Storage Management for Files of Dynamic Records

Justin Zobel
Department of Computer Science,
RMIT, GPO Box 2476V, Melbourne 3001, Australia.
jz@cs.rmit.edu.au

Alistair Moffat
Department of Computer Science
The University of Melbourne, Parkville 3052, Australia.
alistair@cs.mu.oz.au

and

Ron Sacks-Davis
Collaborative Information Technology Research Institute
723 Swanston St., Carlton 3053, Australia.
rsd@cs.rmit.edu.au

Abstract

We propose a new scheme for managing files of variable-length dynamic records, based on storing the records in large, fixed-length blocks. We demonstrate the effectiveness of this scheme for text indexing, showing that it achieves space utilisation of over 95%. With an appropriate block size and caching strategy, our scheme requires on average around two disk accesses—one read and one write—for insertion of or change to a record.

1 Introduction

Many database applications require that files of variable-length dynamic records be managed. For example, inverted files are frequently used for indexing stored data. For non-key attributes, each distinct indexed value can occur many times, so that the entry for each distinct value—that is, the list of records containing that value—can be of any length, and, moreover, can change in length as records are inserted, deleted, or updated. Space efficient representations of these entries require that each be stored as a unit[1,2,3], hence the necessity that variable-length dynamic records be stored efficiently. This behaviour is also typical of other applications, for example, the disk buckets of a hashed file, and the nodes of a B-tree.

We propose a new method for managing dynamic records on disk, based on storing records in large, fixed-length blocks. This scheme was sketched briefly by us in a previous paper[3]; here we give both details of our proposal and experimental results in support of our performance claims. In our scheme, each block contains several records, a table describing those records, and, in general, some free space. The amount of free

space is kept small by relocating records when blocks overflow. Disk access costs are kept low by the observation that, with current technology, it is not much more costly to retrieve a large block than a small block, and by storing the contents of each record contiguously.

We have tested our scheme, using as an example the management of indexes for full-text database systems. We generated inverted files for several test databases and simulated the performance of our scheme on these indexes, measuring space utilisation and the number of disk accesses required as each term was inserted into its inverted file entry. These simulations showed space utilisation of, in most cases, over 95%.

Our results compare well with those quoted for earlier schemes. Knuth's description of memory management schemes, including best fit methods and the buddy system, includes experiments that show space utilisation of less than 80%[4]. Biliris has described a page allocation scheme for large objects that achieves excellent within-page space utilisation, but relies on the buddy system to supply it with appropriate pages[5,6] and thus, if we include the cost of unused pages, does not achieve such good utilisation overall. Nor have other schemes for index management been as effective. The schemes surveyed by McDonell[7] appear to have space utilisation of up to 50%; the exact utilisation is unclear, as items such as pointers between list nodes are regarded as part of the data, whereas we would include these as overhead. The schemes described by Faloutsos and Jagadish do better; they do not give space utilisation for their experiments, but in their graphs the maximum utilisation is about 90%[8].

Our results can be applied to problems such as B-tree indexes. Utilisation in B-trees is usually quoted at 69%, and a recent B-tree scheme 'with good performance' only achieves space utilisation of 82%[9]. We are currently developing B-tree algorithms based on our space management scheme that we believe will have substantially higher utilisation ratios.

In Section 2 we review inverted file indexing. The indexes and test databases we used to test our scheme are described in Section 3. In Section 4 we describe the structures we propose for managing dynamic records on disk; maintenance of these structures is described in Section 5. Results of simulation of our scheme are given in Section 6.

2 Inverted file indexes

A general inverted file index for a text database consists of two main components: a set of *inverted file entries*, being lists of identifiers of the documents containing each indexed term; and a *search structure* for identifying the location of the inverted file entry for each term, containing a *vocabulary* of query terms, and an *index mapping* that maps ordinal term numbers onto disk addresses in the inverted index. This arrangement is illustrated in Figure 1. The index mapping can either be stored on disk as a separate file, or can be held in memory with the vocabulary. We assume that inverted file entries store ordinal document numbers rather than addresses, and

so to map the resulting document identifiers to disk addresses there must also be a *document mapping*.

