Fast Incremental Indexing for Full-Text Information Retrieval*

Eric W. Brown
brown@cs.umass.edu

James P. Callan
callan@cs.umass.edu

W. Bruce Croft
croft@cs.umass.edu

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
USA

Abstract

Full-text information retrieval systems have traditionally been designed for archival environments. They often provide little or no support for adding new documents to an existing document collection, requiring instead that the entire collection be re-indexed. Modern applications, such as information filtering, operate in dynamic environments that require frequent additions to document collections. We provide this ability using a traditional inverted file index built on top of a persistent object store. The data management facilities of the persistent object store are used to produce efficient incremental update of the inverted lists. We describe our system and present experimental results showing superior incremental indexing and competitive query processing performance.

Keywords: full-text document retrieval, incremental indexing, persistent object store, performance

1 Introduction

Full-text information retrieval (IR) systems are well established tools for satisfying a user's information need when the information base is a relatively static collection of documents. However, modern information management systems must be able to handle a steady influx of new information. Applications such as information filtering and daily news feed services are constantly processing new documents. If IR systems are to support such applications, they must be able to manage continually growing document collections.

A prerequisite to supporting a growing document collection is the ability to update the data structures used to index the collection. An indexing structure used by many IR systems is the inverted file index [SM83, Fal85, HFBYL92]. An inverted file index consists of a record, or inverted list, for each term that appears in the document collection. A term's inverted list stores a document identifier and weight for every document in which the term appears. The weight might simply be the number of times the term appears in the document, or a more sophisticated measure of the significance of the term's appearance in the document. Additionally, the location of each occurrence of the term in the document may be stored in order to support queries based on the relative positions of terms within documents.

When a batch of new documents is added to an existing document collection, a small number of the terms in the batch will be new to the collection, while the majority of the terms in the batch will already have inverted lists in the index. These lists must be updated by appending to them the term occurrences found in the new documents. This task is made difficult by the size characteristics of inverted lists and the techniques used to manage them in traditional IR systems. The inverted lists for a multi-gigabyte document collection will range in size from a few bytes to millions of bytes, and they are typically laid out contiguously in a flat inverted file with no gaps between the lists.

Adding to inverted lists stored in such a fashion requires expensive relocation of growing lists and careful management of free-space in the inverted file. Rather than update existing inverted lists when adding new documents, many IR systems simply rebuild the inverted file by adding the new documents to the existing collection and indexing the entire collection from scratch. This technique is expensive in terms of time and disk space, resulting in update costs

*This work is supported by the National Science Foundation Center for Intelligent Information Retrieval at the University of Massachusetts.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

proportional to the size of the total collection after the addition. Instead, we would prefer the cost of the update to be proportional to the size of the new documents being added.

The INQUERY full-text information retrieval system [TC91, CCH92] provides this desirable update performance using the Mneme persistent object store [Mos90] to manage its inverted file index [BCCM94]. The key to providing fast incremental indexing is a unique inverted file structure made possible by the data management facilities of the persistent object store. Inverted lists are allocated in fixed size objects with a finite range of sizes, limiting the number of relocations a growing list will experience. When an inverted list has exceeded the largest object size, additional large objects are allocated and chained together in a linked list. Experimental results show that our system is able to provide superior incremental indexing performance in terms of both time and space, with only a small impact on query processing.

In the next section we briefly describe the main architectural features of INQUERY and Mneme. In Section 3 we discuss our inverted file structure in detail, including the motivation for the design. We present experimental results in Section 4 comparing the performance of our system with traditional techniques. In Section 5 we discuss related work, and we offer concluding remarks in Section 6. Contributions of our work include a technique for supporting incremental update of inverted lists in full-text information retrieval systems, along with an empirical analysis of a working implementation. Also, we continue to demonstrate that data management facilities for IR systems need not be custom built to obtain superior performance. Rather, IR systems can be effectively supported using appropriate "off-the-shelf" data management software.

2 Architecture

In this section we highlight some of the basic features of INQUERY and Mneme that are relevant to this work. Throughout the paper we will refer to the system as it is described here as the "old" version.

2.1 INQUERY

INQUERY is a probabilistic information retrieval system based upon a Bayesian inference network model [TC91, CCH92]. The power of the inference network model is the consistent formalism it provides for reasoning about evidence of differing types, allowing multiple retrieval models, document representations, and query representations to be combined simultaneously. Extensive testing has shown INQUERY to be one of the best IR systems, as measured by the standard IR metrics of recall and precision [Har94, TC92]. INQUERY is fast, scales well to large document collections, and can be embedded in specialized applications.

In INQUERY, document retrieval is accomplished by combining evidence from the document collection with evidence from the query to produce a ranking of the documents in the collection. The evidence for a document collection is pre-computed and stored as the weights and locations in an inverted file index. During retrieval, the inverted list for each term in the query is accessed and the evidence in the lists is accumulated and combined as dictated by the operators in the query. The inverted list for a term is obtained by looking up the term in the term dictionary. The term dictionary is built as a hash table with open-chaining conflict resolution. A term's entry in the dictionary contains collection statistics for the term and a reference to the term's inverted list. Inverted lists are stored as Mneme objects, where a single object of the exact size is allocated for each inverted list.

