requirements of the single-pass approach. Three variants of this approach were tested. The first, uses as a lexicon a burst trie, the second uses an intermittent splay tree, and the third uses a hash table.

For single-pass inversion, the hash table was the fastest, but the speed margin was surprisingly small. For example, for a memory limit of 256 Mb on *Web X*, times were about 4,650 seconds for a hash table, 4,700 seconds for a burst trie, and 4,950 seconds for a splay trie. Detailed results for inversion with the hash table are shown in Table 4, discussed below.

Total disk requirements were extremely consistent between the methods, but dropped as memory limits were increased. With a 40 Mb memory limit, total disk usage was about 115% of the size of the final inverted file, falling to around 108% for a memory limit of 300 Mb. A smaller memory limit increases, in addition to the number of runs, the amount of storage space required to accommodate the runs on disk. The reason for this is twofold. First, each run contains the set of distinct index terms observed since the last run was written to disk. Hence, many index terms are included in multiple runs, increasing temporary storage space. In addition, if runs are short due to small volumes of available main memory, front-coding becomes less efficient and compression leakage increases. For these reasons, we observe that the amount of temporary disk space during the construction of a word-level inverted file with the single-pass approach accounts for around 8% of the size of the overall final inverted file, for main memory volumes well within the capabilities of modern workstations. For the construction of document-level inverted files, additional temporary disk space of approximately 26% of the final inverted file size is required, since front-coded index terms account for a larger part of the runs.

Another trend is that running times are largely independent of the volume of available main memory. Even 40 Mb gives good running times and performance improves only slightly with the amount of available main memory. The main reason why small memory limits yield higher running times is the increase in the number of runs generated, which leads to more random disk seeks and transfer costs during the merge phase. For a memory limit of 300 Mb, nine runs are generated by the single-pass approach for collection *Web X*. For a limit of 40 Mb, with a hash table lexicon 84 runs are constructed. Restricting memory usage from 300 Mb to 40 Mb increases inversion times by only around 2%.

Figure 4 shows the break-up of the running time for word-level inversion with a hash table lexicon on collection *Web V*. Typically around 35% of the total time is spent merging runs. Parsing the collection into index terms is also a major cost confirming that multiple-pass inversion approaches are not likely to perform well. The cost of sorting the hash tables prior to generating a sorted run amounts only to a tiny fraction of the total elapsed time. For a memory limit of 40 Mb, 43 runs are processed and for a limit of 300 Mb only four sorting steps are required for the construction of a word-level inverted file for collection *Web V*. These sorting costs affect the overall performance but a hash table still outperforms a burst trie by a small margin.

Table 4 presents the running times for the construction of document-level and word-level inverted files using both inversion techniques. For a fixed memory limit, the running times of the single-pass approach appear to increase linearly with collection size. On our standard machine