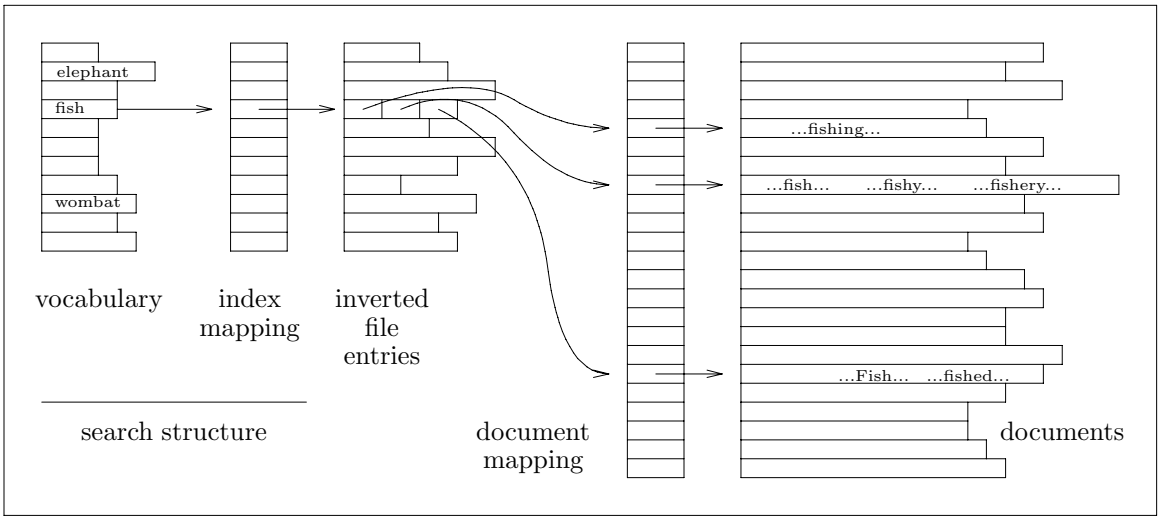

Figure 1: Inverted file index structure

We assume that the contents of each inverted file entry are stored contiguously, in contrast to other schemes, in which entries are often stored as linked lists with each node randomly placed on disk[7,8,10], or with the documents chained together on the basis of common attribute values[11]. Given this structure, queries are processed by using the vocabulary and index mapping to find the location of the inverted file entry for each term in the query; accessing the disk to retrieve the inverted file entries for those terms; performing any logical operations on those entries; and then using the document mapping to access the document file to retrieve the answers. If the search structure is a B-tree, typically one or at most two disk accesses will be required to find the location of the inverted file entry of each term, since the memory of current machines is large enough to hold a substantial portion of the search structure. The number of accesses to retrieve the entry itself is the number of blocks containing the entry, which in our scheme will usually be one; then, depending on whether the document mapping is in memory or on disk, one or two disk accesses will be required for each answer.

To simplify query evaluation, inverted file entries should be maintained in sorted order of document number. On insertion of new terms into existing documents, this will involve shifting large volumes of data in memory to make space for the new identifier, but such shifts are faster than retrieval of the same volume of data from disk. Since shifts of data will generally precede a write to disk, the shift will be dominated by other costs.

	Bible	*Comact*	*Trec*
Text size (Mb)	4.4	132.1	1,198.7
Number of records	31,102	261,829	511,514
Distinct terms	9,073	36,840	407,820
Distinct terms per record (av.)	23	49	165
Longest entry (bytes)	3,888	32,729	63,939
Average entry length (bytes)	60	242	154
Total number of terms	701,533	12,745,527	84,286,478
Compressed index size (Mb)	0.5	8.5	59.9

Table 1: Sizes of test databases

3 Test databases and indexes

We used the inverted files of three text databases to test our method for storing files of dynamically changing variable length records. Each database consists of documents; the indexed values were the words occurring in each document. Thus some indexed values, such as the word 'the', were frequently repeated, about half occurred more than once, and each document contained many values to be indexed. Index entries noted only whether or not a document contained the word being indexed, so that even if a document contained several occurrences of a word the inverted file entry only listed that document's number once. Before the indexes were generated, the words in the documents were stemmed—that is, case was folded and variant endings such as 'ed', 'ing', and 'lessly' were removed[12].