2.2 Mneme

The Mneme persistent object store [Mos90] was designed to be efficient and extensible. The basic services provided by Mneme are storage and retrieval of objects, where an object is a chunk of contiguous bytes that has been assigned a unique identifier. Mneme has no notion of type or class for objects. The only structure Mneme is aware of is that objects may contain the identifiers of other objects, resulting in inter-object references.

Objects are grouped into files supported by the operating system. Within files, they are physically grouped into *physical segments*. A physical segment is the unit of transfer between disk and main memory and is of arbitrary size. Objects are also logically grouped into *pools*, where a pool defines a number of management policies for the objects contained in the pool, such as how large the physical segments are, how the objects are laid out in a physical segment, how objects are located within a file, and how objects are created. Object format is determined by the pool, allowing objects to be stored in the format required by the application that uses the objects (modulo any translation that may be required for persistent storage, such as conversion of main memory pointers to object identifiers). Pools provide the primary extensibility mechanism in Mneme. By implementing new pool routines, the system can be significantly customized.

The base system provides a number of fundamental mechanisms and tools for building pool routines, including a suite of standard pool routines for file and auxiliary table management. Support for sophisticated buffer management is provided by an extensible buffering mechanism. Buffers may be defined by supplying a number of standard buffer operations (e.g., allocate and free) in a system defined format. How these operations are implemented determines the policies used to manage the buffer.

3 Indexing

The old version of INQUERY uses the traditional method of indexing a document collection and building the inverted

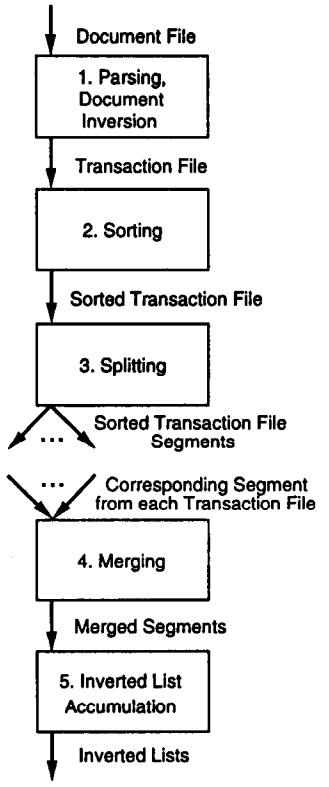

Figure 1: Traditional indexing process.

file [HFBYL92]. This method is referred to as the *alternative* scheme in [STGM94]. The process involves multiple steps, diagramed in Figure 1. The input to the process is a file of documents, which in INQUERY may currently be at most 256 Mbytes[1]. Collections larger than this limit must be broken up into multiple files. In step 1, document parsing assigns an identifier to each document and extracts the terms from the documents, stemming each term to its root and eliminating any stop words (words too frequent to be worth indexing). A surviving term must then be located or inserted in the term dictionary to obtain its term identifier and update the statistics stored there. After each document is parsed, it is inverted in main memory and a temporary file of *transactions* is generated for the document. A transaction consists of a term identifier, the document identifier, and the locations of each occurrence of that term in the document, representing the portion of the inverted list for the term that is contributed by the document.

To convert the transactions into inverted lists, all of the per document inverted list portions for each term must be combined by sorting on the term identifier and document identifier. The transaction files may be over one-and-a-half times the size of the document files, however, so for large (multi-gigabyte) collections, it is impractical or impossible to combine the transactions for all of the documents into a

[1]This is due to the current encoding method used for document file byte offsets. The largest number that can be encoded is 2^{28}.

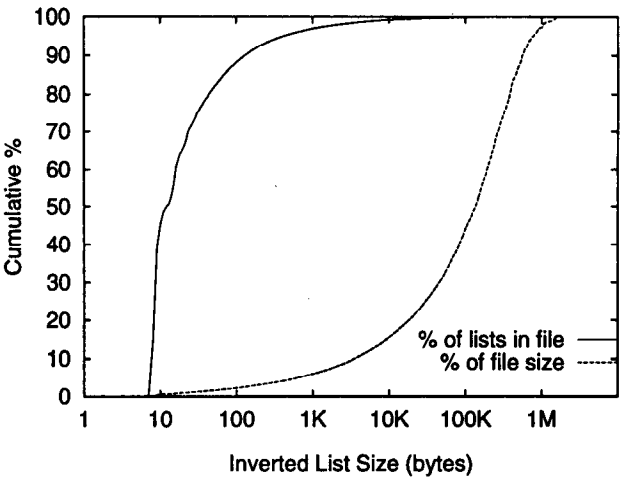

Figure 2: Cumulative distributions over inverted list size for TIPSTER volume 1, with 627072 lists and 420 Mb total.

single file. Instead, the transactions are stored in multiple files, one for each document file.

The transaction file from each document file is sorted in step 2, and the sorted transaction file is split one or more times into head and tail segments in step 3. All of the transaction files are split at the same term t_k, such that the head segment for each transaction file will contain terms $\{t_1 \ldots t_k\}$ and the tail segment for each transaction file will contain terms $\{t_{k+1} \ldots t_n\}$, where there are n terms in the vocabulary. The corresponding segments for each transaction file can then be merged in step 4 to produce a fully sorted set of transactions for the entire document collection, broken up into multiple segments.

The final step is to build the inverted file. This is simply a matter of reading each inverted list from the sorted transactions, compressing the list, creating a Mneme object for the list, and storing the object identifier for the list in the term dictionary entry for the term. In step 5, the transaction segments are processed in order and the inverted lists are created. Obviously, steps 3 and 4 are required only when there are too many transactions from the document collection to handle in a single file. These steps may be eliminated when indexing smaller collections.

