Incremental Updates of Inverted Lists for Text Document Retrieval *

Short Version of Stanford University Computer Science Technical Note STAN-CS-TN -93-1

Anthony Tomasic
Stanford University†

Hector Garcia-Molina
Stanford University‡

Kurt Shoens
IBM Almaden§

December 9, 1993

Abstract

With the proliferation of the world's "information highways" a renewed interest in efficient document indexing techniques has come about. In this paper, the problem of incremental updates of inverted lists is addressed using a new dual-structure index data structure. The index dynamically separates long and short inverted lists and optimizes the retrieval, update, and storage of each type of list. To study the behavior of the index, a space of engineering trade-offs which range from optimizing update time to optimizing query performance is described. We quantitatively explore this space by using actual data and hardware in combination with a simulation of an information retrieval system. We then describe the best algorithm for a variety of criteria.

1 Introduction

As the world's "information highways" proliferate and grow in capacity, they are providing access to an ever growing number of electronic document repositories. At each repository, the number of documents available on-line is rapidly increasing. At the same time, the number of end-users with network access is rapidly growing, and a variety of tools [8] such as World Wide Web and WAIS make it possible to reach even more information sources. This rapidly growing number of documents, sites, and user queries has brought about renewed interest in efficient document indexing techniques.

The underlying index structure for most document retrieval systems is the *inverted list* [7]. The inverted list for a particular word w contains a sequence of *postings*, each reporting the occurrence of w in a document. Each posting may include a variety of information, such as the word offset (within the document) where w occurs or the region where w occurs (title, abstract, author list,

*This research was sponsored by the Advanced Research Projects Agency (ARPA) of the Department of Defense under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of ARPA, the U. S. Government or CNRI.

†Department of Computer Science, Stanford, CA 94305-2140. e-mail: tomasic@cs.stanford.edu

‡Department of Computer Science, Stanford, CA 94305-2140. e-mail: hector@cs.stanford.edu

§IBM Almaden Research Center, 650 Harry Road San Jose, CA 95120. e-mail: shoens@almaden.ibm.com

etc.) In a full text index, every word occurring in documents (minus perhaps some *stop* words) has an inverted list. As a rule of thumb, the size of the inverted lists for a full text index is roughly the same size as the text document database itself. In an *abstracts* index, only words appearing in the bibliographic information (e.g., title, abstract) have lists.

In an information retrieval system, users submit queries that consist of a set of words and some condition. The exact form of the condition varies: in a boolean system, queries are boolean expressions such as "(cat and dog) or mouse." In this example, the system would retrieve the inverted list for "cat" and "dog", intersect them, and then would union the result with the list for "mouse." The query may also give additional conditions, such as requiring that "cat" and "dog" occur within so many words of each other, or that "mouse" occur within a title region. In a vector model system, the query specifies weights for the words, and the system must locate documents that maximize the weighted sum of occurring words. Vector model systems typically use inverted lists to prune the set of candidate documents before the vector condition is evaluated.

Traditional information retrieval systems, of the type used by libraries (e.g., Stanford University's Socrates or the University of California's MELVYL) or information vendors (e.g., Dialog Inc. or Mead Data Central Inc.), assume a relatively static body of documents. Given a body of documents, these systems build the inverted list index from scratch, laying out each list sequentially and contiguously to others on disk (with no gaps). (They also built a B-tree that maps each word to the locations of its list on disk.) Periodically, e.g., every weekend, new documents would be added to the database and a brand new index would be built. Rebuilding the index is a massive operation, but its cost is amortized over multiple days of operation.

In many of today's environments, such full index reconstruction is not feasible. One reason is that text document databases are more dynamic. For instance, if one is indexing news articles, electronic mail, or stock information, the latest information is required. Thus, one would like to update the index in place, as new documents arrive. (Updating the index for each *individual* arriving document is inefficient, as we will discuss later. Instead, the goal is to batch together small numbers of documents for each in-place index update. To maintain access to the batch, it can be searched simultaneously with the larger index.)

A second reason why in-place updates are desirable is that they eliminate (or at least postpone) resource consuming reorganizations. Massive reorganizations may be acceptable in conventional systems where user load is minimal over weekends, but in today's world of 7 days a week, 24 hours a day continuous operation, degradation of service for prolonged periods is not acceptable.

A third reason why in-place updates may be desirable is that the index may simply be too massive for reorganization. As the volume of documents grows in some applications, it may be more desirable to have a dynamic index that can grow and dynamically migrate to new disk drives, without ever being fully reorganized.

In spite of the natural attractiveness of in-place index updates, very little is known about their implementation options or their performance. Systems that implement in-place updates typically use (as far as we know) relatively naive strategies that may be inefficient. For example, any time a WAIS index needs to grow an inverted list, it copies the whole list to a new disk area, leaving no free space at the end for future updates. Perhaps it would be more effective to leave some space, and to make additions that fit in that space? If multiple disks are available, can we stripe large lists across multiple disks to improve performance? Inverted lists vary tremendously in size: the ones for frequently occurring words can be huge, but there may be many that have only a few postings. What is the most effective layout of these lists to make their updates efficient? Which layouts lead to less disk space utilization? To better query performance?

Although we will not fully answer all these questions, in this paper we make the following contributions:

- A new dynamic dual-structure data structure for inverted lists. Lists are initially stored in a "short list" data structure; as they grow they migrate to a "long list" data structure. Our proposed algorithm dynamically selects lists to migrate.
- A family of disk allocation policies for long lists. Each policy dictates, among other things, where to find space for a growing list, whether to try to grow a list in place or to migrate all or parts of it, how much free space to leave at the end of a list, and how to partition a list across disks.
- A detailed performance evaluation of the dual-structure lists and the various allocation policies. The evaluation is based on a collection of 64 days worth of NetNews that are indexed according to our algorithms. Our experimental system generates the exact sequence of disk block updates that each policy produces; this sequence is then executed on an IBM Risc System 6000 Model 350 computer with 3 disks to measure the update time. Based on the resulting disk layout, we also compute disk space utilization and estimate query performance. Real disk I/O operations have the advantage of not simplifying the dynamic nature of the execution (which occurs with a simulation).

In this paper we do not consider issues related to fault tolerance. We assume that the hardware is highly reliable. However, to be fair in estimating and comparing the I/O costs of various policies, we will periodically flush to disk all the data and directory information for each policy. In addition, the algorithms and data structures are constructed so that the incremental update of the index can be restarted if it is aborted. We believe fault tolerance issues are a rich area for future research.

In the next section we describe the dynamic dual-structure for inverted lists. In Section 3 we describe a model for the various allocation policies of inverted lists that reside on disk. We evaluate these policies in an experimental design described in Section 4 and report the results of the evaluation in Section 5. In Section 6 we describe related work in the field. Finally, Section 7 concludes the paper. In addition, we have extrapolated our results to larger synthetic text document databases and describe the results in [10].

2 Dual-Structure Index

In this paper we assume that when a new document arrives it is parsed and its words are inserted into an in-memory inverted index. At some point the in-memory inverted index must be written to disk. Collecting many documents into an in-memory inverted index before writing the index to disk amortizes the cost of storing a posting. Our objective is to incrementally update the disk with the in-memory inverted index as efficiently as possible.

The lengths of the inverted lists for a database of text documents have a roughly exponential distribution (the Zipf curve [11]). This presents a dilemma for the in-place update of inverted

Text Document Database	NEWS
Total Raw Text	686 MB
Total Words	788,256
Total Postings	48,526,577
Documents	138,578
Average Postings per Word	61
Frequent Words	39,413
Infrequent Words	748,843
Postings for Frequent Words	93.6%
Postings for Infrequent Words	6.4%

Table 1: Statistics for a NEWS abstracts text database. Abstracts databases index general information about a document such as author names, title, the set of words in the abstract, etc. A frequent word for this table ranks in the top 5% of all words (in order of frequency). Postings for frequent words are given as the percentage of all postings in the database. Infrequent words are all words that are not frequent.

lists since some inverted lists (corresponding to frequently appearing words) will expand rapidly with the arrival of new documents while others (corresponding to infrequently appearing words) will expand slowly or not at all. In addition, new documents will contain previously unseen words. Table 1 shows some statistical properties of a database of NEWS articles (cf. Section 4 for a complete description of the database). For example, if we consider frequently appearing words as those that rank in the top 5.0% of all words (in order of frequency) we see that the postings for the frequently appearing words account for the vast majority (93.6%) of the postings.