The databases were the complete King James edition of the Bible, or *Bible*, in which each document is a verse; the complete Commonwealth Acts of Australia, 1901 to 1989, or *Comact*, in which each document is a page; and the (first) TREC collection of articles extracted from the TIPSTER database, or *Trec*, in which each document is an article from such sources as the Wall Street Journal or Associated Press, or a legal document such as a United States Federal Regulation. The parameters of these databases are given in Table 1.

Each inverted file entry is a sorted list of ordinal document identifiers. Since the largest identifier is not known in advance, in a fixed-length representation some number of bits must be chosen in advance to store each identifier. Typically 24 or 32 bits would be suitable. As we have shown elsewhere, however, compression techniques can be used to represent these entries much more compactly, typically reducing the space required by a factor of about five while still permitting fast decoding[1,2]. With such compression, each addition of a record will add, for each term in the record, a small number of bits to the inverted file entry of that term. In our simulations we have concentrated on compressed entries; the figures in Table 1 assume that each

entry is compressed using a 'local V_G' code[2].

4 Storage of records on disk

There are two difficulties in managing dynamic records on disk. The first is that they vary in length. The second is that each record can change in length over time.

We propose a space management scheme in which records are stored in large, fixed-length blocks[3]. In this scheme, records are numbered contiguously from zero. This number is used as an index into a *record mapping* indicating which disk block contains each record. Records are stored in fixed-length blocks of B bytes, where B is significantly greater than the average record length. (We consider records of more than B bytes later.) Each block contains a *block table* of addresses of records within the block, used to find the location of an record within the block once the block has been retrieved from disk. The record mapping need only store the number of the block containing each record; to retrieve a record, the whole block is fetched. With current disk performance, it only takes about twice as long to fetch a block of 64 Kb than a block of 1 Kb*, and, as we show in the next section, block sizes of around 32 Kb to 64 Kb give good results with our data.

Blocks will in general have some free space, because the records and block table do not occupy exactly B bytes, as illustrated in Figure 2. To minimise shifting of data within each block, the block table is stored at one end of the block and the records at the other, so that both grow towards the middle.

To manage free space we require a *free list*, which will generally be short, of descriptions of blocks (other than the last block of the file) whose free space is in excess of some fixed *tolerance*, typically a few percent. This list should be held in memory; if it becomes large, indicating that the file is becoming disorganised, data should be moved amongst blocks to get better space utilisation. However, the free list itself will not occupy much space: a linked list representation will require at most 12 bytes per block, to hold the number of the block, the number of free bits in the block, and a pointer to the next node in the list. For example, on a file of 1 Gb stored in 64 Kb blocks, the free list will consume at most 200 Kb, and normally much less.

The last block in the index should also be held in memory, rather than on disk, to facilitate update. The value of keeping the last block in memory is illustrated by the insertion and update algorithms shown below.

The block table can account for a significant fraction of a block's space. For example, if record numbers and record lengths are represented in 32 bits each, a block of only 128 entries will have a 1 Kb table; in the simulations described below, blocks of several hundred small records were common. A simple technique for reducing the

*We note that typical figures for disk retrieval are not very informative. For example, for the three kinds of disk drives attached to our Sun SPARCserver 2, the overheads of retrieving 64 Kb compared to retrieving 1 Kb were 30%, 45%, and 400%. Factors involved include seek, latency, caching and prefetch strategies, and disk channel speed.