Since the entries in each inverted list are sorted by document identifier, if new documents are always assigned increasing document identifiers, their inverted list entries may simply be appended to any existing inverted lists. Therefore, adding new documents to an existing collection does not necessarily require that the entire document collection be re-indexed. However, it does require the ability to grow inverted lists. Providing this functionality is non-trivial, such that many IR systems, the old INQUERY included, simply re-index the entire collection.

Using the full functionality of Mneme, however, we can create a more sophisticated inverted file structure that will

in fact support growing inverted lists. To guide this design, we first consider some of the characteristics of inverted lists, all of which can be derived from the early observations of Zipf [Zip49]. Figure 2 shows the distribution of inverted list sizes for the TIPSTER volume 1 document collection used in our performance evaluation below (see Table 1). For a given inverted list size, the figure shows how many lists in the inverted file are less than or equal to that size, and how much those lists contribute to the total file size. Our first observation is that over 90% of the inverted lists are less than 1000 bytes (in their compressed form), and account for less than 10% of the total inverted file size. Furthermore, nearly half of all lists are less than 16 bytes. This means that many inverted lists will never grow after their initial creation. Therefore, they should be allocated in a space efficient manner, i.e., reserving extra space for these lists in anticipation of growth would be a mistake. Also, it is well known that the vocabulary will continue to grow indefinitely [Hea78], so we must always be prepared to create more of these small inverted lists.

Most of the inverted file size is accounted for by a very small number of large inverted lists. These inverted lists will experience continuous, possibly vigorous growth. We have also observed that these large lists have a high probability of being accessed during query processing [BCCM94], so they must be allocated in a manner that affords efficient access.

The main extension we make to the old inverted file structure is this; instead of allocating each inverted list in a single object of the exact size, we allocate lists using a range of fixed size objects, where the sizes range from 16 to 8192 bytes by powers of 2 (i.e., 16, 32, 64, ..., 8192). When a new list is created, an object of the smallest size large enough to contain the list is allocated. A list can then grow to fill the object. When it exceeds the object, a new object of the next larger size is allocated, the contents of the old object are copied into the new object, and the old object is freed. When a list exceeds the largest object size (8192 bytes), rather than copy the list into an even larger object, we start a linked list of 8192 byte objects. Inverted list growth is then accomplished by appending to the tail object in the linked list, and adding a new object to the linked list when the tail is full.

The largest objects are each allocated in their own physical segment and managed by a *large object pool*. The smallest (16 byte) objects are stored in 4 Kbyte physical segments, 255 objects per segment, and managed by a *small object pool*. The remaining objects are stored in 8 Kbyte physical segments, where each segment stores objects of only one size, and contains as many objects of that size as will fit. These objects are managed by a *medium object pool*.

This scheme efficiently allocates the large number of relatively small inverted lists in the small and medium object pools, limits the number of times a list will be relocated due to growth, and most importantly, needs to access only the tail object when growing a large inverted list, leaving the majority of the data in the inverted file untouched during an update.

Now, the inverted file index can be built incrementally. To add a batch of documents to the index, we parse just the new batch, sort the generated transaction file to create inverted lists for the new documents, and then add the inverted lists to the existing inverted file, creating and growing inverted lists as described above. This corresponds to steps 1, 2, and 5 in Figure 1, with step 5 modified appropriately. The first two steps are performed independently of the existing index. They are also disk oriented operations, with small main memory requirements. Furthermore, they produce complete inverted lists for the new documents. During the final step an inverted list in the index will be updated only once, minimizing the number of costly relocations. Consequently, best performance is achieved when new documents are added to the index in the largest batches possible, reducing the overall number of updates.

4 Performance Evaluation

To evaluate our new inverted file structure, we measured incremental indexing speed, disk space requirements, and query processing speed using the resulting inverted file. For comparison, we also measured the same costs using traditional techniques. Below we describe our experimental platform, the test collection used, and the results of our measurements.

4.1 Platform

All of our experiments were run as superuser with logins disabled on an idle DECSystem 3000/400 (Alpha AXP CPU clocked at 133 MHz) running OSF/1 V1.3. The system was configured with 64 Mbytes of main memory and six 1.3 Gbyte RZ58 SCSI disks. The executables were compiled with the DEC C compiler driver 3.11 using optimization level 2. All of the data files and executables were stored on the local disks, and a 64 Mbyte "chill file" was read before each batch update or query processing run to purge the operating system file buffers and guarantee that no inverted file data was cached by the file system across runs. All times reported below are real (wall clock) time.

4.2 Test Collection

For our experiments, we used volume 1 of the *TIPSTER* document collection, a standard test collection in the IR community. Volume 1 is a 1.2 Gbyte collection of full-text articles and abstracts, divided into seven main files. Table 1 gives the relevant statistics for each of the files. The first three files contain *Wall Street Journal* articles from years 1987, 1988, and 1989 respectively, **doe** contains Department of Energy abstracts, **ziff** contains articles and abstracts

Table 1: TIPSTER volume 1 file characteristics. Terms are new with respect to all files earlier in the table.

