Memory Management Strategies for Single-Pass Index Construction in Text Retrieval Systems

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

ABSTRACT

Many text retrieval systems construct their index by accumulating postings in main memory until there is no more memory available and then creating an on-disk index from the in-memory data. When the entire text collection has been read, all on-disk indices are combined into one big index through a multiway merge process.

This paper discusses several ways to arrange postings in memory and studies the effects on memory requirements and indexing performance. Starting from the traditional approach that holds all postings for one term in a linked list, we examine strategies for combining consecutive postings into posting groups and arranging these groups in a linked list in order to reduce the number of pointers in the linked list. We then extend these techniques to compressed posting lists and finally evaluate the effects they have on overall indexing performance for both static and dynamic text collections. Substantial improvements are achieved over the initial approach.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Textual databases; H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness)

General Terms

Experimentation, Performance

Keywords

Information Retrieval, Indexing Performance, Index Compression, Dynamic Indexing

1. INTRODUCTION & RELATED WORK

Inverted files have proved to be the most efficient data structure for large-scale text retrieval systems [15]. For every term that appears in a given text collection, the inverted file contains a list of all positions in the text at which the term occurs. Inverted files that contain less information, e.g. lists of documents identifiers instead of exact positions or frequency-ordered lists with cut-off threshold [9], are very similar in structure and therefore not explicitly considered in this work.

One of the most efficient techniques to build an inverted index from a text collection is through hash-based in-memory inversion: For every input token (<*term, position*> pair) that is read from the input file, the corresponding term descriptor is looked up in the hash table (inserted if it is not present yet), and the position is added to the term's list of postings. Because the total number of occurrences of a term within the text corpus is not known beforehand, an efficient, dynamic data structure is needed to maintain the postings for a term. Usually, linked lists are used for this purpose (cf. Witten et al. [12, pp. 228-230]).

Dictionary implementations other than hash tables, such as burst tries [5], have been examined as well, but hash tables seem offer the greatest performance, especially when used with move-to-front heuristics, which very accurately models the term distribution in most English texts (as discussed by Zobel et al. [14]).

The power of the hash-based in-memory inversion with linked lists is limited by the amount of main memory available. When the text collection is much larger than the available memory, pure in-memory inversion cannot be used. Fox and Lee [2] and Heinz and Zobel [4] have discussed ways to extend the in-memory approach and make it usable for larger collections by splitting the text collection into smaller parts, each of which can be inverted in memory. The size of each part is defined by the amount of memory available; whenever main memory is exhausted, an on-disk index is created from the data in memory. After the whole collection has been processed, the final index is generated by merging the indices created so far through a multiway merge process. If the on-disk data are arranged carefully, this is possible with almost no additional storage space [7].

Although these techniques can be used to employ the memory-inversion technique to invert text collections that are many times the size of the available main memory, further improvements are possible. After a short description of

our experimental setup in the next section (2), we discuss how the space overhead caused by the linked lists in the in-memory inversion approach described above can be reduced in order to minimize the number of sub-indices that have to be created (section 3). This is done by combining sequences of postings into groups and linking these groups instead of individual postings. In the following section (4), we give an introduction to index compression and show how compression techniques can be used to increase memory efficiency even further.

In section 5, we present an experimental evaluation of the performance gains achieved by reducing the memory consumption. Experimental results are given for two different scenarios – static and dynamic document collections. The ability to maintain a dynamic index is crucial in many applications, such as real-time file system search. Limiting the focus to static collections is therefore not appropriate.

Depending on whether the underlying text collection is assumed to be static or dynamic, different results are obtained. For static text collections, it is known that the amount of main memory available has only a small impact on indexing performance [4]. For dynamic collections, however, keeping as much information in memory as possible is crucial. On-disk index structures have to be reorganized whenever main memory is exhausted, which involves a large number of disk operations. For the dynamic case, our experiments have shown speedups of over 200% caused by the increased memory efficiency.

2. EXPERIMENTAL SETUP

For all experiments described in this paper, we used two different text collections:

- TREC disks 4+5 without the Congressional Record (referred to as TREC4+5-CR);
- a subset of the PubMed Medline corpus used in the TREC 2004 Genomics track (referred to as TREC-Genomics).

Basic corpus statistics for both collections can be found in Table 1. TREC4+5-CR has a special status, since it is in the critical range where increasing the memory efficiency can actually shrink the size of the inverted file enough so that the whole index can be built in memory.