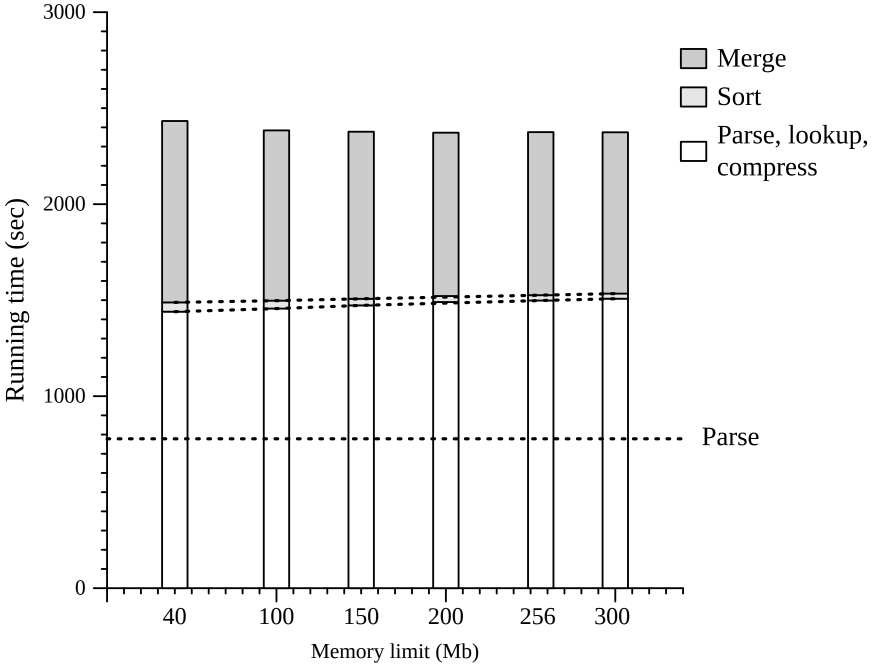


Figure 4: *Break-up of the total elapsed time in seconds for constructing a word-level inverted file for* Web V*, with the single-pass inversion approach and a hash table lexicon, for different main memory limits. The total elapsed time consists of the time to parse, lookup, and compress postings, the time spent sorting the hash tables to create a sorted run, and the time to merge the runs. The total elapsed time for the parsing process is approximately 778 seconds.*

using 256 Mb of main memory, it takes, on average, less than 5 minutes to construct a document-level inverted file for 1 Gb of text, and less than 8 minutes when word positions are included in the index.

When compared to the single-pass approach, the sort-based approach is significantly slower. The fastest running times of the sort-based approach are around 15% slower than the fastest running times of the single-pass approach for constructing a word-level inverted file. For building document-level inverted files, the performance gap is larger and the sort-based approach is typically around 20% slower. For the single-pass approach, the best times are observed with memory limits of 100–150 Mb; including lexicon size as explained below, for the sort-based approach the best times are observed when 106–198 Mb of memory is used.

An interesting trend in our experiments is that large volumes of available main memory do not decrease running times for the sort-based approach; rather, they affect the performance adversely. We surmise that the increase in running time is caused by the fact that longer runs must be sorted. (However, for memory limits significantly below 40 Mb, inversion times rise dramatically due to merging costs.) On the other hand, as fewer runs are generated, there is a corresponding decrease in merge time, but these savings do not compensate for the higher sorting costs, as Figure 5 illustrates. With a limit of 40 Mb, 77 runs are required for building a word-level inverted file for collection *Web V*, and 146 runs for collection *Web X*, but for 300 Mb, the number of runs

Table 4: *Elapsed time in seconds to construct inverted files with the single-pass and sort-based inversion approaches for the reference collections, with different main memory limits in megabytes. (Note that, for the sort-based method, the memory limits do not include the space required to store the vocabulary.) The times for constructing document-level inverted files are shown in each column to the left, the times for constructing word-level inverted files to the right.*

Memory Limit (Mb)	Web V	Web X	Web XX
Single-pass inversion			
40	1,510/2,433	2,970/4,786	5,961/ 9,625
100	1,475/2,384	2,893/4,668	5,782/ 9,370
150	1,479/2,377	2,898/4,650	5,763/ 9,307
256	1,483/2,375	2,930/4,659	5,811/ 9,291
300	1,484/2,374	2,925/4,656	5,833/ 9,328
Sort-based inversion			
40	1,791/2,702	3,496/5,291	6,993/10,695
100	1,820/2,726	3,558/5,328	7,097/10,710
150	1,837/2,739	3,592/5,361	7,158/10,749
256	1,884/2,759	3,727/5,411	7,722/11,021
300	2,041/2,805	4,088/5,500	8.955/12,048

is only 11 and 21, respectively; running times increase by around 4%. The running times for constructing document-level inverted files increase similarly with higher main memory limits.

As discussed earlier, the sort-based approach maintains in main memory a lexicon to map terms to term numbers, which grows with collection size. These additional memory requirements are not included in the main memory limits in our experiments. Around 66 Mb is required for the lexicon of collection *Web V*, approximately 82 Mb for collection *Web X*, and more than 148 Mb for *Web XX*. Hence, the sort-based approach achieves its fastest running times on collection *Web V*, not with 40 Mb, but effectively with around 106 Mb of main memory.