File	Mb	Docs	Posts	Terms
wsj87	125.5	46449	11142690	125666
wsj88	104.3	39906	9402198	54292
wsj89	36.5	12380	3260711	17293
doe	183.8	226087	17061350	119467
ziff	242.5	75180	19764843	97489
ap	254.9	84678	21701358	78475
fr	249.5	26207	23599380	134390
total	1197.0	510887	105932530	627072

from various periodicals, **ap** contains Associated Press articles, and **fr** contains Federal Register articles. For all indexing and query processing, we use stemming to reduce words to roots, and a stop words list to eliminate the frequent words not worth indexing.

4.3 Large Updates

Our first experiment treats each of the seven files as a separate batch update and incrementally indexes the volume one file at a time, in the order listed in Table 1. The results are plotted in Figure 3. For the two "per update" lines ("Old, per update" and "New, per update"), each point represents the time required to add the respective batch to the existing index (the first batch creates the initial index), where the points moving left to right correspond to the files in Table 1 moving top to bottom. The per update times are plotted against the total number of posts in the indexed collection after the update.

"Old" is the traditional scheme that re-indexes the entire collection each time a new batch is added. The figure indicates that the time to add a batch is proportional to the size of the entire collection after the update. This scheme clearly does not scale well with the size of the collection. "New" is the new scheme which builds the inverted lists for an update batch separately and then adds them to the existing index, using the new inverted file structure. The time to add a batch is more proportional to the size of the batch, yielding incremental update times that are significantly better than in the old scheme.

Each point for the old scheme also gives the total time required to index a collection of the cumulative size in one batch (since the entire collection is re-indexed). The comparable times for the new scheme are the cumulative times for the incremental updates, shown as "New, cumulative". This plot shows that the total time to incrementally index the entire volume in the given batches with the new scheme is actually less than the time required by the old scheme to index the entire volume in a single batch. This rather surprising result is due to the elimination in the new scheme

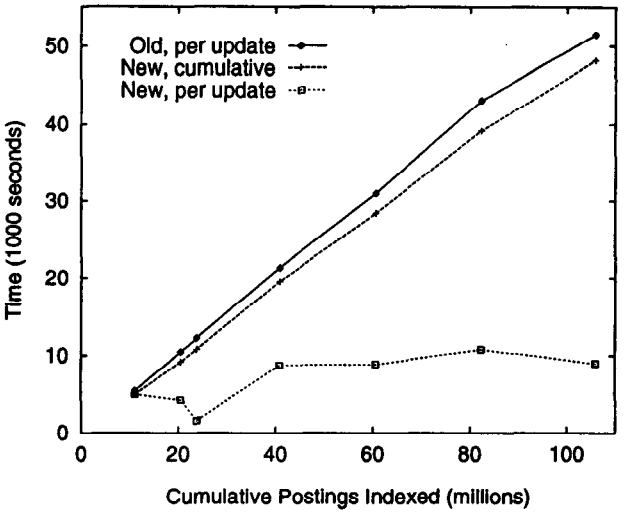

Figure 3: Per batch update times for files in TIPSTER volume 1, plotted versus the total number of postings in the indexed collection after the update.

of a number of sequential passes over the temporary transaction files. Recall from Figure 1 that in order to handle a large document collection, the old scheme must split and merge transaction files in steps 3 and 4. The new scheme eliminates these steps, saving up to four sequential passes through the large temporary files. This represents a fairly significant constant factor and the difference in the slope of the "Old, per update" and "New, cumulative" lines. Note that a cumulative plot for the old scheme is not shown. Clearly, that plot would quickly shoot up off of the top of the chart in a super-linear fashion.

In Figure 4 we show the batch update time per posting for the two schemes. This is the time required to add the batch to the index divided by the number of postings in the batch. Here we can more clearly see that the update cost of the old scheme grows with the size of the existing index, while the update cost of the new scheme remains nearly constant. The large peak for the old scheme during the **wsj89** update is due to the relatively small size of the update compared to the size of the existing collection. The slight dip for both schemes (in Figures 3 and 4) during the **fr** update occurs because the documents in **fr** are relatively large. Compared to a similar size batch with smaller documents, there are fewer document identifier/weight entries in the inverted lists for the same number of postings, and the per posting update cost is reduced.

4.4 Small Updates

The batches used above might be considered large for applications that support periodic updates to the document collection. Therefore, we also measured our new scheme on a range of small batch sizes. We used TIPSTER volume 1 for our existing indexed collection, and generated new doc-

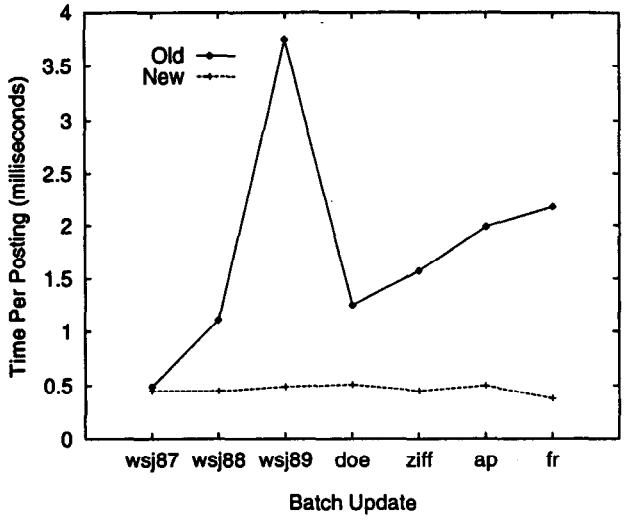

Figure 4: Update time per posting for batch updates in TIPSTER volume 1.