For our experiments, we used a 32-bit index address space. This was possible because none of the test collections contains more than 2^{31} postings. However, all results are applicable to 64-bit indices as well. The only difference is that the uncompressed postings would consume twice as much memory if 64-bit postings (and pointers) were used.

All experiments were conducted on a PC based on an Intel Pentium-M processor (1.6 GHz) with a 7200-rpm hard drive. The operating system was GNU/Linux.

3. GROUPS OF POSTINGS

The additional memory requirements caused by the pointers in the linked lists used by the in-memory inversion method described in section 1 are dramatic. Under the assumption that both postings and pointers have the same size (32 bits in our experiments), the space overhead is 100%. This extreme increase is able to reduce the system's indexing performance significantly.

Table 1: Text collections used for the experiments: TREC disks 4+5 without the Congressional Record (TREC4+5-CR) and the subset of the Medline database used for the TREC 2004 Genomics track (TREC-Genomics).

	TREC4+5-CR	TREC-Genomics
Corpus size	1,904 MB	14,332 MB
# documents	528,155	4,591,008
# tokens	$3.23 \cdot 10^8$	$2.05 \cdot 10^9$
# distinct terms	$1.16 \cdot 10^6$	$7.87 \cdot 10^6$
Size of final index	573 MB	3,696 MB

One of the first solutions to this problem was proposed by Fox and Lee [2]. If more than one pass is made over the entire collection, the first pass can be used to gather collection statistics, such as the number of occurrences of each term. This information can then be used in the second pass to allocate enough memory for every vocabulary term to store all its postings. This way, no linked lists are needed, and the pointer overhead can be avoided. We call this the *Two-Pass* strategy. Its disadvantage is that the 50% decrease in memory consumption is likely to be outweighed by the additional disk operations caused by the second pass over the collection.

Another solution was presented and analyzed by Heinz and Zobel [4]. For every term found in the text collection, an initial array is allocated to hold all postings for the term. Every time time the array is full, its size is doubled by allocating new memory. Like in the Two-Pass strategy, no linked lists are necessary to store the postings. (The compression techniques also used by Heinz and Zobel are ignored here but considered in section 4.)

In this section, we present a third solution to the pointer problem. We keep the general technique of organizing postings for the same term in linked lists, but allow pointers in a linked list to point to *posting groups* instead of individual postings. When using posting groups of size 2, for example, we need one pointer for every 2 postings. In the following paragraphs, we define several families of strategies to group uncompressed postings in the in-memory index. All strategies are evaluated theoretically using a statistical approach based on Zipf's law and experimentally using the two test collections.

$\mathbf{Const_n}$

The Const_n strategy allocates a posting group of size n for every term that is added to the vocabulary. Whenever a group becomes full, a new group of size n is created and added to the linked list of groups for the respective term. If the entire collection contains less than 2^{31} postings, every posting can be encoded in 31 bits, which allows us to use the 32nd bit to distinguish between postings and pointers and thus to implicitly encode the size of a posting group in the pointer at the end of the group.

Const_1 is the special case in which the posting list for a term is a linked list of individual postings. The Const_8 strategy is visualized in Figure 1(a).

$\mathbf{Exp_{n,k}}$

When a new term is added to the in-memory index, an initial posting group of size n is created for that term. When-

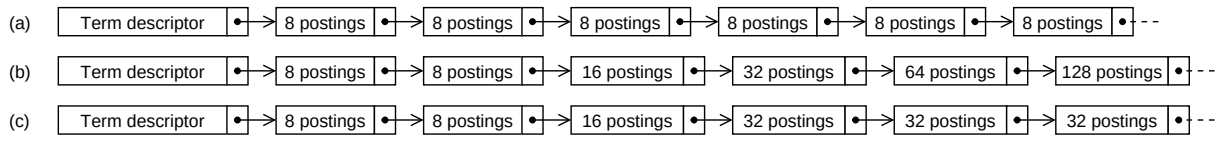

Figure 1: Different strategies for combining postings into groups. (a) Const_8: Linked list with constant group size. (b) $\text{Exp}_{8,2}$: Exponentially increasing group size. (c) $\text{ExpLimit}_{8,2,32}$: Exponentially increasing group size with upper limit.

Table 2: Memory consumption of different grouping strategies. Simulation results for hypothetical text collections with $2.05 \cdot 10^9$ postings, $7.87 \cdot 10^6$ distinct terms, and different Zipf-α values. The space required to store all postings is given relative to Const_1.