In our scheme there are two data structures for lists. We place short inverted lists (of infrequently appearing words) in a fixed size region of disk where the region contains postings for multiple words. These lists are referred to as *short lists* and the fixed size regions are known as *buckets*. The idea is that every inverted list starts off as a short list; when a bucket fills up with inverted lists, the longest inverted list becomes a long list. We place the long inverted lists (of frequently appearing words) in variable length contiguous sequences of blocks on disk. We refer to these inverted lists as *long lists*. Each block of a long list contains postings for only one word. Given a word w, we examine a *directory* which determines if the word has a long inverted list. If the word does not have a long inverted list, it has a short inverted list or no inverted list at all. In this case, a function $h(w)$ (e.g., a hash function or a tree search) returns the bucket where the short inverted list, if any, for the word is stored.

At some point, an in-memory list L for word w (generated from arriving documents) must be moved to disk. First, if w already has a long list (on disk), L is appended to the long list as discussed in the next section. Otherwise, we assume L is a short list and insert it into bucket $h(w)$. If the bucket is not already in memory, it is read in, and L inserted. (If a list for w already existed in the bucket, L is added to it; else a new short list is created in the bucket.) If the bucket overflows, we then pick the longest short list[1] in block, say M, remove it, and make M a long list. Once M is removed, the bucket will be partially empty. The updated bucket $h(w)$ is written to

[1]If there are multiple longest short lists, we choose on arbitrarily.

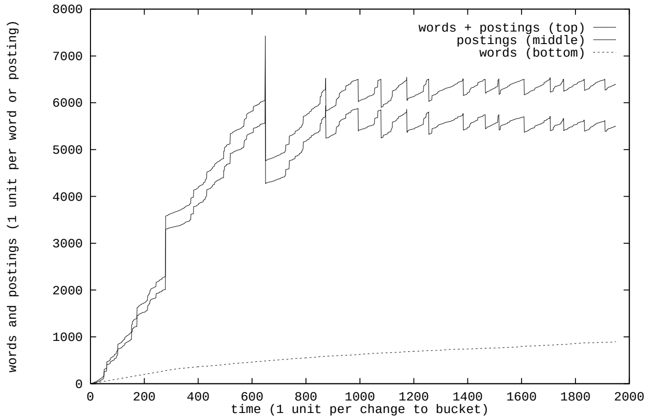

Figure 1: An animation of the behavior of bucket 0 for the first 6 updates for a system with 250 buckets. The top line is words + postings, the middle line is postings, and the bottom line is words.

disk (eventually), and list M is written to disk as discussed in the next section. Note that a word w never has both a short list and a long list associated with it. The buckets dynamically determine which words have inverted lists containing only a few postings, since these words are unlikely to grow enough to overflow into a long list. (Assuming that the bucket data structure is large enough to hold all the infrequent words.)

Figure 1 shows an animation of the behavior of buckets. We choose bucket 0 as an example bucket and run the bucket algorithm for a short time on a small system. Data is from a NEWS, as explained in Section 4. The bucket has a size of 6500 units, where each posting is charged 1 unit and each word is charged one unit too. (For each inverted list in the bucket, we need to store the word it represents plus all of its postings.) Each time step on the x-axis corresponds to a change in the bucket – the insertion of a new word with its postings, the appending of postings to an existing word, or the removal of a word and its postings from the bucket. The y-axis measures the combined number of words and postings in the bucket. The top line in the figure is the total number of words and postings, the middle line is the total number of postings and the bottom line is the total number of words in the bucket. For the total number of words in the bucket, we see a slow rise in the number of words in the bucket as new words are continuously inserted into the bucket. For the number of postings in the bucket (the middle line), we see a steep climb as the bucket fills up and two leaps where a very long in-memory list is inserted into the bucket at approximately time 300 and time 700. The second insertion of the long in-memory list causes an overflow and the list is removed from the bucket, shown as a downward spike in the graph. After the spike, the bucket continues to fill to about time 900 where it overflows and again the longest short list is removed from the bucket.

In summary, the dual-structure index allows us to apply different storage structures to the huge number of infrequent words and to the relatively few frequent words. Through the use of fixed-size buckets, this approach dynamically discovers the frequent words that require their own long list. Updates to the large number of infrequent words are amortized into a relatively small number of disk operations, since the buckets are small enough to fit in memory. In addition, coalescing infrequent words reduces wasted disk space due to allocation of complete disk blocks to very short lists.

3 Policies for Allocation of Long Lists

In this section we present policies for the allocation of long lists to disk. Implementations of IR systems indexes merge inverted lists to compute the answer to a boolean query. This is possible because the document identifiers appear in sorted order in inverted lists. We assume that new documents are numbered with identifiers in increasing order and that all long lists are updated by appending new postings to them. With these assumptions, the merge operation can be used to compute answers to boolean queries with our long list data structure.

Long lists are created initially by the overflow of a bucket. Once a word has a long list on disk, subsequent in-memory lists for that word will be appended to that long list. In this section we consider only long lists and refer to them by the shorthand *lists*.

In allocating lists to disk, there are two extremes policies. One extreme policy optimizes the time to incrementally update. Let L be the list for a word w and let M be the in-memory list to be appended to L. If M is written to disk sequentially at the current end of the data on disk, irrespective of L, then update performance is optimized because the disk head never seeks during a sequence of updates. The other extreme policy optimizes the reading of a list during query processing. To update a word w, we can read L from disk, append M to it, and write the new combined list to a new location on disk. This optimizes query performance because exactly one seek is required to read any list. However, query performance for the update optimized policy is poor (since the list for a word will be spread over the disk) and the update performance for the query optimized policy is poor (since reads are intermixed with writes).

Between these two extreme policies, there are many intermediate ones that move lists and allocate new space for them in a variety of ways. In this section we present a framework for describing these intermediate policies. Obviously many more choices exist than what we will describe here, but we cover most of the interesting choices.

The first issue is to compose lists from disk blocks. We use the term *chunk* for variable sized contiguous regions of disk and reserve the term *extent* for fixed sized contiguous regions of disk. (We also study the extent case later.) Multiple chunks for an inverted list may be allocated. The pointers to all chunks are recorded in the directory. The directory entries for a word may point to chunks on multiple disks. The directory resides in memory at all times. Periodically, the directory is written to disk.

The second issue is to assign a disk unit to a new word or chunk. When the list for a new word w is added to the directory or a new chunk of a list for a word w is allocated, a disk is chosen. Let there be n disks, numbered from 0 to $n-1$, and let i be the disk chosen when the last new word or chunk was allocated (i is initially 0). The strategy considered here is to chose disk is $i+1 \bmod n$. (Other strategies could be to look for the most empty disk or a disk where the list has the fewest chunks. These strategies are not considered in this paper to keep the space of possible solutions manageable.)

The third issue is to combine in-memory lists with long lists. We have an in-memory list M that we wish to append to a long list L. Both lists are for a word w. Let x be the size (in postings) of the long list, let y be the size (in postings) of the in-memory list, and let z be the size (in

Variable	Value	Meaning
Limit	0	Never update in-place
	z	Update in-place if enough space
Style	fill ($e = 3$)	Fill in fixed size extents
	new	Write a new chunk when appropriate
	whole	Long lists are single whole chunks
Alloc	constant ($k = 10$)	Constant extra postings reserved
	block ($k = 3$)	Multiple of a fixed sized block reserved
	proportional ($k = 1.1$)	Proportional extra postings reserved

Table 2: The variables and values that determine a policy for the allocation of long inverted lists to disks. The values in parenthesis are for each allocation strategy or style.

postings) of the space remaining in the chunk which can accommodate new postings. As described below, when a chunk is allocated to disk, it may have *reserved space* at the end of the chunk where future postings may be append. Note that z may be zero or positive and that x and y are always positive. Our strategies for appending will call on the following basic operations which operate with respect to long list L. The RELEASE list is used to delay the deallocation of long lists while they are copied.