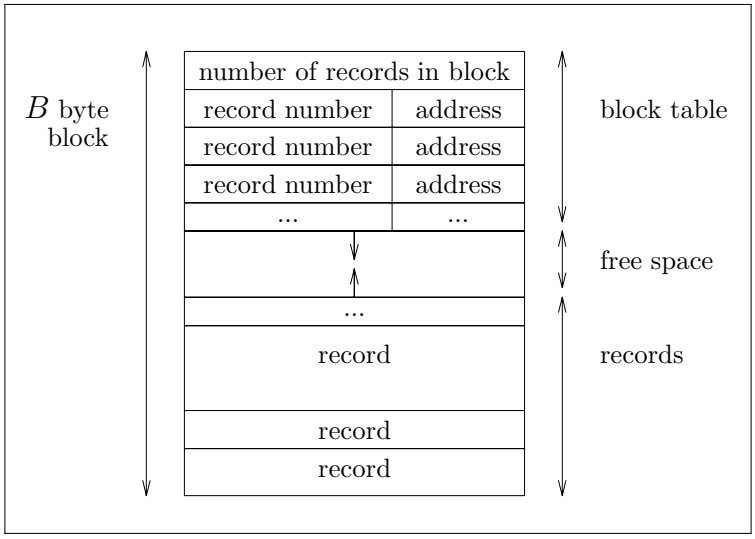

Figure 2: Block structure

size of the block table is to use fixed length representations of values, but keep the length of these representations as small as possible. For example, if the block size is 16 Kb then 17 bits is sufficient to identify any bit in the block. The maximum record number can vary, but it is straightforward to let the largest record number held in the block govern the number of bits used to store each record number; for example, if the largest record number was 881,634 then $\lceil \log_2 881,634 \rceil = 20$ bits should be used. This implies that the block table must occasionally be rebuilt, but in practice rebuilds will be rare and in any case require only a few instruction cycles per record.

Another technique for reducing the size of the block table is to compress it. Since the records can easily be stored in order, the techniques we use for compressing inverted file entries can be applied to record numbers. However, compared to the fixed length scheme described above, the additional space saving would be relatively small. Space in the table might also be saved by byte or word alignment of records within the block. For example, byte alignment would waste an average of four bits per record, but would allow table pointers that were three bits shorter. Moreover, in many situations records will in fact be an integral number of bytes, or even words, long.

The final component of our scheme is an *update cache*, used to keep access costs down during insertion, update, and deletion. This cache contains a series of pairs, each pair being the number of the record to be changed and the change to be made to that record. In text databases, for example, it is common for adjacent documents to contain duplicate terms, so if updates to the index are cached then disk accesses can be saved. The *size* of the cache is the number of pairs it can contain. In our

simulations, described in Section 6, we investigated a range of cache sizes.

5 Index maintenance

Given the file structures described in the previous section, file maintenance is straightforward. New records should be inserted according to the following algorithm.

1. If there is a block in the free list with sufficient free space, add the record to that block and update the free list.
2. Otherwise, if the last block has sufficient free space, add the record to the last block.
3. Otherwise, write the last block to disk, add it to the free list if necessary, create a new last block in memory, and add the record to that block.

'Adding a record to a block' consists of copying the record into the block; updating its block table; and updating the record mapping. Whichever of the above steps is used, if the record is smaller than a block, and if the record mapping is in main memory, then at most one disk read and one disk write is required to update the file; and, assuming that inserted records are small, will usually require no disk operations.

Record extension should be performed with the following algorithm.

1. Read the block containing the record to be extended, if it is not already in memory.
2. If the block contains sufficient free space for the extended record, move other records to accommodate the altered record, update the block table, and write the block to disk.
3. If there is not sufficient free space,
   (a) If moving a record in the block to either a block in the free list or the last block leaves sufficient space, then move that record, update the record mapping, and return to step (2).
   (b) Otherwise, move the extended record to the last block, creating a new last block if necessary.
4. Update the free list.

Assuming that the record is smaller than a block, at most two reads and two writes are required by this procedure, and usually one read and one write will be sufficient because only the block containing the record will be affected. Note that moving the average $B/2$ bytes sideways within a block, as required for update in place, will take considerably less time than a single disk access for reasonable values of B. For

example, on a Sun SPARCserver 2, 32 Kb can be moved in about 5 milliseconds, compared to about 40 milliseconds to write a 64 Kb block.

Variations on the above procedures based on strategies such as best fit, worst fit, and so on, can be applied. Record deletion and contraction can be handled by procedures that are similar to those described above, with similar costs.

With large blocks, many records can be kept in each block and most records will fit within a block. Some records may, however, be larger than a block. There are several possible strategies for dealing with such long records. We chose to allow each block to contain, in addition to short records, up to one record that was larger than the block size and a series of pointers to overflow blocks containing the parts of the record that did not fit in the main block. There is a trade-off between choice of block size and the proportion of records that are longer than a block. Large blocks imply greater retrieval time, but the block size should be substantially greater then the size of the average record, or the average number of disk accesses required to retrieve a record will be large.