$\alpha =$	0.90	0.95	1.00	1.05	1.10
Const_1	100%	100%	100%	100%	100%
Const_2	75.1%	75.1%	75.1%	75.1%	75.1%
Const_4	62.9%	62.9%	62.9%	62.8%	62.8%
Const_8	57.0%	57.0%	57.0%	57.0%	56.9%
Const_{16}	54.8%	54.6%	54.8%	54.5%	54.6%
Const_{32}	55.1%	54.3%	54.3%	54.7%	55.3%
Const_{64}	56.2%	56.7%	57.6%	58.7%	59.7%
$\text{Exp}_{8,2.0}$	73.0%	73.2%	71.1%	72.7%	70.4%
$\text{Exp}_{8,1.5}$	62.4%	62.9%	63.0%	61.8%	60.6%
$\text{Exp}_{8,1.2}$	56.5%	56.3%	55.8%	55.6%	56.1%
$\text{Exp}_{8,1.1}$	54.7%	54.4%	54.3%	53.8%	54.2%
$\text{Exp}_{16,1.1}$	54.6%	54.2%	54.6%	54.1%	53.8%
$\text{ExpLimit}_{8,2.0,256}$	59.5%	57.4%	56.2%	54.7%	53.5%
$\text{ExpLimit}_{8,1.5,256}$	56.1%	55.2%	54.2%	53.3%	52.6%
$\text{ExpLimit}_{8,1.2,256}$	54.2%	53.6%	53.1%	52.6%	52.1%
$\text{ExpLimit}_{8,1.1,256}$	53.8%	53.2%	52.8%	52.4%	51.9%
$\text{ExpLimit}_{8,1.1,512}$	53.8%	53.3%	52.9%	52.4%	51.9%
$\text{ExpLimit}_{16,1.1,256}$	53.6%	53.1%	53.1%	52.4%	52.3%
Two-Pass	50.0%	50.0%	50.0%	50.0%	50.0%

ever a posting group is full, a new group is created and added to the linked list of groups. The size of the new group is

$$\max\{\ n,\ \lceil p \cdot (k-1) \rceil\ \},$$

where p is the total number of postings accumulated for that term so far and k is a tuning parameter whose optimal value depends on the term distribution of the text corpus. For $n = 8$ and $k = 1.5$, for example, posting groups of sizes 8, 8, 8, 12, 18, 27, ... are created.

$\text{Exp}_{8,2}$ is similar to the the growing-array strategy analyzed by Heinz and Zobel [4]. It is shown in Figure 1(b).

$\text{ExpLimit}_{\mathbf{n,k,l}}$

This strategy is similar to $\text{Exp}_{n,k}$. For a new term, an initial group of size n is created. Whenever a group is full, the size of the new group is

$$\min\{\ l,\ \max\{\ n,\ \lceil p \cdot (k-1) \rceil\ \}\ \},$$

where p is again the total number of postings accumulated for that term so far.

The rationale behind limiting the maximum size s of a posting group to some constant l is that, after a certain point, the savings caused by reducing the number of pointers in the linked list become negligible (increasing the group size from $s = 256$ to $s = 512$, for instance, reduces the relative pointer overhead from $4 \cdot 10^{-3}$ to $2 \cdot 10^{-3}$). However, the amount of memory that is potentially wasted – because the last posting group for a term has been allocated but is only partially used – increases as the group size grows.

As an example of the ExpLimit family of grouping strategies, $\text{ExpLimit}_{8,2,32}$ is depicted in Figure 1(c).

3.1 Theoretical Evaluation

We used a theoretical language model based on Zipf's law [13] [3] to evaluate the different grouping strategies. According to Zipf, the number of occurrences of the i-th most frequent term in a text collection of size s with t different terms is

$$f_i = \frac{s}{i^\alpha \cdot H_\alpha(t)},$$

where $H_\alpha(t)$ is the t-th harmonic number of order α, and α is close to 1 (in his original work, Zipf claimed $\alpha = 1$). Thus, Zipf's law can be used to approximate the distribution of the posting lists.

For any given triple (s, t, α), the total amount of memory needed to build an in-memory index for the whole collection consisting of s tokens can be simulated very efficiently. We ran simulations for hypothetical text collections of the same size as the TREC-Genomics corpus and different values of the Zipf parameter α.