UPDATE(a) reads the last block containing postings for word w of in-memory list a, appends a to it, and then writes the result back as an in-place update.

b **:= READ**(a) reads all the postings for long list a, places a on the RELEASE list, and returns the postings read as in-memory list b.

WRITE(a,b) writes up to e blocks worth of postings from in-memory list a and returns the remaining postings as in-memory list b. The global parameter e is called the *extent size.* (The fill style, below, breaks up in-memory lists into extent size chunks.) If a contains less than e blocks worth of postings, e blocks are still allocated on disk.

WRITE RESERVED(a) writes the a in-memory list to disk with reserved space at the end of the list (see fourth issue below).

A strategy for appending an in-memory to a long lists is specified by two variables, *Limit* and *Style*. *Limit* is either 0 or z. *Style* is *fill*, *new*, or *whole*. Table 2 summaries the variables and values governing policies.

Figure 2 shows the algorithm for updating long lists. The first three lines check if the existing chunk can and should be extended with the in-memory postings. If this isn't possible or desirable (*Limit* $= 0$), the fourth through sixth lines (whole) copy the old postings to a new location with the in-memory postings appended. Lines seven, eight and nine (fill) write out multiple extents. Lines ten and eleven (new) write a new chunk with reserved space. One consequence from lines one and two is that an in-memory inverted list is never split into two different chunks for an in-place update.

```
if y ≤ Limit then
   UPDATE(M) update long list in-place with the in-memory list
else
   if Style = whole then
      b := READ(L) read long list
      WRITE RESERVED(M and b) append in-memory list and write with reserved space
   if Style = fill then
      WHILE (M not empty) in-memory postings remain
              WRITE(M, M) write in-memory postings
   if Style = new then
      WRITE RESERVED(M) write in-memory postings with reserved space
```

Figure 2: The algorithm for updating long lists.

Periodically, the buckets and the directory are written to disk. At this time, the disk blocks for the previous buckets and directory are returned to free space for the disks. In addition, in the case of the whole strategy, the old long lists on the RELEASE list are returned to free space for the disks.

The fourth and final issue is to allocate space on a disk for a chunk. Given a request for a chunk of size f and a disk, we need a contiguous region of free space on the disk to satisfy the request (this is similar to the problem of free space allocation for blocks in a file system). We use a first-fit strategy by scanning the free list for the disk from the beginning of the disk. Upon finding a contiguous sequence of f or more blocks, the chunk is placed at the beginning of the free blocks and the remaining free blocks are returned to free space. (Other strategies could be to best-fit or to use a buddy system. These strategies are not considered in this paper to keep the space of possible solutions manageable.)

In addition, the WRITE RESERVED call reserves space at the end of every list for future growth. That is, additional space is allocated to a chunk to hold postings which will appear in subsequent updates. Let x be the size (in postings) of the inverted list being written to disk and let $f(x)$ be the allocated space (list plus reserved space). The resulting size (in blocks) of a chunk is the number of blocks needed to hold $f(x)$. For the new style x is typically the size of an in-memory list. For the whole style x is typically the size of the entire long list for a word.

We consider three choices for the definition of $f(x)$. The *constant* strategy adds a constant number k of postings to the end of the inverted list, i.e., $f(x) = x + k$. The *block* strategy insures the chunk is of constant multiple of size k, i.e., $f(x) = k \cdot \lceil \frac{x}{k} \rceil$. (In practice, we specify k for this strategy in terms of blocks instead of postings.) Finally, the *proportional* strategy allocates a chunk in proportion to the number of postings being written to disk, i.e., $f(x) = kx$. The variable *Alloc* equals *constant*, *block*, or *proportional*, for the corresponding choice of strategy.

3.1 Policies

A *policy* is determined by the values of the variables *Style*, *Limit* and sometimes *Alloc*. If *Limit* = 0, then any reserved space for a chunk is never used, so we automatically set *Alloc* = *constant* with $k = 0$. If $Style = fill$ then the allocation strategy is irrelevant since it is never considered.

Consider the update optimized policy described earlier. This can be achieved by setting $Limit = 0$ and $Style = new$. This policy minimizes update time by simply writing out the update list blocks as fast as possible. No reading is done because no updates in-place occur. We expect that this policy will have the best update time and that the query time for the resulting index will be poor.

Consider the policy for fast queries. Let $Limit = 0$ and $Style = whole$. Setting $Style$ to $whole$ insures that the inverted list for any word will always be a single contiguous chunk and thus query time is minimized. We expect that the update time for this organization will be high, due to the amount of moving of lists that must be done. To ameliorate this situation, we can let $Limit = z$ and $Alloc = proportional$ with a constant of, say, 1.1 (i.e., reserved space that is 10% of the size of the long list). With each move of the long list, the reserved space will grown by 10%, permitting more and more in-place updates of in-memory lists.

Finally, we consider a policy that attempts a trade-off: to minimize query time and keep the cost of updates low by organizing inverted lists into chunks that never move once they are full. Let $Limit = z$ and $Style = fill$ with an extent e size, say, 3 (blocks). With this policy, each inverted list will grow until it reaches the limit of its chunk and then a new chunk will be started on a new disk. We expect comparatively good query and update times for this policy. (Note that our model of extents uses only one size for an extent. We do not model multiple fixed extent sizes since this policy is approximated by the new style with a block allocation strategy.)

So far our discussion has focused on the addition of documents to an index since typically databases only grow in size, or deletion is infrequent enough that the entire index is rebuilt. The addition of incremental deletion of documents poses some problems to the design of an index. One method maintains an index of document identifiers and all the words in the document (or the words are extracted from the original document). Given this index, each inverted list for a word in the document would be fetched, the reference to the document deleted, and the new inverted list rewritten to disk. However, the size of this index is the same as the size of the inverted index. To avoid this cost, existing implementations typically maintain a list of deleted document identifiers and filter any answer to a query through this list. This deletes the document from the point of view of the user since a deleted document identifier will never appear in an answer. To reclaim the space taken by the deleted document identifiers in the index, a background process sweeps the lists in the index one list at a time, removing any deleted documents. After a sweep of the index, the list of delete document identifiers can be thrown away. Since this issue is orthogonal to the the issues in this paper, we do not consider deletion further.

In summary, the parameters described in this section span an interesting range of approaches to storing long lists. By varying these parameters, we can model schemes that keep the lists sequential and those that break the lists into contiguous chunks. We can control the size of the chunks allocated, either as fixed-length chunks or as chunks whose size is controlled by the frequency of a word. Finally, we can control whether unused space at the end of a list is filled in or not. The choice of parameters permits trade-offs between index build performance, query performance, and index disk space consumption.

4 Experiment Design

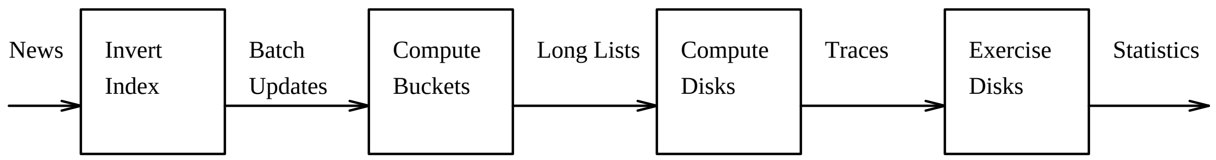

Figure 3: The flow of data for the experiment design. Arrows represent data. Boxes represent the transformation of data via a process.

```
for years. And it was a total flop, in all the years it was available
very few people ever took advantage of it so it was dropped.
```

(a)

```
a advantage all and available dropped ever few flop for in it of
people so the took total very was years
```

(b)

Figure 4: (a) A fragment of a document from November 18th, 1992; (b) the tokens (in sorted order) resulting from the document fragment.