Update of long records requires that each of the blocks containing part of the record be read and written, so in practice it is desirable to store the overflow blocks contiguously, possibly in a separate file. Although dynamically maintaining contiguity would be expensive, an occasional re-ordering could be used to keep access costs down. In our simulations, we have pessimistically assumed that a separate access is required for each overflow block. Biliris[5,6] considers in detail the problem of storing dynamic records that are large compared to the block size.

6 Results

To test our space management scheme we implemented a space usage simulator that kept track of space utilisation, block table size, and number of disk accesses during creation of inverted file indexes. The parameters of this simulator were block size, tolerance, and cache size. As input, we used sequences of 'entry-num, bit-incr' pairs, where the bit-incr is the number of bits an update operation adds to the entry with identifier entry-num. These pairs were output from a program for creating inverted indexes, and so the data stream was, in effect, the sequence of term insertions that would take place if a compressed inverted index was being built during a linear pass over the document collection, assuming that the vocabulary and compression parameters had been accumulated in a pre-scan of the text.

Space utilisation

Our measure of space utilisation is the space required to store entries as a percentage of the total file size, so that a file 8,000 bits long with 7,000 bits of entries would have an 87.5% space utilisation, with the remaining 12.5% occupied by block tables and

any free space[†]. The final space utilisation for our test databases is given in Table 2 in the first group of columns; the second group of columns shows the average ratio of records to blocks. The tolerance used for these experiments was 5%.

Block size (Kb)	Utilisation (%)			Records per block		
	Bible	*Comact*	*Trec*	*Bible*	*Comact*	*Trec*
4	95.5	97.5	97.3	64.3	30.7	70.3
8	94.9	97.3	96.9	127.8	46.5	101.9
16	93.5	97.4	96.7	252.0	76.8	157.6
32	93.5	97.1	96.6	504.1	131.1	249.1
64	93.5	96.7	96.8	1,008.1	261.3	409.5

Table 2: Space utilisation

As this table shows, excellent space utilisation is achieved across a range of block sizes. The utilisation does not change once the block size significantly exceeds the size of the longest entry, so that, for example, the utilisation for *Bible* does not change for blocks of 16 Kb or over; as shown in Table 1, the longest *Bible* entry is about 4 Kb. In these results, when space had to be created in a block the smallest sufficiently large entry was moved; we also tried moving the first sufficiently large entry (to minimise shifting of data within the block), but did not achieve such good space utilisation with this strategy. The entry, when moved, was placed in any block with sufficient free space; use of other strategies, such as best fit or worst fit, did not appear to make a significant difference to space utilisation.

Space utilisation as a function of index size during the construction of the *Trec* index is shown in Figure 3. The dip in the graph beginning at 15 Mb is a quirk of our data: a large number of new terms are introduced at that point. The minimum utilisation of 93%—excluding the initial index creation phase, in which utilisation is poor because the entries are very small compared to the block size and the block table size is more significant—is typical of all of the indexes and block sizes.

There was no marked relationship between cache size and space utilisation. However, we would expect use of a large cache to degrade utilisation slightly—the grouping of updates means that larger numbers of bits are being added to each record, so that fine control over allocation of entries to blocks is not possible. In our experiments, a cache of size 1,000 typically led to a 0.1% decline in utilisation. We found that space utilisation was also almost independent of tolerance for values between 1% and 10%; the main effect of changing tolerance was in the length of the free list. In our experiments, for tolerances of 5% or more the free list did not exceed 500 nodes.

[†]For consistency with previous reports[4,5,6,7,8], we have not, however, included the cost of the mapping that identifies, for each record number, the block containing the record.