The results (shown in Table 2) are surprising in that, due to the properties of the Zipfian distribution, even a very simple strategy like Const_{16} can reduce the amount of memory needed by 45% (compared to the initial strategy, Const_1). Group sizes bigger than 16, however, increase the memory consumption for the Const strategies by increasing the amount of memory that is allocated but not used. The same reason prevents $\text{Exp}_{8,2.0}$ from decreasing memory consumption beyond a certain point; for long lists, one can expect that about 50% of the last group (and thus 25% of the total amount of memory allocated) is unused. As a result, $\text{Exp}_{8,2.0}$ reduces the memory consumption by less than 30%. Decreasing the growth factor k, however, can improve this strategy significantly; $\text{Exp}_{8,1.1}$ achieves a 45-46% space reduction, which is slightly better than Const_{16}.

The savings caused by the Exp family of grouping strategies can be improved further by introducing the group size limit that is used by ExpLimit. The $\text{ExpLimit}_{8,1.1,256}$ strategy is among the best in this family. For the hypothetical test collections, it is able to decrease memory consumption by 46-48%. Compared to the optimal allocation strategy of a two-pass indexing process, $\text{ExpLimit}_{8,1.1,256}$ requires only 4-8% more memory. ExpLimit's advantage over the other grouping strategies grows as the collection parameter α becomes larger.

Table 3: Memory consumption of different grouping strategies. Experimental results for the TREC4+5-CR and TREC-Genomics test collections without in-memory index compression. The space required to store all postings is given in MB.

	TREC4+5-CR	TREC-Genomics
Const_1	2451 (100%)	15640 (100%)
Const_8	1407 (57.4%)	9010 (57.6%)
Const_{16}	1363 (55.6%)	8751 (56.0%)
$\text{Exp}_{8,2.0}$	1837 (74.9%)	11965 (76.5%)
$\text{Exp}_{8,1.1}$	1322 (53.9%)	8431 (53.9%)
$\text{ExpLimit}_{8,2.0,256}$	1294 (52.8%)	8168 (52.2%)
$\text{ExpLimit}_{8,1.1,256}$	1274 (52.0%)	8105 (51.8%)
Two-Pass	1226 (50.0%)	7820 (50.0%)

Table 4: Memory consumption of different grouping strategies. Experimental results for the TREC4+5-CR and TREC-Genomics test collections with byte-aligned on-the-fly compression. The space required to store all postings is given in MB.

	TREC4+5-CR	TREC-Genomics
Const_1	2451 (100%)	15640 (100%)
$\text{Const}_{32}^{(C)}$	622 (25.4%)	4005 (25.6%)
$\text{Const}_{64}^{(C)}$	615 (25.1%)	3979 (25.4%)
$\text{Const}_{128}^{(C)}$	662 (27.0%)	4315 (27.6%)
$\text{ExpLimit}_{16,2.0,512}^{(C)}$	550 (22.4%)	3479 (22.2%)
$\text{ExpLimit}_{16,1.1,512}^{(C)}$	547 (22.3%)	3464 (22.1%)
$\text{Two-Pass}^{(C)}$	512 (20.9%)	3278 (21.0%)

3.2 Experimental Results

In addition to the theoretical results presented above, we conducted experiments using the two test collections summarized in Table 1. The set of strategies evaluated on the test collections was restricted to the ones that were most space-efficient in the simulations described above. The results of our experiments are shown in Table 3.

In general, all numbers are very close to those obtained in the simulations run before, showing once more that the Zipfian distribution can be used to realistically model term distributions for large text collections. $\text{Exp}_{8,1.1}$ and the ExpLimit strategies perform slightly better in the experiments than in the simulations (for both text collections, $\text{ExpLimit}_{8,1.1,256}$ stays within 4% of the optimal, two-pass allocation strategy), while all other strategies perform slightly worse than in the simulations.

4. ON-THE-FLY INDEX COMPRESSION

It is known that the memory efficiency of an indexing system can be increased by storing postings in a compressed form. Most compression techniques work by transforming the list of postings for a term into a list of word gaps, i.e. a list of distances between two consecutive postings of the same term. Gap lists can then be compressed using various compression algorithms, such as Gamma encoding [1], the Hyperbolic method [11], or Interpolative compression [8].