Figure 3 diagrams the flow of data for our experiments in building inverted indexes. Each arrow represents a data set and each box represents a process that transforms data sets. The diagram also serves as an outline for this section and we describe each part of the diagram in turn.

4.1 NEWS

The source text document database is 64 days of NEWS articles gathered from November 18th, 1992, to January 21st, 1993. (December 10th is missing.) See Table 1 for statistics on the database. Once per day the the local server was scanned for new documents. NEWS documents less than 2560 characters in length were eliminated to increase the average document size to a more typical range of about 5K characters [2]. Also, non-English language documents (e.g., encoded binaries and pictures) were filtered out.

Each day of documents is a *batch* and is processed separately from other days. (We do not consider document deletions in this study.) While the dual-structure index does not require periodic updates, this arrangement is good for measuring activity at periodic intervals.

4.2 Invert Index

The *invert index* process accepts a sequence of document batches as input, processes them, and

abandons 1	abashed 2	abate 1
abated 1	abatement 1	abb 2

Table 3: A part of the batch update for November 18th, 1992, shown as pairs of words and the number of documents the word occurs in.

Variable	Value	Description
$Buckets$	1,500	Number of buckets
$BucketSize$	6,500	Size of bucket
$BucketTotal$	9.75 M	$Buckets \cdot BucketSize$
$BlockPosting$	683	Postings per Block
$Disks$	3	Number of Disks
$BlockSize$	4,096	Bytes per Block
$BufferBlock$	400	I/O buffer memory

Table 4: The experimental parameters and base case values.

```
0 0
125144 4746
133663 4123
180761 4084
124376 3637
313269 4567
```

Figure 5: A part of the output of the compute buckets process. Each line is a word-occurrence pair. The left column contains integers representing words. (Words are numbered alphabetically.) The right column contains the lengths of the corresponding in-memory lists. The line "0 0" indicates the end of a batch update.

generates a *batch update* for each batch. A batch update contains a list of words that appear in the documents of the batch and the number of times each word occurs in the batch. A word and its frequency of occurrence is termed a *word-occurrence pair.*

To generate a batch update, each document in the batch is lexically analyzed to produce a token stream. Sequences of letters and sequences of number are tokens – all other characters are ignored. Certain lines of a document (such as "Date: " lines) are also ignored. Finally, duplicate tokens for a document are dropped. After all documents for a batch are reduced to sets of tokens, an inverted file is constructed for the batch. Tokens are converted to words by converting upper case letters to lower case. The batch update containing all the words and the lengths of the inverted lists for each word is then constructed. Figure 4a shows a fragment of a document and Figure 4b shows the resulting set of tokens from the document fragment. Table 3 shows a part of a batch update. Note that the misspellings of words are part of the batch update as well. At this point all words in batch updates are converted to unique integers to simplify the remaining computations.

An implementation of an information retrieval system proceeds in the same way we have described here, except that it would keep, for each word, its complete inverted list, as opposed to the simple word-occurrence pair we keep here. For our performance evaluation, we do not need to know the contents of each inverted list, only its size, which is what the word-occurrence pair gives us. Note, thus, that our batch update is our representation of the in-memory index of Section 2.

4.3 Compute Buckets

```
update bucket disk 0 id 0 size 1587
update bucket disk 1 id 0 size 1587
update bucket disk 2 id 0 size 1587
update chunk disk 0 id 0 size 0
write word 125144 posting 4746 disk 1 id 1587 size 7
write word 133663 posting 4123 disk 2 id 1587 size 7
write word 180761 posting 4084 disk 0 id 1587 size 6
write word 124376 posting 3637 disk 1 id 1594 size 6
write word 313269 posting 4567 disk 2 id 1594 size 7
```

Figure 6: An I/O trace corresponding to the previous figure.

The *compute buckets* process takes the sequence of batch updates as inputs, runs the bucket algorithm described in Section 2 on the sequence (we use a modular arithmetic hash function for $h(w)$), and generates a single trace file of updates to long lists. Each update in the file indicates the word involved, and the number of postings to be add to the corresponding long list on disk. (Note that the postings for an update can comes from the new postings in a batch or from previous postings in a bucket.) In addition, a marker for the end of each batch update is added to the trace. Figure 5 shows a part of the output of the compute buckets process. For instance, in the second line of this figure, the number 125,144 is the unique identifier for a particular word and the number 4,746 is the number of postings to be appended to the long list for that word.

Table 4 lists the variables that control the bucket computation and the base values used for those variables in the experiments reported in the next section. Variable *Buckets* records the number of buckets and variable *BucketSize* records the size of each bucket (we count 1 for each word and posting placed in a bucket). Variable *BucketSize* implicitly models the efficiency of the compression algorithm applied to in-memory inverted lists since computations are in terms of postings instead of bytes. The remaining variables in this table we describe in the next section.

In comparing the compute bucket process and an implementation of the bucket data structure in an information retrieval system, we note that an implementation would perform a similar computation using inverted-lists as the compute bucket process does using word-occurrence pairs. A implementation would produce the same set of long lists. We assume that during the update process the buckets are kept in memory since they are referenced much more frequently than the long lists. At the end of each batch update, all buckets are flushed to disk. Note that the cost of maintaining all the buckets in memory during the update process can be avoided by sorting the in-memory lists into bucket order and then merging the in-memory list with the buckets, requiring only one bucket to be in memory at any single point in time.

4.4 Compute Disks

The *compute disks* process takes as input the trace file of long list updates and computes the sequence of I/O systems calls required to implement the policies described in Section 3. In addition, the write operations for saving the buckets and the directory are added at the end of each batch update. Figure 6 shows a sample of a I/O trace file. The first three lines indicate that the write of the bucket data structure occurs on three disks starting at location 0 and continuing for 1587 blocks. The next line writes an empty directory. (The directory is empty because this is the beginning of the trace, i.e., no long lists have been written to disk – no actual I/O is performed for this line). The following lines write inverted lists for each word. For instance, the first "write word" line indicates that word 125,144 writes 4,746 postings on disk 1 starting at block 1,587 for a size of 7 blocks.

In addition to Table 2, Table 4 lists the variables and values used for the compute disks process. Note that the variables $BlockPosting$ and $BlockSize$ implicitly model the efficiency of the compression algorithm applied to long lists. A disk trace corresponds closely to the sequence of system I/O calls an implementation would perform for a given policy.

4.5 Exercise Disks

The *exercise disks* process takes a trace of I/O operations as input and executes it on an IBM RS 6000 Model 350 computer (64 MB memory, UNIX AIX 3.2 operating system with 3 disks (Seagate ST41200NM, 1 GB capacity, 5.25 inch, SCSI-1 standard) and an I/O bus (SCSI-1 standard). Each line of the trace generates a read or write system call request and after the update of the buckets and the directory all system buffers are flushed to disk.

Requests to each disk are issued by independent processes to achieve maximum parallelism. Request are directed to "raw" partitions of the disk, bypassing the operating system's file system and disk buffer pool. Our algorithms do not revisit the same blocks within a single batch, thus eliminating the advantage of a buffer pool. In addition, we assume that relevant data will not remain in buffers from one batch update to the next. Furthermore, bypassing the file system saves CPU overhead and results in slightly superior data rates. Finally, bypassing the operating system isolates our experiment design from effects introduced by the file system and thus experiments are independent of any particular file system implementation.

One drawback to using raw disk partitions is that the operating system obeys the disk requests exactly and does not coalesce adjacent write requests into single disk I/O operation. For this reason, the disk exerciser program does its own coalescing of I/O operations where possible without reordering the execution trace. To be faithful to real systems with a finite amount of buffering, the disk exerciser will only coalesce up to *BufferBlock* blocks (each of size $BlockSize$) in a single request.