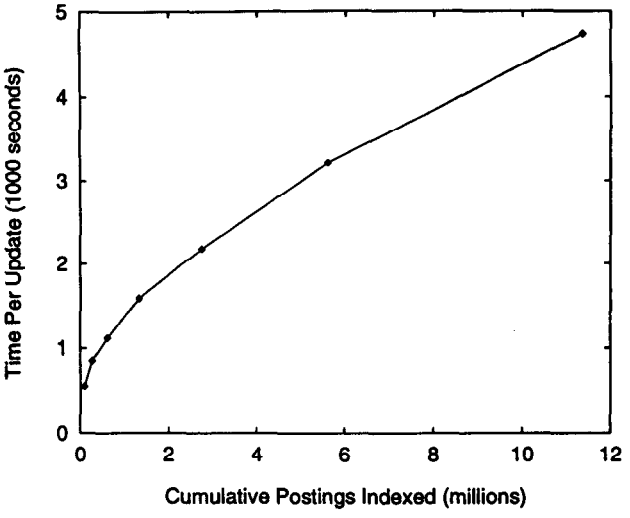

Figure 5: Per batch update times for range of batch sizes cumulatively added to TIPSTER volume 1 index, plotted versus the cumulative postings from the batches. Batches range in size from 1 Mbyte to 64 Mbytes by powers of 2.

Table 2: Characteristics of batches built from TIPSTER volume 2. Terms are new with respect to all batches earlier in the table, with TIPSTER volume 1 as the base.

Batch	Mb	Docs	Posts	Terms
1	1	172	89567	443
2	2	432	176409	1053
3	4	1076	350666	1256
4	8	2012	714205	2517
5	16	3668	1438467	4758
6	32	7363	2841542	10727
7	64	12620	5760450	18770
total	127	27343	11371306	39524

ument batches from volume 2 of the TIPSTER collection, which contains the same types of files as volume 1. To build the batches, we selected documents from the different file types in round robin fashion until we had built seven batches ranging in size from 1 Mbyte to 64 Mbytes, by powers of 2. Statistics for the batches are given in Table 2.

Figure 5 shows the time required to incrementally index each of the batches and cumulatively add them to the existing index. The data points moving left to right correspond to batches 1 through 7 in Table 2, and are plotted against the cumulative postings from the batches. We also plot the time per posting for each batch update in Figure 6. The figures show that as the size of the batch increases, the cost per posting decreases. This is due to the following factors. First, when building new inverted lists for a batch, enough (maybe all) of the term dictionary must be read in to identify all of the terms in the batch and make entries for new terms. Similarly, the dictionary must again be read in during the build phase when adding the new lists to the existing index. With larger batches, this cost is amortized over more postings. Second, larger batches will benefit from locality in the small and medium object pools. The inverted lists in these pools are clustered in segments, such that reading in a list to modify it causes all lists in the same segment to be read in. The cost of reading in the segment can then be amortized over any updates to other lists in the same segment that occur before the segment is flushed out.

In an effort to better accommodate small batch updates, we implemented an in-memory version of our system which performs updates on-line. Rather than build complete inverted lists for the batch off-line and add them to the index in a second step, the in-memory version builds the inverted lists for the batch in main memory as the documents in the batch are parsed. When the main memory inverted list buffer becomes full, the partial inverted lists for the batch are added to the existing index. The rationale behind this scheme is that we will eliminate redundant I/O caused by writing transactions, sorting off-line to build the inverted lists, and then reading the lists again to add them to the index. To test this scheme, we allocated 10 Mbytes for the inverted list buffer (which accommodates around 800,000 compressed postings), 20 Mbytes of Mneme buffer space for the existing inverted index, and 24 Mbytes of Mneme buffer space for the term dictionary. We found, however, that for batch sizes up to 8 Mbytes the in-memory version was no faster than the off-line version, and much worse for larger batches. The poor performance on batches larger than 8 Mbytes is due to the inverted list buffer being flushed to the index multiple times per update. The surprisingly mediocre performance for batches that fit entirely in the in-

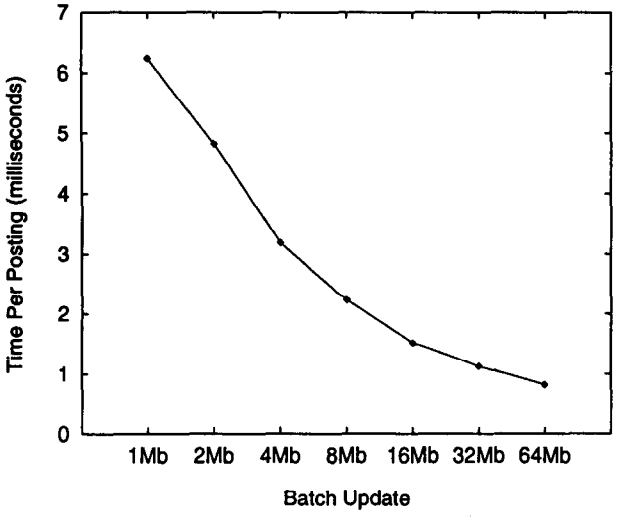

Figure 6: Update time per posting for range of batch sizes cumulatively added to TIPSTER volume 1 index.

verted list buffer is attributed to insufficient main memory. Besides the buffers mentioned above, the on-line version requires additional space to parse the documents and build the inverted lists in the main memory buffer. We suspect that the system is paging excessively, although due to a bug in OSF/1 V1.3, we could not obtain process resource usage statistics to confirm this. We did observe, however, that running the in-memory version with a 20 Mbyte inverted list buffer on a DECSystem 3000/500 with 128 Mbytes of main memory gave much improved performance. Therefore, we suggest that this technique bears further investigation and may be appropriate for certain applications with adequate resources and small updates.