Compression algorithms for posting lists can be divided into two categories: parameterized and non-parameterized methods. Parameterized compression models assume the availability of some information about the distribution of the gaps between two consecutive occurrences of the same term. If no such information is available, they cannot be used, and a non-parameterized method has to be chosen instead.

We use the byte-aligned compression algorithm evaluated by Scholer et al. [10]. It is a non-parameterized method, similar to Elias' Gamma encoding, that can be used to compress postings on-the-fly as they are appended to existing in-memory posting lists. Gaps between two consecutive postings for the same term are stored in a variable number of bytes. The lower 7 bits of every byte are used to store the actual gap information, while the most significant bit is used as a continuation indicator, i.e. it is 0 if the last byte of the current gap has been reached and 1 otherwise.

Because the compression algorithm operates on whole bytes instead of individual bits, as most of the other compression schemes, it is extremely fast, yet offers compression rates close to those of the best bitwise methods. C code for the compression and decompression of posting lists is given in Figure 2.

The grouping strategies defined and reviewed in the previous section can be extended to support compressed posting lists. Instead of allocating chunks of memory that can hold a certain number of postings, the system allocates byte arrays that are used to store posting lists in compressed forms. These byte arrays are then arranged in linked lists as before. The compressed representation of a single gap may cross chunk boundaries, having the first 2 bytes in one chunk and the remaining 2 bytes in the next, for example.

In order to be able to employ on-the-fly gap compression, the last posting for every term has to be kept in memory as part of the term descriptor. This increase in memory consumption ($\approx$ 5 MB for TREC4+5-CR and $\approx$ 30 MB for TREC-Genomics) is not reflected by the experimental results shown in this section because we focus on the amount of memory occupied by the postings themselves and disregard the memory consumption of the term descriptors. It is, however, taken into account in the experiments discussed in section 5, which focus on overall indexing performance instead of the memory consumed by the postings.

In analogy to the previous section, we define three different families of grouping strategies:

$\textbf{Const}_{\mathbf{n}}^{(\mathbf{C})}$

A byte array of size n is allocated for every new term. Every time the array is full, a new array of size n is allocated. Because the compressed postings that are stored in the byte array can take arbitrary values, it is no longer possible to use the most significant bit to distinguish between postings and pointers (cf. section 3). Therefore, the size of each chunk is stored explicitly as a 1-byte integer. Including the `next` pointer, this leads to a total overhead of 5 bytes per chunk. Thus, $\text{Const}_{16}^{(C)}$, for instance, would allocate memory in pieces of 21 bytes each.

$\textbf{Exp}_{\mathbf{n,k}}^{(\mathbf{C})}$

Because we decided to encode the size of each chunk in a 1-byte integer, it cannot grow beyond a certain bound. Thus, no experimental results for $\text{Exp}_{n,k}^{(C)}$ are available. We conjecture, however, that the effect of index compression on

```
int compress(int *uncompressed, byte *compressed, int count) {
  int pos = 0;
  for (int i = 0; i < count; i++) {
    int delta = uncompressed[i];
    if (i > 0)
      delta = delta - uncompressed[i - 1];
    while (delta >= 128) {
      compressed[pos++] = (delta % 128) + 128;
      delta = delta / 128;
    }
    compressed[pos++] = delta;
  }
  return pos;
}
```

```
void uncompress(byte *compressed, int *uncompressed, int count) {
  int pos = 0;
  for (int i = 0; i < count; i++) {
    int delta = 0, shift = 1;
    while (compressed[pos] >= 128) {
      delta = delta + shift * (compressed[pos++] % 128);
      shift = shift * 128;
    }
    delta = delta + shift * compressed[pos++];
    uncompressed[i] = delta;
    if (i > 0)
      uncompressed[i] += uncompressed[i - 1];
  }
}
```

Figure 2: Compression and decompression routines for the byte-aligned word gap compressor. An efficient implementation would use bit shifts instead of the multiplications/divisions shown here.

the Exp strategies is similar to that on ExpLimit.

$\mathbf{ExpLimit}^{(\mathbf{C})}_{\mathbf{n,k,l}}$

Initially, n bytes are allocated for every new vocabulary term. Every time the memory allocated for a term is exhausted, a new byte array of size

$$\min\{\ l,\ \max\{\ n,\ \lceil p \cdot (k-1) \rceil\ \}\ \}$$

is created, where p is the total number of bytes consumed by the term's postings so far.