In summary, the experimental design presented here has many advantages. One of the most important is the decoupling of each process from the subsequent process, which permits varying parameters of a process to study the effects on the corresponding data transformation. However, the design rests on the assumption that the CPU costs of each process do not dominate the total computation time. To test this assumption, we tested an actual running IR system. We selected the RUFUS system [9] and built an inverted index for 307 megabytes of documents from a collection of IBM internal bulletin board articles. Our experiments [10] confirm that I/O time dominates the building of inverted indexes.

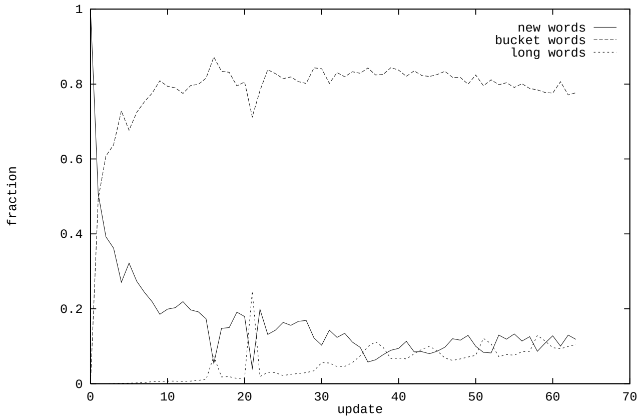

Figure 7: The fraction of words per update in each category.

5 Results

In this section we describe results for a set of experiments. An *experiment* is the execution of the sequence of processes described in Section 4 over the 64 days of collected data[2] using the defaults values given in Table 2 and Table 4, except as otherwise noted. The values of variables are sometimes systematically varied to show the sensitivity of a variable to some measurement. An *update* refers to the incremental batch update of the index. Some measurements apply only to the update. Other measurements apply to the index which results from the sequence of updates. In this case we refer to the *index after update*. For this section, the *final index* is the index which is produced after all the updates have been processed. We describe the results for each process in turn.

5.1 Compute Buckets

In this section we show the behavior of buckets and the generation of long lists. The issue of tuning the size of the buckets and the number of buckets is a complex issue in itself. We describe in detail the impact of tuning buckets and our approach in [10]. For the issues presented in this paper, essentially the tuning of the bucket data structure *uniformly* affects the results presented here. Thus, while there are quantitative changes in the behavior of the system, the qualitative differences remain the same.

To show the behavior of the buckets, we measure the number of long lists in an update. For each word-occurrence pair in an update, we can categorize the word of the pair as one of three types: a *new* word (a previously unseen word), a *bucket* word (a word that is already in a bucket), or a *long* word (a word that has a long list).

Figure 7 shows, for each update, the fraction of words belonging to each category. Observing the "new words" curve, we see that initially all word-occurrence pairs contain new words since the buckets are empty and there are no long lists. This behavior very rapidly drops off as the buckets fill up with frequently appearing words. Eventually, after about 40 updates, the fraction

[2]We explore larger data sets in [10]

of new words per update stabilizes around 10%. Observing the "bucket words" curve, we see that the fraction of words in an update which are in buckets rapidly rises until about the 15th update. Since the majority of words are the same in every update, this rise indicates that the buckets are filling up with words. The curve declines (roughly linearly) after update 40 as new words (containing typically short in-memory lists) fill the buckets and cause words to overflow into long lists. Observing the "long words" curve, we see that initially, no word-occurrence pairs contain long words because a few initial updates fit into the bucket data structure. The fraction of long lists rises (roughly linearly) after the buckets fill up in the initial stage. (The spike at update 21 is due to a very small size of the update for that day introduced by an interruption in the gathering of data.) Finally, we note a periodic set of peaks spaced seven days apart on the "long words" curve. Each peak corresponds to a Saturday when the corresponding update is smallest for the week. Small updates have higher fractions of frequently appearing words.

5.2 Compute Disks

In this section we consider the effects of the various allocation strategies described in Section 3. As our unit of measurement for this section only we count I/O operations. Each I/O operation corresponds to a call to the operating system that results in a disk seek and transfer of information. Note however that counting I/O operations is only an estimate of the time taken for a sequence of I/O operations. We consider the actual time taken for I/O operations when the exercise disk results are presented in Section 5.3. We study I/O operations in addition to actual times because they provide insights into the behavior of the long list policies, because a wider range of parameters can be studied simply because each experiment takes less time, and because I/O operations closely estimate actual times. To compare allocation strategies, first we compare the three styles with zero reserved space to study the effect of in-place updates with respect to index build time, disk space utilization and the query performance of the resulting long lists. Then allocation strategies are added to study the effects of reserved space for the same issues.

5.2.1 Styles

The number of I/O operations needed for each of the three policies is shown in Figure 8. In the case that *Limit* $= z$, we use *Alloc* $=$ *constant* with a constant of 0. This removes the effect of the allocation policies. However, in-place updates are still possible by filling the empty space in the block(s) at the end of the list. The x-axis is the index after the given update. The y-axis is the *cumulative* number of I/O operations needed to incrementally build the index.

The first observation is that all the curves in the graph have increasing slope. This means that the time to run each update takes longer as the index grows in size. This is due to the increasing number of long lists. Second, the bottom two lines have *Limit* $= 0$ and the next two lines have *Limit* $= z$ for the new and fill styles. This means that in-place updates *double* the number of I/O operations required. This is due to each in-place update needing a read and a write operation. The graph shows that the whole style requires more I/O operations than either the fill or new style, regardless of the use of in-place updates. Since the whole style costs one read and one write operation for each append of an in-memory list to a long list, whether an in-place update occurs or not, the whole style is the upper bound in number of I/O operations for any style. From the

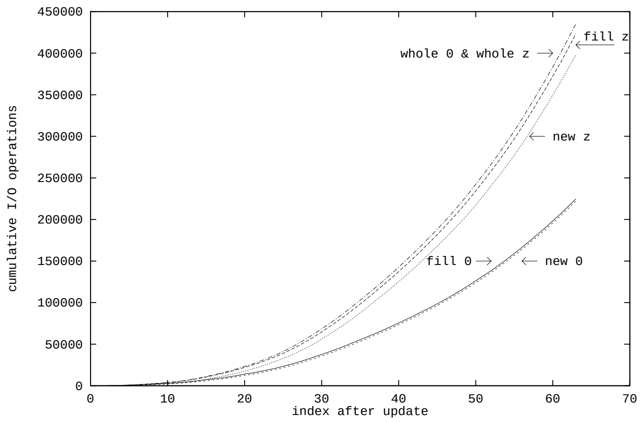

Figure 8: The *cumulative* number of I/O operations needed to build the final index. The x-axis is the index after the given update. The y-axis is the cumulative number of I/O operations needed to incrementally build the index. Each curve is label with the values of *Style* and *Limit*.

figure, we see that value for the final index for the whole style and for the fill and new styles with in-place updates are within 10% of each other. Thus, these policies use approximately the same time to build the final index.

Another measure of performance of a style is the long list utilization rate, namely the fraction of space allocated in long lists disk blocks that have postings. Thus we measure the *internal* utilization of the long lists. (The amount of *external* fragmentation is a consequence of the strategy taken for managing the free blocks on disk (cf. Section 3) which we do not study here.) Figure 9 shows the long list utilization rate for the index, measured at the end of each update, for the same set of policies as the previous figure. The spike for all curves between update 0 and 1 is due to the utilization rate of 0.0 when there are no long lists. We see that that utilization without in-place updates for the new and fill styles falls dramatically. This is due to the large amount of wasted space for small in-memory lists for these styles. Adding in-place updates to the new and fill style permits blocks to be more efficiently utilized as shown by the figure. The whole style has good utilization regardless of in-place updates since each list is stored contiguously.