As shown in Figure 5, sorting is a major cost of the sort-based approach. We use the standard quicksort package under the Linux operating system to sort accumulated postings. It may be contended that a custom-built sorting algorithm or an optimised implementation of the quicksort algorithm could do better. We tried other sorting algorithms, and a fast quicksort implementation based on work by Bentley & McIlroy (1993), but achieved no gains in speed. However, even if two-thirds of the sorting costs could be saved—in our opinion an unlikely achievement—the sort-based approach would still be slower than the single-pass approach.

We also observe that, in common with the single-pass approach, large volumes of main memory reduce the additional temporary disk space needed to store the sorted runs, due to less compression leakage. In our experiments, the sort-based approach requires for the construction of a word-level inverted file around 5% more disk space than the final inverted file. The size of the temporary storage space is largely independent from the amount of available main memory. Even with only 40 Mb of memory, temporary storage space accounts for only around 6% of the final word-level inverted files. Relatively more temporary storage space is required for the construction of document-level inverted files with an additional storage requirement of around 16%.

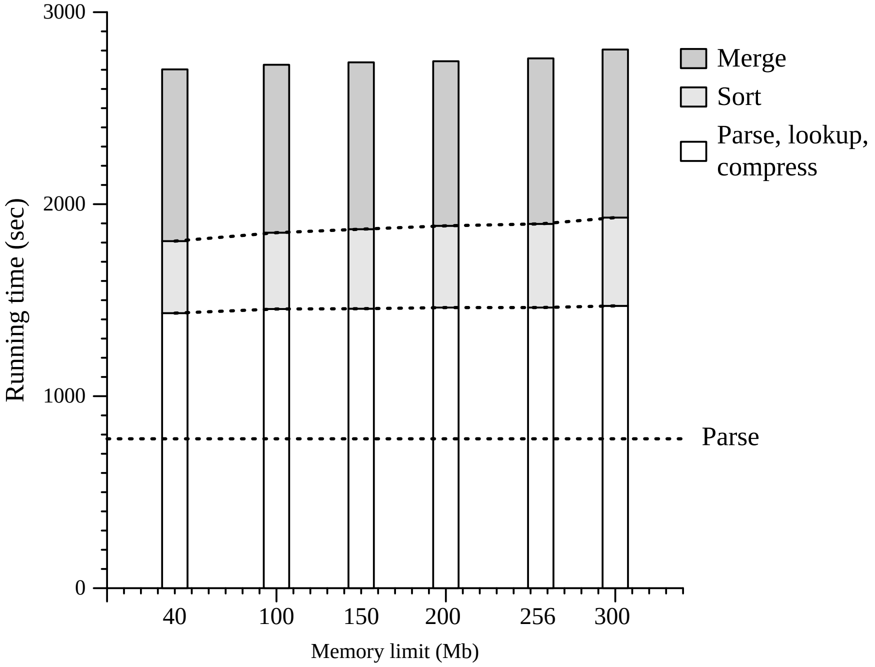


Figure 5: *Break-up of the total elapsed time in seconds for constructing a word-level inverted file for* Web V*, with the sort-based inversion approach, for different main memory limits. The total elapsed time consists of the time to parse, lookup, and compress postings, the time spent sorting the accumulated postings, and the time to merge the runs. The total elapsed time for the parsing process is approximately 778 seconds.*

Overall, the results of the single-pass approach are promising. For moderate volumes of main memory and the construction of word-level inverted files, the single-pass approach outperforms the sort-based approach in terms of running time, by around 15%. The main reason why the single-pass approach outperforms the sort-based approach is the more efficient use of main memory and the avoidance of sorting a large array of postings. Instead of accumulating uncompressed postings in an array, then sorting and compressing them when memory is filled up, postings are directly stored and compressed in bitvectors managed by the lexicon. Each bitvector requires in our implementation 40 bytes for internal counters and variables alone, that is, without storing any postings, but these additional costs are offset by storing postings compressed in main memory. As a result, significantly fewer runs are constructed and merged—typically less than half compared to the sort-based approach.