4.1 Experimental Results

The experimental results for the grouping strategies with built-in compression are shown in Table 4. From the results for the Two-Pass strategy, we can see that the compression method employed decreases the net memory consumption of the postings by about 58% (comparing TwoPass in Table 3 and $\text{TwoPass}^{(C)}$ in Table 4). The reason why this increase is not completely reflected by the other strategies ($\text{Const}^{(C)}_{64}$ saves 55% compared to Const_{16}; $\text{ExpLimit}^{(C)}_{16,1.1,512}$ saves 57% compared to $\text{ExpLimit}_{8,1.1,256}$) is the relative increase of the overhead introduced by pointers and the additional chunk size information.

However, the loss of space efficiency is minimal. Even if compression is used, $\text{ExpLimit}^{(C)}_{16,1.1,512}$ is still within 6-7% of the optimal Two-Pass strategy for both test collections.

5. INDEXING PERFORMANCE

So far, we have discussed the impact that different grouping strategies have on the memory consumption of the indexing system. The more relevant aspect, however, is their effect on overall indexing performance. In this section, we study their influence on performance. For this purpose, we consider two different scenarios: indexing a static document collection and indexing a dynamically growing collection.

Traditionally, indexing performance has only been studied for static collections, which do not change over time. However, for most applications, having a static collection is not a realistic assumption and an oversimplification of the actual conditions. For some collections, the degree of dynamics might even be so great that the number of updates to the index is bigger than the number of queries that have to be answered. One such application is file system search, in which every change to an indexable file leads to an index update operation. Static and dynamic collections have to be studied separately, as they require different indexing approaches.

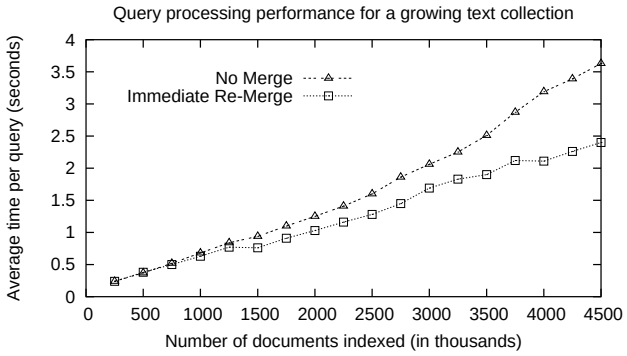

Figure 3: Query processing performance on a dynamic text collection (TREC-Genomics), using the $\text{ExpLimit}^{(C)}_{16,1.1,256}$ grouping strategy. As the collection grows, merging the individual inverted files becomes more and more important. Performance is decreased significantly by the large number of disk seeks necessary if sub-indices are not merged.

For the static collection, the standard indexing approach with text-based partitioning described in section 1 is taken. A sub-index is created every time main memory is full; after the whole collection has been indexed, all sub-indices are merged into the final index. In order to avoid unnecessary disk operations, no on-disk index is created for the last part of the collection. Instead, the in-memory postings are fed directly into the merge process.

For the growing collection, the situation is fundamentally different. Because the collection is dynamic, we do not know at what point the entire collection has been indexed (in fact, that point is never reached). In addition, the system has to be able to efficiently process queries that are submitted while the indexing takes place. Therefore, instead of keeping a large number of sub-indices that are eventually merged, a new on-disk inverted file is created by immediately merging the existing one with the in-memory postings every time the system runs out of main memory. This technique, called *Immediate Re-Merge*, has been analyzed by Lester et al. [6] and is their suggested way to deal with a growing text collection.

Table 5: Experimental results for static text collections. The number of sub-indices created during index construction is proportional to the memory consumption of the respective strategy.