Comparing the I/O performance of policies to the corresponding utilization rates, we see that the two best performing policies in terms of I/O performance are unrealistic due to the resulting extremely poor utilization rates. Thus, the doubling of the I/O operations for update cannot realistically be avoided. In choosing among the remaining alternatives, if update performance is crucial then the new style with in-place updates is best, and if utilization is crucial than either of the whole policies is best.

Measuring query performance for a policy is difficult since the typical workload depends on the information retrieval model (IRM). For a typical boolean IRM, a query contains a few words (less than 50) and the words tend to be the less frequently appearing words since frequently appearing words do not discriminate strongly between documents. Thus we would expect many query words to reside in buckets for this model. For a typical vector space IRM, a query may be derived from a document, consequently the query often contains many words (more than 100) and the words tend to be frequently appearing words. We concentrate on the vector space IRM for this paper (see [10]

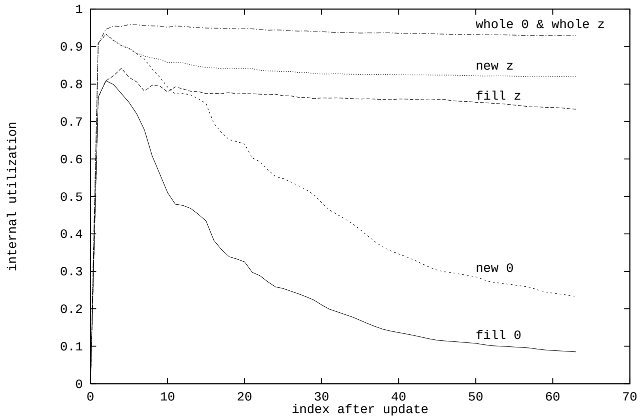

Figure 9: The long list disk utilization of the various policies. The x-axis is the database after the update. The y-axis is the fraction of space in the long lists which contain postings. Each curve is label with the values of *Style* and *Limit*.

for results on the boolean IRM).

For a vector space IRM, we assume the distribution of words in such a query to approximate the frequency of words in documents. To measure query performance, we measure at the end of each update the average number of read operations needed to read a long word. This is computed by counting the total number of chunks in the index and dividing by the number of words with long lists. Figure 10 graphs this result for the various policies. The graph shows that in-place updates are need for competitive query performance for the new and fill styles. In the final index, the whole style performs about 2.8 times better than the fill style with in-place updates and about 5 times better than the new style with in-place updates for this metric.

5.2.2 Allocation Strategies

There are several issues to consider with the allocation strategies. How is the constant value for an allocation strategy selected? Given an allocation constant, is there some rule to select its constant, independent of a policy? Given any style, is one of the allocation strategies best? Let us start by focusing on a particular style, say the new style. (We assume in-place updates since allocation strategies are not otherwise used.)

For the new style, as the amount of reserved space for each list rises (by increasing k), the number of in-place updates rises and behavior converges towards a style where most updates long lists are in-place. In addition, as the amount of reserved space rises the disk utilization falls and the average number of reads for a long list approaches 1. This presents a classical trade-off between disk utilization and query performance. Experiments described in [10] describe this trade-off in more detail. In-place updates also increase the update time for the new style, but the range of update times for in-place updates is only 10% in terms of I/O operations (cf. Figure 8). Thus, allocation strategies can only have a small impact on update time.

Table 5 compares various allocation strategies and constants for the new style. The "Read" column is the average number of read operations required to read a long list. The "Util" column is the internal utilization of the long lists. The "In-place" column is the total number of in-place

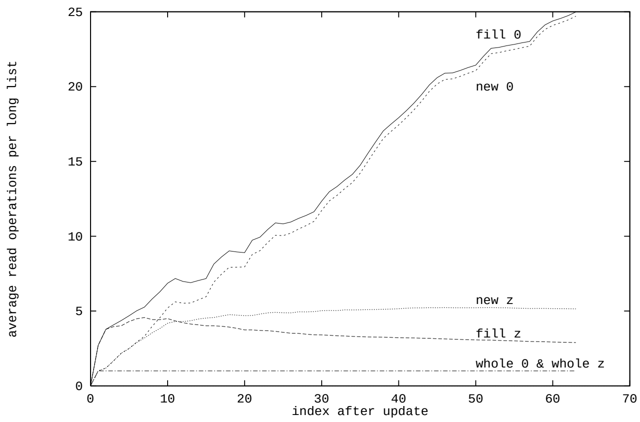

Figure 10: The average number of read operations to read a word with a long list. The x-axis is the database after the given update. The y-axis is the average number of read operations to read a long list.

New Style					
Allocation	k	Read	Util	In-place	Frac
constant	500	3.60	0.78	189865	0.89
constant	1000	2.66	0.73	198292	0.93
block	2	3.36	0.78	192059	0.90
block	3	2.61	0.73	198746	0.93
proportional	2.0	2.89	0.80	196271	0.92
proportional	3.0	1.88	0.73	205316	0.96

Table 5: A comparison of allocation strategies with respect to the final index for the new styles. The meaning of each column is described in the text.

Whole Style				
Allocation	k	Util	In-place	Frac
constant	0	0.93	179603	0.84
constant	500	0.90	188836	0.89
constant	1000	0.82	198309	0.93
block	2	0.87	194003	0.91
block	3	0.80	200156	0.94
block	5	0.68	205735	0.96
proportional	1.1	0.90	193695	0.91
proportional	1.25	0.85	202087	0.95
proportional	2.0	0.67	209994	0.98

Table 6: A comparison of allocation strategies with respect to the final index for the whole style. The table is describe in the body of the text.

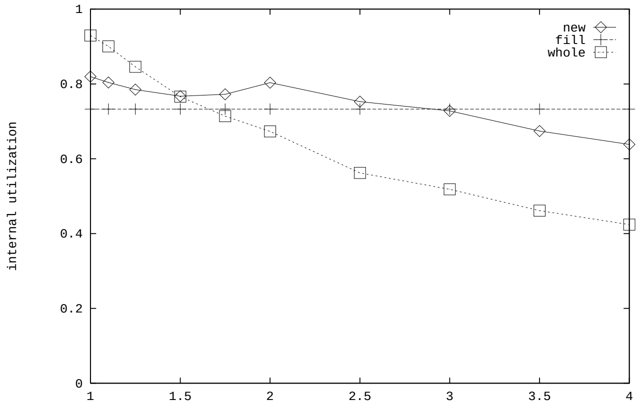

Figure 11: The impact of the constant k for the proportional allocation strategy on the utilization of long lists in the final index. The fill style (with the extent allocation strategy with extent size 3) is include for comparison.

updates needed to incrementally build the final index. The "Frac" column is the fraction of in-place updates of the total possible number of in-place updates. (The total possible number of in-place updates which are possible is 213,256.) (The "In-place" and "Fraction" columns are included only for comparisons with Table 6.) The constant value for each strategy was chosen by increasing it until long list utilization was at 73%. This utilization rate was chosen since it offered good read performance, which was not available at higher long list utilization rates. Some additional values of interest are also included in the table. The table suggests (and other results not shown here confirm) that the new style with a proportional allocation strategy offers the best trade-off by having the best read performance at this level of utilization.

Let us now turn our attention to the whole style. There is also a space-time trade-off for the allocation strategies for this style. The space trade-off is the utilization of long lists (as for the new style) but the time trade-off is only update time, not query performance, since all allocation strategies offer the same query performance. To compare update time, we cannot count I/O operations since this measure not distinguish between reading the tail of a list to append an in-memory list and reading the entire list. To account for this, we directly compare the number of in-place updates for each allocation strategy.

Table 6 shows statistics for various allocation strategies. The number of read operations for a long list is always 1.0 with the whole style. The "Util" column is the internal long list utilization. The "In-place" column is the total number of in-place updates needed to incrementally build the final index. The "Frac" column is the fraction of in-place updates of the total possible number of in-place updates. (The total possible number of in-place updates which are possible is 213,256.) The table shows that the proportional allocation strategy is the best overall strategy since it is the only strategy to offer at least 90% for both utilization and the fraction of in-place updates.