4.5 Space Considerations

In addition to time, we must consider the disk space required by each indexing scheme. There are two aspects to disk space: permanent space consumed by the index files, and additional temporary space required for indexing. To reduce the amount of permanent space consumed by the inverted file, the inverted lists are compressed in a two step process. First, the location information associated with each document entry for a given term is run-length encoded, where the first location is stored as an absolute value and all subsequent locations are stored as deltas from the previous location. This yields numbers of significantly smaller magnitude. Then, all numbers (document identifiers, weights, and encoded locations) are represented in base 2 using the *minimum* number of bytes (up to four), with a continuation bit reserved in each byte. This results in variable length numbers where the largest representable number is 2^{28}.

Compression yields the same space savings for both the old and new indexing schemes. However, the old scheme allocates objects exactly of the required size, while the new scheme allocates fixed size objects for the variable length lists. Therefore, we would expect a certain amount of internal fragmentation and an increase in inverted file size for the new scheme. The inverted file created in seven batches by the new scheme for TIPSTER volume 1 consumes 473 Mbytes, of which 420 Mbytes is inverted list data. Internal object fragmentation, or free space in objects that could be allocated in the future, accounts for 50 Mbytes, or about 12% of the size of the real data. The remaining 3 Mbytes is Mneme overhead, including auxiliary file structures and links between objects, a relatively insignificant 0.8% overhead. The total overhead is quite tolerable, and other results show that as the index becomes even larger, the overhead decreases slightly due to the full utilization of all but the tail objects in the linked lists for large inverted lists.

The difference in temporary disk space requirements between the two schemes during indexing is much more significant. The transaction files generated by the old scheme for TIPSTER volume 1 consume a total of 1.7 Gbytes, or nearly 50% more disk space than the raw document collection. Each time an update is made, the transaction files for the entire collection must be generated, resulting in temporary disk space costs proportional to the size of the total collection. The new scheme only requires enough space to generate and sort the transaction file for the new documents, yielding temporary disk space requirements proportional to the size of the update. Both schemes could benefit from using suitable compression techniques on the transaction files to reduce temporary disk space requirements, but again the old scheme will still require temporary disk space proportional to the size of the existing database.

4.6 Query Processing

The last performance issue is query processing, which we evaluate by comparing query processing speeds on the inverted files built in Section 4.3. Since the new scheme must assemble large inverted lists with multiple disk reads, we expect its query processing performance to suffer. However, the allocation strategy for large lists, combined with the small number of batch updates to build the index, work to reduce this effect. Here is why. Large inverted lists allocated in a linked list of large objects are grown by adding new large objects to the tail of the list. New large objects are each created in their own physical segment, and new physical segments are allocated at the end of the file. Therefore, all objects added to a given linked list during a single batch update will be allocated sequentially at the end of the file. Since the index was built in seven batch updates, a large inverted list will consist of at most seven separate blocks of contiguous large objects, greatly reducing the disk seek costs to assemble the list.

To further explore this effect, we measured query processing performance on an index built using the on-line

scheme described in Section 4.4 with a main memory inverted list buffer of 10 Mbytes. This produced an inverted file laid out as if it had been built with many small batch updates, where each batch contained approximately 800,000 postings.

The query set we used was generated locally from *TIPSTER topics 51-100* using automatic and semi-automatic methods. We report the average of running the query set on each scheme six times, where each run differed from the average by less than 5%. The old scheme, with each list stored as a single contiguous object, required 975 seconds. The new scheme built on-line to simulate many small batches required 1256 seconds, or 28.8% longer than the old scheme. The new scheme built off-line with seven batch updates required 1033 seconds, or only 5.9% longer than the old scheme.

The results for the new index structure are surprisingly good. The improvement from the second to third scheme above indicates that query processing can be greatly improved by simple periodic reorganization of the inverted file to sequentially arrange the objects in linked lists. We expect that query processing can be further improved for the following reasons. First, the query processing model used in these experiments is "term-at-a-time", where the entire inverted list for a term is read and processed all at once. Many modern systems have adopted "document-at-a-time" processing [Wil84], which calculates the complete score for one document before proceeding to the next. In this model, the inverted lists are read in small chunks, a technique ideally suited to the linked list structure. Second, our inverted file structure might be combined with the query optimization techniques proposed by Wong and Lee [WL93] and Moffat and Zobel [MZ94b], who describe methods for eliminating processing on portions of inverted lists. Again, the linked list structure could be used to avoid I/O on these portions of the lists. Third, buffer management has not been tuned for the new file structure. Fourth, we have not considered any low level optimizations, such as overlapping I/O with processing, sophisticated pre-fetching schemes, or disk optimizations such as striping, all of which will reduce the time to assemble large inverted lists.