	TREC4+5-CR				TREC-Genomics			
	# Sub-indices	Disk I/O	Indexing time		# Sub-indices	Disk I/O	Indexing time	
Const_1	11	3,674 MB	4m09s	(100%)	68	26,714 MB	29m01s	(100%)
Const_8	7	3,643 MB	3m28s	(84%)	42	26,271 MB	24m29s	(84%)
Const_{16}	6	3,648 MB	3m23s	(82%)	43	26,393 MB	24m06s	(83%)
$\text{Exp}_{8,2.0}$	8	3,620 MB	3m25s	(82%)	48	26,428 MB	24m33s	(85%)
$\text{ExpLimit}_{8,2.0,256}$	6	3,499 MB	3m19s	(80%)	40	26,325 MB	23m30s	(81%)
$\text{ExpLimit}_{8,1.1,256}$	6	3,525 MB	3m18s	(80%)	39	26,265 MB	23m34s	(81%)
Two-Pass	6	5,419 MB	4m36s	(111%)	36	38,037 MB	31m26s	(108%)
$\text{Const}_{16}^{(C)}$	4	3,586 MB	3m21s	(81%)	22	25,945 MB	22m49s	(79%)
$\text{Const}_{32}^{(C)}$	3	3,280 MB	3m06s	(75%)	20	25,837 MB	22m12s	(76%)
$\text{ExpLimit}_{16,2.0,256}^{(C)}$	3	3,355 MB	3m05s	(74%)	18	25,556 MB	21m45s	(75%)
$\text{ExpLimit}_{16,1.1,256}^{(C)}$	3	3,356 MB	3m07s	(75%)	18	25,581 MB	21m58s	(76%)

One could argue that, even if the collection is dynamic and queries have to be processed while the indexing takes place, the on-disk sub-indices do not necessarily have to be merged. This is true. It is in fact possible to keep multiple on-disk inverted files and combine the posting lists retrieved from the individual indices into a bigger list whenever a query has to be processed (we call this the *No Merge* strategy).

However, the potentially large number of additional disk seek operations caused by having more than one on-disk index is likely to significantly deteriorate query processing performance. The relative query processing performance of *No Merge* and *Immediate Re-Merge* on a growing text collection is shown in Figure 3. Using $\text{ExpLimit}_{16,1.1,256}^{(C)}$ to group postings in memory, the 18 on-disk indices that exist after 4.5 million documents have been indexed cause *No Merge* to have a query processing performance that is approximately 35% lower than that of *Immediate Re-Merge*. This demonstrates how important sub-index merging is in dynamic environments.

5.1 Experimental Results

All experiments were conducted with 256 MB of main memory available for all in-memory data. This includes the hash table, term descriptors, and postings. Therefore, the number of sub-indices created in the individual experiments differs from what would have been expected, given the memory consumption figures from the previous sections.

Nonetheless, the results for the static text collections (shown in Table 5) are roughly in line with what we have seen in sections 3 and 4. The number of sub-indices can be decreased by up to 74% relative to Const_1. The strategies with on-the-fly compression slightly outperform the strategies that do not use compression, mainly by reducing the amount of redundant data (term descriptors) written to the individual sub-indices. The best strategy reduces the indexing time by 25% compared to the initial approach.

The biggest performance improvement happens when going from Const_1 to Const_8. For the most part, it does not stem from the decrease in the number of disk operations, but from a decrease in CPU time! In order to understand this, one has to realize that for Const_1 the traversal of an in-memory posting list is a random walk through the available memory, causing a large number of CPU cache misses.

The results for the dynamic collection (Table 6) are fundamentally different from the static case. When indexing a dynamic collection, reducing the indexing system's memory consumption decreases the number of re-merge operations and thus significantly increases the system's indexing performance. In fact, the total indexing time is almost proportional to the number of merge operations performed. This means that – in contrast to the static case – the amount of main memory available and the memory efficiency of the system greatly affect indexing performance for dynamic collections; the time needed to build the whole index is essentially linear in the memory consumption caused by the in-memory posting lists. Compared to Const_1, the total time needed to build the index can be decreased by 70% if $\text{ExpLimit}_{16,2.0,256}^{(C)}$ is used.

For both static collections, it can be seen that Two-Pass is not a competitive strategy, exhibiting worse performance than Const_1. The additional disk operations caused by the second pass are not compensated for by increased memory efficiency. In the dynamic case, the situation is different. On the smaller TREC4+5-CR collection, Two-Pass can beat Const_1, and its advantage over the single-pass strategies grows as the collection becomes larger: on the TREC-Genomics collection, Two-Pass shows performance very close to the ExpLimit strategies. However, this is not a fair comparison. Two-Pass is inherently non-dynamic, as it needs to wait until the available main memory is exhausted before new documents are actually added to the index. Nobody would actually use it in a dynamic environment.

6. CONCLUSION

We have studied the most popular indexing technique used in text retrieval systems, hash-based in-memory inversion, and examined the effect that different strategies to organize in-memory postings have on both memory consumption and indexing performance. A theoretical and experimental evaluation of different strategies has shown that, in conjunction with on-the fly index compression, the right strategy is able to decrease the indexing system's memory consumption by more than 75%.