Turning to the fill style, recall it has its own extent allocation strategy. The same space-time trade-off as with the new strategy exists. So as the number of extents e is increased, disk utilization falls and query performance improves. We conducted the same analysis for this style as for the others, increasing e until utilization falls to 73%. Our experiments show that a value of 3 for e

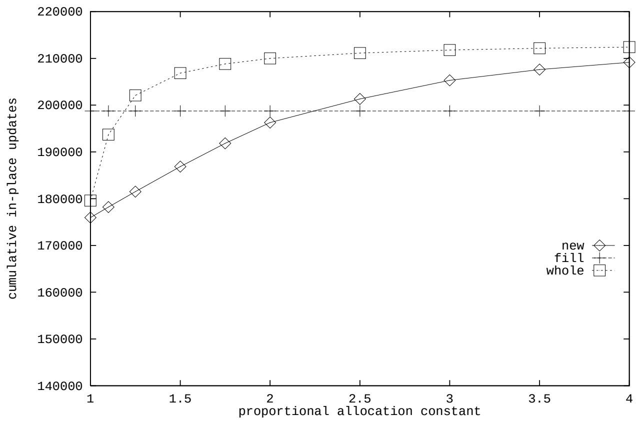

Figure 12: The impact of the constant k for the proportional allocation strategy on the cumulative number of in-place updates which occur in building the final index. The fill style (with the extent allocation strategy with extent size 3) is include for comparison. Note that the y-axis starts at 140,000.

gives an average number of read operations for a long list of 2.90, and 198,746 in-place updates at this utilization level. Both of these performance measures are worse than the best new style policy. However, the fill style as an advantage of limiting the maximum required contiguous region of disk (in this case to 3 blocks, since e has a value of 3).

We have seen that the proportional allocation strategy is a good choice for the new and whole styles. We now consider in greater depth the selection a good constant k for this allocation strategy. Figure 11 shows the impact of varying k on the utilization of long lists. The figure shows that, generally, as k rises, the utilization falls for both the new and whole styles (the fill style does not interact with the proportional allocation strategy and it is included for comparison). However, there is a cusp in the new style at a constant value of 2. This is due to the fact that multiple updates to the same word have approximately the same length. A constant value of 2 reserves space for one additional in-place update. The simultaneous increase in in-place updates is shown in Figure 12. We see that only a marginal improvement from 84% of the in-place updates at a constant value of 1.0 to 100% of the in-place updates at a constant value of 4.0 is possible. Considering both figures, we see the the majority of gains are from constant values less or equal to 3.0. In summary, based on the trade-offs presented, we recommend the proportional allocation scheme a constant of 1.1 for the whole style and 3.0 for the new style.

5.3 Exercise Disks

In this section we compare the performance of the various allocation schemes by the actual times taken to execute a trace by the execute disks process.

Figure 13 shows the cumulative time taken to incrementally build the final index. The fill style without in-place updates (fill 0) is not shown since our disks were not large enough to store the long lists for this policy due to gross underutilization of disk space (cf. Figure 9). Notice that the range of cumulative times for the final index vary by a factor of 4 as opposed to a factor of 2 determined by comparing total I/O operations (cf. Figure 8). The very significant difference

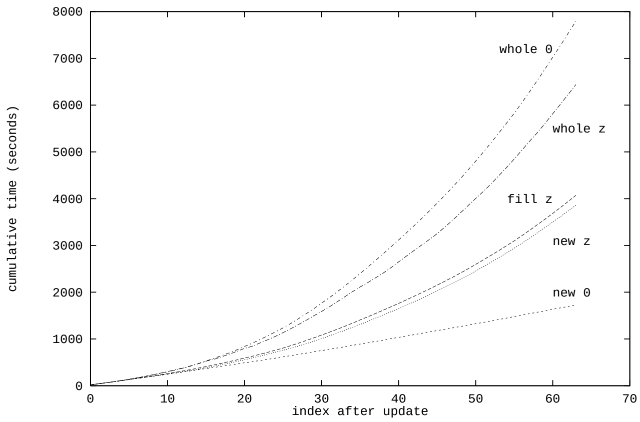

Figure 13: The *cumulative* time needed to build the final index. The x-axis is the index after the given update. The y-axis is the cumulative time needed to build the index incrementally. Each curve is labeled with the values of *Style* and *Limit*.

between the different policies implies that a policy must be chosen with care.

Comparing Figure 13 with Figure 8, we see that measuring cumulative I/O operations produces the same *qualitative* comparison of policies as measuring real execution time. That is, the ordering of policies from best to worst is the same (accounting for the addition of whole with in-place updates and the removal of fill without in-place updates). This confirms our use of I/O operations to compare policies.

We also see that the new style with limit of zero policy has an almost linear growth in the cumulative time taken as opposed to a more steep increase in the cumulative number of I/O operations. This is due to the coalescing of I/O operations by the exercise disk process. That is, since for long list updates this policy only writes sequentially to the disk, all the write operations in an update can be coalesced (up to the buffer size imposed by the exercise disk process). Figure 13 also shows that the whole style without in-place updates takes the longest cumulative time to build the final index. This is due to the additional movement of long lists compared to in-place updates.

Figure 14 shows the time taken to perform each update. That is, this figure is the non-cumulative version of the previous figure. The update times grow over time as the number of long lists in the index grows. However, the increase for new style without in-place updates is slight because updates to different long lists are coalesced into single I/O operations. A second effect shown in the figure is that the whole style with in-place updates ($Limit = z$) is the only policy whose per update time is sensitive to the variations in the size of the update. The average number of in-place updates for this policy is sensitive to the average number of postings in a long list update.

5.4 Bottom Line: What is the Best Policy?

This section has compared the performance of the various options for storing long lists. Generally, the schemes that rewrite the unused space at the end of a long list before allocating more space take considerably longer than the schemes that don't rewrite the space, due to the extra read required for each long list. However, the extra space consumed by *not* rewriting the tail of long lists makes

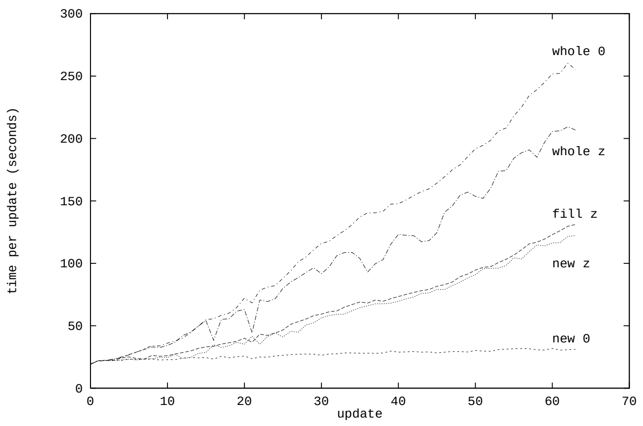

Figure 14: The time per update. The x-axis is the update number. The y-axis is the time to execute the update by the exercise disk process. Each curve is labeled with values of *Style* and *Limit*.

that option impractical for most applications.

While we have analyzed various situation, a designer of an IR system is faced with the issue of chosing a policy. Is there a best policy? In general, no policy is best. Some policies favor update time and others favor query time. We consider each policy in turn.

New style: The new style provides the best update performance, since only the last block of each long list need ever be read during the updates. The new style without in-place updates is the best if update time is critical. The policy offers a factor of 2 in update time over the next best policy and a factor of 4 over the slowest policy. However, the policy exhibits very poor query time and disk utilization. The new style with proportional allocation with constant 3 compensates for the poor query time and disk utilization. This policy is best if update time is important with reasonable query time. The policy is faster by a factor of 2 over the slowest policy and offers query performance within a factor of 2 of the best query performance. **Bottom Line:** Use the new style only if query performance is not critical. If you do use the new style, we recommend in-place updates with a proportional allocation strategy with a constant of 3.0.