Each run contains the set of index terms that have occured since the last run, leading to higher storage requirements compared to the sort-based approach. However, for a skewed distribution, the majority of index terms occur only once throughout the collection and, as a result of emptying the lexicon occasionally, main memory is not permanently filled up by index terms that are never accessed again; thus, main memory is used efficiently.

The empirical results of the inversion approaches do not quite match the predicted times calculated earlier. The predicted times are faster than the measured times in our experiments by around 10%. Given our basic computational model, some inaccuracy is not surprising. However, the relative performance of the inversion approaches in our experiments matches the predictions: for a fixed memory limit, the single-pass approach outperforms the sort-based approach by a significant margin.

6 Improvements

We suggest that efficiency of the single-pass approach can be improved by partitioning the index into b logical buckets such that each is of roughly the same size and is small enough to fit into main memory. Each bucket contains a lexicon that is responsible for a range of lexicographically-adjacent index terms. The ranges R_i are non-overlapping, that is, $R_i = (\alpha_i, \omega_i]$ with $R_i \cap R_j = \emptyset$ for $1 \leq i < j \leq b$. During index construction, each lexicon accumulates postings of those index terms that belong to its corresponding term range. To identify the corresponding bucket for a posting, an in-memory data structure is queried that returns the range into which a posting is to be inserted. We refer to this structure as a *range selector*.

When main memory is exhausted, a bucket is chosen and flushed to disk as a run. Runs are written into a file of contiguous disk pages. This file is partitioned into b segments, one segment for each bucket. Each segment is designed to accommodate all runs that are constructed by its corresponding bucket, but if a segment overflows, additional storage space is allocated at the end of the file. To construct the final inverted file, the segments are processed one after the other and the runs from the current segment are merged by an in-place multi-way merge. Since the segments are designed to fit individually into main memory and, since they are stored on contiguous disk pages, the retrieval costs during merging are low.

Arranging runs in segments has potential advantages for large volumes of text. Consider, for example, the inversion of a text collections in the terabyte range on a workstation with 1 Gb of main memory. For such a collection, a large number of runs, say 1,000, would be generated by the single-pass approach and the runs would be scattered among a large temporary file whose size far exceeds that of main memory. Input buffers help to decrease retrieval costs, but to fill input buffers, we have to seek to file locations. In contrast, a multi-way merge of runs from a segment involves more local disk activity and therefore disk accesses which are less costly. Further, by choosing small segment sizes, the number of runs to be merged can be kept small decreasing costs for the heap management of runs. We surmise that, for large volumes of text, these savings could amount to substantial gains.

The maximum size of an individual segment should be limited by the volume of available main memory M. The number of segments (and therefore also buckets), b, must be chosen at the start of the inversion such that each segment is smaller than M when merging starts. Since the exact size of the temporary inverted file to be merged is not known for a text collection *a priori*, b can only be approximated. For this, we must take into account the size and characteristics of the collection, the granularity of the inversion and the compression scheme used, and the results of previous inversions of similar data. In our experiments on word-level inversions of Web documents, we observed that the size of the inverted file prior to the merge is typically around 15% of the collection size S. For each of the b buckets, we have to assign at the start of the inversion a fixed

Table 5: *Elapsed time and time spent on merging in seconds to construct a segmented word-level inverted files with the multiple-bucket single-pass inversion approach for the reference collection* Web X *for different values of b using burst tries to organise the in-memory buckets. The first table entry with $b = 1$ refers to the times measured with the original single-pass inversion approach. Main memory usage is limited to 64 Mb and the maximal size of each input buffer during merging is 2.5 Mb.*

b	Total time	Merge time
1	4,755	1,636
12	4,694	1,578
24	4,695	1,572
36	4,688	1,566
48	4,703	1,564
60	4,712	1,566
72	4,806	1,594

term range R_i. Our approach is to build an in-memory inverted index for a snapshot of documents of the collection. We then divide the accumulated vocabulary into b term ranges such that each range corresponds to a set of inverted lists of roughly the same memory size.