5 Related Work

Efficient management of full-text database indexes has received a fair amount of attention. Faloutsos [Fal85] gives an early survey of the common indexing techniques. The two techniques that seem to predominate are signature files and inverted files. Since INQUERY uses an inverted file index, we do not discuss signature files. Zobel et al. [ZMSD92] investigate the efficient implementation of an inverted file index for a full-text database system. Their focus is on compression techniques to limit the size of the inverted file index. These techniques could be usefully incorporated into our system. They also address updates to the inverted file using large fixed length disk blocks, where each block has a heap of inverted lists at the end of the block and a directory into the heap at the beginning of the block. As inverted lists grow they are rearranged in the heap or copied to other blocks with more space. Techniques for handling inverted lists larger than a disk block are not discussed, nor is the disk block technique fully evaluated. Our experience indicates that efficient management of large inverted lists is critical to performance, and we present experimental results demonstrating the effectiveness of our solution.

Tomasic et al. [TGMS94] propose a new data structure to support incremental indexing, and present a detailed simulation study over a variety of disk allocation schemes. The study is extended with a larger synthetic document collection in [STGM94], and a comparison is made with the traditional indexing technique. Their data structure manages small inverted lists in buckets (similar to the disk blocks in [ZMSD92]) and dynamically selects large inverted lists to be managed separately, not unlike our use of different object pools for different sized lists [BCCM94]. Their simulation results indicate that the best long list allocation scheme for update performance is to write the new portion of a long list in a new chunk at the end of the file. This is essentially what we do with our linked lists. However, they predict that query performance with this strategy will be poor. On the contrary, we have shown with an actual implementation that our linked list strategy can in fact provide good query performance, while simultaneously providing superior update performance. Moreover, their simulations assume that all buckets can fit in main memory during indexing, potentially requiring significant main memory resources. Our scheme makes no such assumption, requiring substantially less main memory.

Another scheme that handles large lists distinctly from small lists is proposed by Faloutsos and Jagadish [FJ92a]. In their scheme, small lists are stored as inverted lists, while large lists are stored as signature files. Again, we are primarily concerned with inverted lists and do not consider signature file solutions. In [FJ92b], Faloutsos and Jagadish examine update and storage costs for a family of long inverted list implementations, where the general case is their "HYBRID" scheme. The HYBRID scheme essentially chains together chunks of the inverted list and provides a number of parameters to control the size of the chunks and the length of the chains. At one extreme, limiting the length of a chain to one and allowing chunks to grow results in contiguous inverted lists, where relocation of the inverted list into a larger chunk is required when the current chunk is filled. At the other extreme, fixed size chunks and unlimited chain lengths give a standard linked list. Our overall scheme does not fit into this model since we initially grow chunks and then chain fixed size chunks. However, our small lists can be modeled as chains of length one where chunks are doubled in size during relocation, and our large lists can be modeled as unlimited length chains with fixed

size chunks. Rather than argue analytically, we have shown experimentally that our scheme provides good update and search costs, with acceptable space overheads.

Cutting and Pedersen [CP90] investigate optimizations for dynamic update of inverted lists managed with a B-tree. For a speed optimization, they propose accumulating postings in a main memory postings buffer, and give both analytical and experimental results. It is difficult to make comparisons with their experimental results due to the size of the collections used. We present results for a collection nearly 100 times larger. We agree that updates should be batched, but our experience with an in-memory scheme indicates that as soon as the batch becomes too large to invert in main memory (i.e., partial inverted lists for the batch must be flushed to the index before the rest of the batch can be processed), an off-line scheme will provide better performance. Again, this requires further research. They also propose storing the smallest inverted lists directly in the B-tree index. An equivalent scheme using our hash term dictionary would be advantageous only if the dictionary did not increase in size. Increasing I/O and main memory costs for the term dictionary for the sake of rarely accessed inverted lists would be disastrous.

A number of other approaches to document indexing have been proposed. Fox and Lee [FL91] describe a technique that eliminates the sorting involved in indexing by making multiple passes over the input documents. Indexing is divided into *loads*, where a load is a contiguous chunk of the final inverted file. First, an initial pass over the input is made to determine the load boundaries. Then, a subsequent pass is made for each load, processing only terms that fall within the boundaries of the load and building the inverted lists for those terms in main memory. At the end of the pass, the inverted lists for the load can simply be appended to the inverted file.

Witten et al. [WMB94] present a variety of indexing algorithms, including an extended version of Fox and Lee's algorithm. They also enhance the traditional sort-based method with compression techniques and in-place and multiway merging to greatly improve efficiency in terms of main memory, disk space, and time. Results of applying some of these techniques to the TIPSTER document collection are presented in [MZ94a].

All of these other approaches were developed for large static document collections, and do not directly support incremental indexing. Some of the techniques, such as the sorting enhancements and making multiple passes through the input to "pre-allocate" the output, might be usefully incorporated into an incremental system. Their benefit, however, would be dependent on the size of the incremental batches, with larger batches deriving more benefit. A better integration might use one of the above algorithms the first time a large collection is indexed, and switch to an incremental technique thereafter.

Properly modeling the size distribution of inverted lists is addressed by Wolfram in [Wol92a, Wol92b]. He suggests that the informetric characteristics of document databases should be taken into consideration when designing the files used by an IR system. We follow this advice, as can be seen in Section 3, with what we consider to be very satisfactory results.