Fill style: Essentially the fill style offers no advantages over the new style except that the maximum contiguous section of disk required is limited (it is unbounded in the new style). This requirement has an advantage in that long lists are automatically divided into sections of disks which can be written to disk and read in parallel (e.g., with a disk array). The cost of satisfying this requirement is small in the case of the fill style with extent allocation with constant 3 since this policy has slightly worse performance on all three metrics than the new style with proportional allocation with constant 3. **Bottom Line:** Performance is comparable to the new style; better for disk arrays. If you do use the fill style, we recommend 3 extent blocks.

Whole style: The major advantage of the whole style is its guarantee of 1 read operation for any long list. Providing this guarantee has a cost in index build time. The penalty arises from the costs of moving long lists to keep them sequential. However, this penalty is not as large as might be expected, due the relative efficiency of performing sequential disk reads and writes. The whole style without in-place updates has an about 12% slower update time compared to the whole style with a proportional allocation with constant 1.1. The latter policy has a long list utilization rate of 90%. Thus the latter policy is the best if query time is critical. **Bottom Line:** Use the whole style if query performance is critical. For this style we recommend in-place updates with a proportional allocation strategy with a constant value of 1.1.

6 Related Work

Cutting and Pedersen [1] consider incremental updates of inverted lists where a B-tree is used to organize the vocabulary. Updates are optimized by storing short inverted lists directly in the B-tree. In our framework this optimization can be represented by a very small bucket for approximately each word in the text document database. However, in [10] we show that using fewer, larger, buckets offer better performance. In addition, our scheme dynamically determines if an inverted list is stored in a fixed sized structure or a variable length one as opposed to a static division. Cutting and Pedersen also described a buddy system for the allocation of long lists. This approach deserves further experimental study since its expected space utilization is lower than the methods presented here, however it may offer better update performance than the methods presented here.

Faloutsos and Jagadish [4] extensively analyze the physical organization of long list. They study three methods that correspond to our whole style with a proportional allocation scheme, our new style with an adaptive allocation scheme (not studied here), and an unique style that combines benefits of the whole and new styles. Performance comparisons between our work and the schemes presented there are difficult since updates are not batched in that paper.

In another work, Faloutsos and Jagadish [3] extensively analyze a dual-structure scheme based on signature schemes for long lists and inverted lists for short lists. The division in the structure is static as opposed to a dynamic scheme presented here. In addition, the we believe that using inverted lists for short lists is computationally expensive since many I/O operations, each containing only a few postings, are required to update this structure.

Zobel, Moffat and Sacks-Davis [12] consider several issues in inverted file indexing. The compression methods presented there complement this paper well. They also consider fixed size buckets for storing inverted lists but do not discuss techniques for handling long lists. To compare approaches, a linear scaling of the 45 minutes update time of 132 MB of documents cited in the paper to the 686 MB text document database used here gives an update time of about 233 minutes. We halve this number to 116 minutes to account for improvements in processor performance. Our study predicts a range of index build times from about 43 minutes to 173 minutes depending on the policy used.

An interesting and entirely different approach, by Fox and Lee, based on preprocessing of document representations and a merge update of inverted lists is described in [5]. A non-incremental update time of 1 minute 14 seconds for 8.68 MB of documents appears in this article. Harman and Candela [6] also describe an update method and cite an indexing time of 313 hours for 806 MB of documents on a minicomputer with six Intel 80386 processors. Finally, our own measurements for freeWAIS version 0.202 on a DEC 5000 Model 240 (32 MB memory) with an external disk (Western Scientific) on a SCSI-I bus shows that to index 82.9 MB of our experimental text document database requires 84.1 minutes using ULTRIX V4.2A (Rev. 47) operating system.

7 Conclusion

For dynamic, time critical text document databases, it is important to modify index structures in place, as documents arrive. We have presented a dual structure index strategy to address this problem. Comparing the results presented here with the literature, we have argued that the dual-structure index has better performance than existing implementations with the added bonus of providing incremental updates. The principle source of our improvement is the dynamic division of postings into short and long inverted lists and the application of appropriate data structures to each type of list. Our evaluation is based on using actual data and hardware and simulation of an information retrieval system.

In studying our index, we found a classical trade-off between update time and query time. That is, more time spent incrementally updating the index is repaid with better query performance. We explored algorithms that optimized the time to update and algorithms with optimized query performance and determine various trade-offs between these algorithms. Performance varies by a factor of 4 in the time to build an index and a factor of 25 in query performance.

Another classical trade-off was found between space and time. As the amount of space wasted in storing long inverted lists rises, the query performance to read those inverted lists falls. We described three different methods for allocating additional space on disk to improve query performance and quantitatively describe the trade-off for these methods. In addition, we quantitatively compared overall performance.

In comparing update performance to query performance, if fast update performance is preferred, we described a policy that offers fast incremental update times with reasonable query performance (the new style with proportional allocation strategy with a constant value of 3.0). Otherwise, if fast query performance is preferred, we presented a policy that offers optimal read performance at a cost of doubling the time to build an index (the whole style with proportional allocation strategy with a constant value of 1.1). We were surprised by this result, since the optimal read performance policy requires extensive copying of long inverted lists.

We also studied an extent based allocation policy. This policy limits the size of a contiguous region of disk to a fixed maximum amount, so it is easier to implement. However, this feature does have an associated cost (about a 54% increase in query performance for the same disk utilization as the new style). This cost can be lowered by using multiple extent sizes, instead of our model of only a single extent size.

We also studied the I/O subsystem extensively and determined that the time required to write the bucket data structure to disk is dominated by the subsystem data rate, whereas the time to incrementally update the long lists is dominated by the disk seek time. We quantitatively describe the performance improvements due to speeding up disk or adding more disks. We also determine the performance of updates on an optical disk.

The work we have presented here is limited in a variety of ways, but we have addressed some of these limitations in the extended version of this paper [10] (a technical note available via FTP).

One issue is that of selecting the right amount of space for buckets, as well as partitioning this space into the right number of buckets. In [10] we illustrate the trade-offs involved, but a more detailed study is required. We also need to study how to dynamically grow the bucket space since, unfortunately, as the size of the index grows from the addition of more documents, the performance of the index degrades. This implies that we need a strategy to rebalance the division between short and long lists for any number of incremental updates i.e., periodically, as the buckets are read, they can be expanded and written in a larger region of disk.

Our results are also limited because we only considered a relatively small database of 686 MB. In [10], we generate synthetic databases with the same characteristics as our real database. The results indicate that, given the correct parameters, our algorithms scale well to larger databases. We also vary the number of disks and their speed and study the impact on performance.

Acknowledgements: Thanks to Mendel Rosenblum for discussions on file system mechanisms related to this paper.

References

[1] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of SIGIR '90*, pages 405–411, 1990.

[2] Samuel DeFazio. Full-text document retrieval benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, chapter 8. Morgan Kaufmann, second edition, 1993.

[3] Christos Faloutsos and H. V. Jagadish. Hybrid index organizations for text databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings 3rd International Conference on Extending Database Technology – EDBT '92*, Vienna, 1992. Springer–Verlag.

[4] Christos Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In *Proceedings of 18th International Conference on Very Large Databases*, pages 363–374, Vancouver, British Columbia, Canada, 1992.

[5] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[6] Donna Harman and Gerald Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, 1990.

[7] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, 1973.

[8] Katia Obraczka, Peter B. Danzig, and Shih-Hao Li. INTERNET resource discovery services. *IEEE Computer*, 26(9), September 1993.

[9] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The Rufus system: Information organization for semi-structured data. In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.

[10] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. Technical Note STAN-CS-TN-93-1, Stanford University, 1993. FTP db.stanford.edu:/pub/tomasic/stan.cs.tn.93.1.

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

[12] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proceedings of 18th International Conference on Very Large Databases*, Vancouver, 1992.