There are several alternative ways to select buckets to be written out to disk when main memory is full. One is to write out the buckets that consume the most main memory. However, this ignores the physical layout of the segments on disk. Our approach is to maintain a list of buckets sorted by the next write position in each bucket. We circle through the list choosing the bucket that is closest to the last write location.

We then tested the efficiency of single-pass inversion using segmented inverted files of b in-memory buckets. The aim is to optimise disk accesses, in particular when runs are merged and when updates are migrated into a large inverted file. Due to data and hardware limitations, however, instead of inverting a 1 Tb collection with 1 Gb of available main memory, which would result in a large number of runs to be merged, we use in our experiments 64 Mb to build word-level segmented inverted files for the 10 Gb collection *Web X*. For this collection, we estimate that the size of the segmented temporary inverted file is around 1.5 Gb and pre-allocate for the b segments a large chunk of disk pages of this size on disk. To minimise disk seeks during merging, we employ for each run in a segment an input buffer of at most 2.5 Mb resulting in a total memory requirement of the input buffers below the memory limit. The term ranges are determined by building an in-memory inverted file for the first 1,000 documents in the collection and partitioning the vocabulary into b ranges such that the sets of inverted lists that belong to the term ranges are of roughly the same size. No action is taken to limit the length of delimiter terms. We use a burst trie in each bucket to accumulate the vocabulary of its term range. We do not use hashing to organise buckets since the storage requirements for b hash tables would amount to a significant portion of the available main memory degrading overall performance.

Table 5 summarises our experimental results. As shown, using multiple buckets slightly decreases overall inversion times compared to inversion with a single bucket. The best results are achieved with 36 buckets; using more buckets degrades performance, presumably because the overall number of runs increases and higher costs for the construction of runs and memory management occur. Interestingly, we observed that the time gains for $b = 36$ are achieved during the merging phase. Merging the 36 segments takes 1,566 seconds, whereas 1,636 seconds are required to merge the runs that are scattered among the temporary file when only one segment is used. We surmise that these gains are due to less disk activity during the merge phase. The results confirm our design decisions for the multiple-bucket single-pass approach. We anticipate that for larger inversions, the performance gap between the segment-wise construction and the non-segment-wise construction of inverted files will widen. Thus, from this preliminary work, we have shown that segmented inverted files are fast to construct and we surmise that index maintenance of segmented inverted files is a promising avenue to explore in future research.

7 Conclusions

We have surveyed existing approaches to inversion of text databases. The main approaches described in the literature have significant drawbacks, in particular that the entire lexicon must be held in memory. Previous comparisons have shown that, of these approaches, sort-based inversion is the most efficient. Our cost estimates confirm these results, which are substantially because the main alternative approach requires two passes over the input data, and parsing the input is a major cost.

We have proposed a new single-pass method that does not require that the whole lexicon be in memory, and which appears to be innately faster than the sort-based method. In outline, this approach is based on simply constructing in-memory inverted files for sequences of documents, continuing until is memory is full, then flushing the inverted file and its lexicon to disk. By making appropriate implementation choices—choice of lexicon data structure, use of list and lexicon compression, and choice of merging strategy—this elementary approach yields excellent performance. Further gains are made by treating the lexicon as a collection of lexicographic partitions, to give good disk locality during merging.

Overall, our single-pass approach is 15% to 20% faster than the sort-based approach for small to large main memory limits on all tested collections from 5 Gb to 20 Gb. This is despite the fact that we allow the sort-based approach to maintain its lexicon outside the memory limit; were the limit enforced, the sort-based approach would not be practical for larger collections. The speed gains are achieved at the expense of an increase in storage requirements, which however is small. Our results show that the single-pass inversion approach is the preferred method for inverted file construction.