6 Conclusions

If IR systems are to satisfy the demand for applications that can manage an ever increasing repository of information, they must be able to efficiently add documents to large existing collections. The main bottleneck in that operation is updating the index structure used to manage the collection. The traditional solution to this problem is to re-index the entire collection, an operation with costs proportional to the size of the whole collection. This solution is clearly unacceptable.

We have proposed an alternative solution that yields costs proportional to the size of the update. Using the data management facilities of a persistent object store, we have designed a more sophisticated inverted file index that provides fast incremental updates. More importantly, we have implemented our scheme in an operational full-text information retrieval system and verified its performance empirically.

The results we present show that our scheme maintains a nearly constant per posting update cost as the size of the collection grows, indicating excellent potential for scale. In fact, we have used our scheme to index the full 2 Gbyte TIPSTER collection in 13 batches and have found the trends described in Section 4 to hold. Our scheme requires considerably less disk space during indexing than traditional techniques, and allows much of the processing for a new batch of documents to be done independently from the existing index. This last point is particularly important for the eventual support of simultaneous query processing and collection updating, since the period of time during which index structures must be locked for updating can be minimized.

We found that best performance is achieved when documents are added in the largest batches possible, both in terms of incremental indexing time and resultant query processing speed. We have also shown that our scheme provides a good level of performance with small batch updates, and have suggested techniques to improve both small batch update and query processing performance. These techniques bear further investigation and represent future work.

Finally, we have achieved these results using "off-the-shelf" data management technology, continuing to show that the data management facilities in IR systems need not be custom built to achieve high performance.

Acknowledgements

We gratefully acknowledge Eliot Moss and the anonymous referees for their comments and suggestions for improvements.

References

[BCCM94] Eric W. Brown, James P. Callan, W. Bruce Croft, and J. Eliot B. Moss. Supporting full-text information retrieval with a persistent object store. In *Proc. of the 4th Inter. Conf. on Extending Database Technology*, pages 365–378, March 1994.

[CCH92] James P. Callan, W. Bruce Croft, and Stephen M. Harding. The INQUERY retrieval system. In *Proc. of the 3rd Inter. Conf. on Database and Expert Sys. Apps.*, September 1992.

[CP90] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proc. of the 13th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, pages 405–411, 1990.

[Fal85] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17:50–74, 1985.

[FJ92a] Christos Faloutsos and H. V. Jagadish. Hybrid index organizations for text databases. In *Proc. of the 3rd Inter. Conf. on Extending Database Technology*, pages 310–327, 1992.

[FJ92b] Christos Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In *Proc. of the 18th Inter. Conf. on VLDB*, pages 363–374, Vancouver, 1992.

[FL91] Edward A. Fox and Whay C. Lee. FAST-INV: A fast algorithm for building large inverted files. Technical Report TR-91-10, VPI&SU Department of Computer Science, Blacksburg, VA, March 1991.

[Har94] Donna Harman, editor. *The Second Text REtrieval Conference (TREC2)*. National Institute of Standards and Technology Special Publication, Gaithersburg, MD, 1994.

[Hea78] H. S. Heaps. *Information Retrieval, Computational and Theoretical Aspects*. Academic Press, Inc., New York, 1978.

[HFBYL92] Donna Harman, Edward Fox, Ricardo Baeza-Yates, and Whay Lee. Inverted files. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, chapter 3, pages 28–43. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Mos90] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, April 1990.

[MZ94a] Alistair Moffat and Justin Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Comput. J.*, 26(1):1–9, February 1994.

[MZ94b] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. Technical Report 94/2, Collaborative Information Technology Research Institute, Department of Computer Science, Royal Melbourne Institute of Technology, Australia, February 1994.

[SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[STGM94] Kurt Shoens, Anthony Tomasic, and Hector Garcia-Molina. Synthetic workload performance analysis of incremental updates. In *Proc. of the 17th Inter. ACM SIGIR Conf. on Res. and Develop. in Infor. Retr.*, Dublin, July 1994.

[TC91] Howard Turtle and W. Bruce Croft. Evaluation of an inference network-based retrieval model. *ACM Trans. Inf. Syst.*, 9(3):187–222, July 1991.

[TC92] Howard R. Turtle and W. Bruce Croft. A comparison of text retrieval models. *Comput. J.*, 35(3):279–290, 1992.

[TGMS94] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proc. of the ACM SIGMOD Inter. Conf. on Management of Data*, Minneapolis, MN, May 1994.

[Wil84] Peter Willett. A nearest neighbour search algorithm for bibliographic retrieval from multilist files. *Inform. Tech.*, 3(2):78–83, April 1984.

[WL93] Wai Yee Peter Wong and Dik Lun Lee. Implementations of partial document ranking using inverted files. *Inf. Process. & Mgmnt.*, 29(5):647–669, 1993.

[WMB94] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, New York, 1994.

[Wol92a] Dietmar Wolfram. Applying informetric characteristics of databases to IR system file design, Part I: informetric models. *Inf. Process. & Mgmnt.*, 28(1):121–133, 1992.

[Wol92b] Dietmar Wolfram. Applying informetric characteristics of databases to IR system file design, Part II: simulation comparisons. *Inf. Process. & Mgmnt.*, 28(1):135–151, 1992.

[Zip49] George Kingsley Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley Press, 1949.

[ZMSD92] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. of the 18th Inter. Conf. on VLDB*, pages 352–362, Vancouver, 1992.