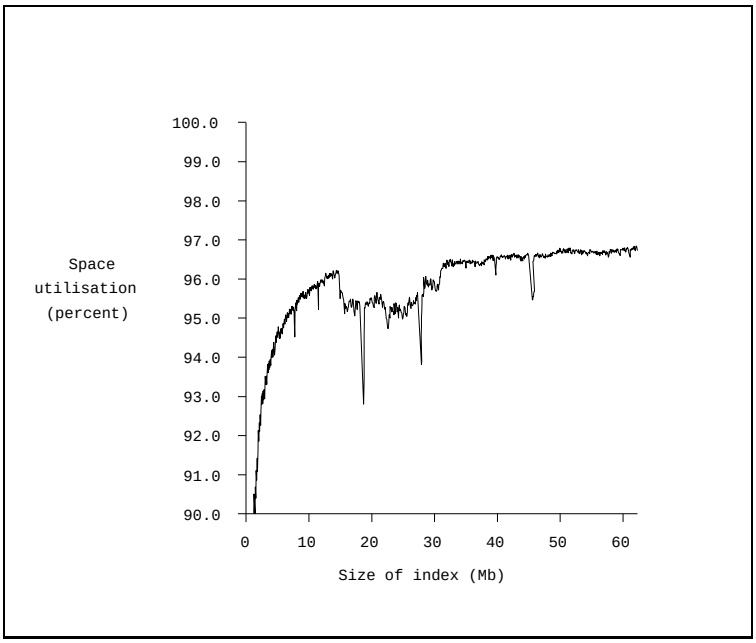

Figure 3: Space utilisation for 16 Kb blocks in *Trec*

We also experimented with use of run-length compression for the record numbers in the block table. This gave only a slight improvement in utilisation, however, with a typical change from 96.0% to 96.5%. Given that an uncompressed table can be searched much more rapidly, we do not believe that such compression is worthwhile.

Disk accesses

Although block size does not have much impact on space utilisation, it does affect the number of disk accesses required for an extension, as shown in Table 3. Since the longest entries will occupy many short blocks, and in our 'extension only' experiments are the entries that are accessed most frequently, the number of accesses required for an average update declines rapidly with increase in block size. Again, once the block size exceeds the length of the longest entry the cost stabilises, as expected, at less than 2 accesses per operation, assuming that the record mapping is held in main memory.

Block size	*Bible*	*Comact*	*Trec*
4	2.0	3.6	4.1
8	1.9	2.9	3.2
16	1.9	2.4	2.7
32	1.8	2.0	2.3
64	1.6	2.0	2.0

Table 3: Average disk accesses per operation

Based upon these results, block sizes of 32 Kb to 64 Kb appear to be well suited to our data; however, the best choice of block size will depend heavily on the average entry length, on the distribution of entry lengths, and the frequency with which longer entries are accessed during operations other than extension. We are currently planning experiments to test our scheme in such an environment.

The number of disk accesses needed for an update is approximately double the number required to retrieve an entry in answer to a query. Thus, although more data may have to be fetched if long blocks are used—because more entries not pertinent to the query are fetched—long blocks also imply fewer accesses, because there are fewer overflow blocks to be retrieved, and, within bounds, less access time overall.

Caching

Use of a cache of updates has, like block size, a dramatic effect on the number of disk accesses required for an average operation, as shown in Table 4. (The cache does not, of course, affect the cost of retrieving an entry.) In our simulations the cache was used in a very simple way: if the cache contained several updates to an entry then these updates were performed together. We did not attempt to order updates so that, for example, two entries residing in the same block were updated at the same time, but such an optimisation would only further improve the performance of the cache; for example, for 16 Kb blocks and a cache of size 1,000,000 we estimate that this optimisation would reduce the number of disk accesses per insertion by a factor of at least 10.

The number of disk accesses required per operation, with and without caching, is shown as a function of index size in Figure 4. The number of disk accesses does grow slowly with the size of the index. This is because the number of entries that are longer than a block increases, and extra disk accesses are required for overflow blocks. However, for large blocks the number of disk accesses is certainly not excessive, and, given that, with our scheme, the index never needs to be rebuilt, the claim that inverted files are expensive to maintain[13] no longer holds.

We have used large caches for our experiments because a single record insertion can generate hundreds—or, in the case of one of the *Trec* records, over twenty thousand—

Cache size	*Bible*	*Comact*	*Trec*
1	1.9	2.4	2.7
10	1.8	2.4	2.7
100	1.3	1.7	2.6
1,000	0.6	0.6	1.7
10,000	0.2	0.2	0.7
100,000	≈0.05	≈0.07	0.2
1,000,000	<0.01	≈0.02	≈0.07

Table 4: Average disk accesses per operation with caching and 16 Kb blocks

updates to an index. For other kinds of data such large caches may not be practical, particularly if insertions are infrequent, although if insertions are infrequent their cost is not so important. With text data, small caches are unlikely to be effective, because each entry will be modified at most once per document insertion, and each document contains many distinct terms.