Unfortunately, the reduced memory consumption only leads to a small, sub-linear speedup when following the traditional information retrieval paradigm of indexing static

Table 6: Experimental results for dynamic text collections. The number of merge operations corresponds to the number of sub-indices created in the static case.

	TREC4+5-CR				TREC-Genomics			
	# Merge op's	Disk I/O	Indexing time		# Merge op's	Disk I/O	Indexing time	
Const_{1}	10	8,614 MB	7m41s	(100%)	67	264,042 MB	205m58s	(100%)
Const_{8}	6	6,403 MB	5m20s	(69%)	41	168,945 MB	127m18s	(62%)
Const_{16}	5	5,333 MB	4m35s	(60%)	42	174,714 MB	137m30s	(67%)
$\text{Exp}_{8,2.0}$	7	7,009 MB	5m53s	(77%)	47	192,076 MB	146m55s	(71%)
$\text{ExpLimit}_{8,2.0,256}$	5	5,467 MB	4m43s	(61%)	39	163,689 MB	124m15s	(61%)
$\text{ExpLimit}_{8,1.1,256}$	5	5,543 MB	4m44s	(62%)	38	159,282 MB	124m59s	(61%)
Two-Pass	5	7,551 MB	6m03s	(79%)	35	159,997 MB	125m20s	(61%)
$\text{Const}_{16}^{(C)}$	3	4,672 MB	4m08s	(54%)	21	98,000 MB	76m36s	(37%)
$\text{Const}_{32}^{(C)}$	2	3,677 MB	3m23s	(44%)	20	89,972 MB	69m06s	(34%)
$\text{ExpLimit}_{16,2.0,256}^{(C)}$	2	3,793 MB	3m22s	(44%)	17	80,868 MB	61m52s	(30%)
$\text{ExpLimit}_{16,1.1,256}^{(C)}$	2	3,795 MB	3m25s	(44%)	17	81,075 MB	62m44s	(30%)

document collections. This result is consistent with earlier findings [4].

For modern retrieval systems, however, which support dynamically growing collections, the performance increase is dramatic, since the number of merge operations involving on-disk inverted files (and thus the number of disk operations) is inversely proportional to the memory efficiency of the indexing system. Speedups of more than 200% can be achieved by choosing the right strategy to organize postings in memory.

As a main contribution of this paper, we have presented a one-pass indexing technique whose memory consumption is very close to that of the optimal two-pass strategy, but which does not require the significant overhead of a second pass over the text collection and does therefore deliver superior performance, especially in dynamic environments.

7. REFERENCES

[1] P. Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, 1975.

[2] E. A. Fox and W. C. Lee. FAST-INV: A Fast Algorithm for Building Large Inverted Files. Technical report, Blacksburg, Virginia, USA, March 1991.

[3] A. Gelbukh and G. Sidorov. Zipf and Heaps Laws' Coefficients Depend on Language. In *Proceedings of the Second Conference on Intelligent Text Processing and Computational Linguistics (CICLing-2001)*, pages 332–335, 2001.

[4] S. Heinz and J. Zobel. Efficient Single-Pass Index Construction for Text Databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.

[5] S. Heinz, J. Zobel, and H. E. Williams. Burst Tries: A Fast, Efficient Data Structure for String Keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[6] N. Lester, J. Zobel, and H. E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *CRPIT '26: Proceedings of the 27th Conference on Australasian Computer Science*, pages 15–23, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[7] A. Moffat and T. C. Bell. In-Situ generation of Compressed Inverted Files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.

[8] A. Moffat and L. Stuiver. Exploiting Clustering in Inverted File Compression. In Storer and Cohn, editors, *Proceedings of the 1996 Data Compression Conference*, pages 82–91, 1996.

[9] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered Document Retrieval with Frequency-Sorted Indexes. *Journal of the American Society for Information Sciences*, 47(10):749–764, 1996.

[10] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, 2002.

[11] E. Schuegraf. Compression of Large Inverted Files with Hyperbolic Term Distribution. *Information Processing and Management*, (4):377–384, 1976.

[12] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes (2nd ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1999.

[13] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, Massachusetts, USA, 1949.

[14] J. Zobel, S. Heinz, and H. E. Williams. In-Memory Hash Tables for Accumulating Text Vocabularies. *Information Processing Letters*, 80(6), 2001.

[15] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted Files versus Signature Files for Text Indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.