8 Acknowledgements

We thank Hugh Williams for his contributions to this work. The project is supported by the Australian Research Council.

References

Bell, T. C., Moffat, A. & Witten, I. H. (1994), Compressing the digital library, *in* 'Proc. of the First Annual Conference on the Theory and Practice of Digital Libraries', College Station, Texas, USA, pp. 41–46.

Bentley, J. L. & McIlroy, M. D. (1993), 'Engineering a sort function', *Software—Practice and Experience* **23**(11), 1249–1265.

Bertino, E., Ooi, B. C., Sacks-Davis, R., Tan, K.-L., Zobel, J., Shidlovsky, B. & Catania, B. (1997), *Indexing Techniques for Advanced Database Systems*, Kluwer Academic Press, Boston, Massachusetts, USA.

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

Fox, E. A. & Lee, W. C. (1991), FAST-INV: A fast algorithm for building large inverted files, Technical Report TR 91-10, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA.

Frakes, W. B. & Baeza-Yates, R., eds (1992), *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

Golomb, S. W. (1966), 'Run-length encodings', *IEEE Transactions on Information Theory* **IT–12**(3), 399–401.

Harman, D. (1995), 'Overview of the second text retrieval conference (TREC-2)', *Information Processing & Management* **31**(3), 271–289.

Harman, D. K. & Candela, G. (1990), 'Retrieving records from a gigabyte of text on a minicomputer using statistical ranking', *Jour. of the American Society for Information Science* **41**(8), 581–589.

Heinz, S. & Zobel, J. (2002), Practical data structures for managing small sets of strings, *in* M. Oudshoorn, ed., 'Proc. Australasian Computer Science Conf.', Melbourne, Australia, pp. 75–84.

Heinz, S., Zobel, J. & Williams, H. E. (2002), 'Burst tries: A fast, efficient data structure for string keys', *ACM Transactions on Information Systems* **20**(2), 192–223.

Melnik, S., Raghavan, S., Yang, B. & Garcia-Molina, H. (2001), Building a distributed full-text index for the Web, *in* 'Proc. of the Tenth International Conference on World Wide Web', ACM, Orlando, Florida, USA, pp. 396–406.

Moffat, A. & Bell, T. A. H. (1995), 'In situ generation of compressed inverted files', *Journal of the American Society of Information Science* **46**(7), 537–550.

Moffat, A. & Zobel, J. (1996), 'Self-indexing inverted files for fast text retrieval', *ACM Transactions on Information Systems* **14**(4), 349–379.

Ribeiro-Neto, B. A., de Moura, E. S., Neubert, M. S. & Ziviani, N. (1999), Efficient distributed algorithms to build inverted files, *in* 'SIGIR '99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 15-19, 1999, Berkeley, CA, USA', ACM, pp. 105–112.

Rogers, W., Candela, G. & Harman, D. (1995), Space and time improvements for indexing in information retrieval, *in* 'Proc. of 4th Annual Symposium on Document Analysis and Information Retrieval (SDAIR '95)', University of Nevada at Las Vegas, USA.

Salton, G. & McGill, M. J. (1983), *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, USA.

Scholer, F., Williams, H. E., Yiannis, J. & Zobel, J. (2002), Compression of inverted indexes for fast query evaluation, *in* 'Proc. ACM-SIGIR Int. Conf. on Research and Development in Information Retrieval'. To appear.

Sedgewick, R. (1998), *Algorithms in C*, third edn, Addison-Wesley, Reading, Massachusetts, USA.

Williams, H. E. (1998), Indexing and Retrieval for Genomic Databases, PhD thesis, Department of Computer Science, RMIT University, Melbourne, Australia.

Williams, H. E. & Zobel, J. (2001), Searchable words on the Web. In submission.

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

Zobel, J., Heinz, S. & Williams, H. E. (2001), 'In-memory hash tables for accumulating text vocabularies', *Information Processing Letters* **80**, 271–277.