7 Conclusion

We have described a scheme for managing large collections of records of dynamically varying length. The application we have used to prove the technique is the storage of a compressed inverted file to a large text collection, and in this case we have achieved space utilisations in excess of 93%; update costs of less than two disk operations per operation or amendment; access costs of a little over one disk operation per retrieval; and have assumed only that the record mapping (two bytes per record) can be stored in memory, and that relatively fast access to large disk blocks is possible. To date we have applied the technique to the storage of inverted indexes for text databases through the use of a simulator, as described in this paper. However, the technique is also applicable to a variety of other application areas, such as the storage of the text of such a document collection; the storage of buckets of data in a hashed file system; and the storage of B-tree nodes in a tree structured file system. Experiments to test the usefulness of the method in these areas with a full implementation are underway.

8 Acknowledgements

We would like to thank Neil Sharman for several suggestions and assistance with the implementation. This work was supported by the Australian Research Council.

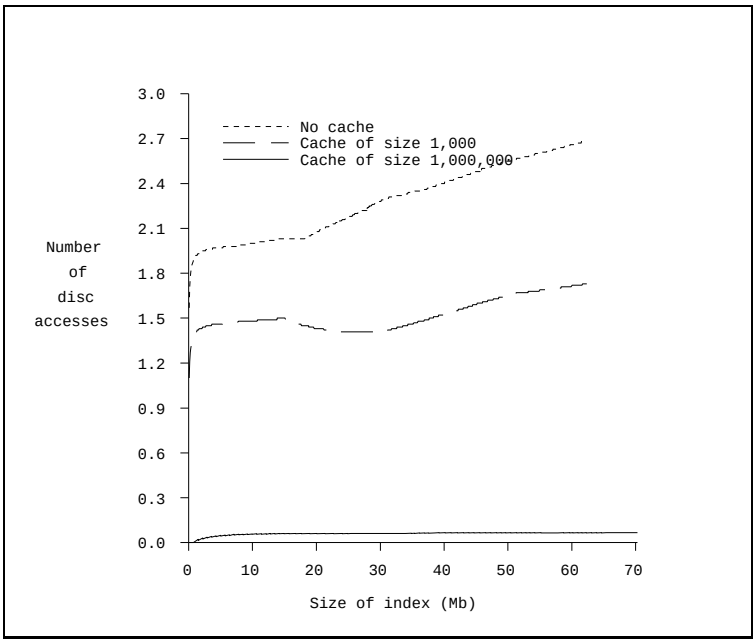

Figure 4: Number of disk accesses for 16 Kb blocks in *Trec*

9 References

1. A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In *Proc. IEEE Data Compression Conference*, pages 72–81, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.
2. A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 274–285, Copenhagen, Denmark, June 1992. ACM Press.
3. J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. International Conference on Very Large Databases*, pages 352–362, Vancouver, Canada, August 1992.
4. D.E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition*. Addison-Wesley, Massachusetts, 1973.
5. A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. ACM-SIGMOD International Conference on the Management of Data*, pages 276–285, 1992.
6. A. Biliris. An efficient data storage structure for large dynamic objects. In *Proc. IEEE International Conference on Data Engineering*, pages 301–308, Phoenix, Arizona, February 1992.
7. K.J. McDonell. An inverted index implementation. *Computer Journal*, 20(1):116–123, 1977.

8. C. Faloutsos and H.V. Jagadish. On B-tree indices for skewed distributions. In *Proc. International Conference on Very Large Databases*, pages 363–374, Vancouver, 1992.
9. D.B. Lomet. A simple bounded disorder file organisation with good performance. *ACM Transactions on Database Systems*, 12(4):525–551, 1988.
10. G. Wiederhold. *File Organisation for Database Design*. Computer Science Series. McGraw-Hill, New York, 1987.
11. D. Grosshans. *File Systems Design and Implementation*. Prentice-Hall, New Jersey, 1986.
12. J.B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computation*, 11(1-2):22–31, 1968.
13. C. Faloutsos. Access methods for text. *Computing Surveys*, 17(1):49–74, 1985.