Synthetic Workload Performance Analysis of Incremental Updates *

Kurt Shoens
IBM Almaden†

Anthony Tomasic
Stanford University‡

Hector Garcia-Molina
Stanford University§

January 5, 1994

Abstract

Declining disk and CPU costs have kindled a renewed interest in efficient document indexing techniques. In this paper, the problem of incremental updates of inverted lists is addressed using a dual-structure index data structure that dynamically separates long and short inverted lists and optimizes the retrieval, update, and storage of each type of list. The behavior of this index is studied with the use of a synthetically-generated document collection and a simulation model of the algorithm. The index structure is shown to support rapid insertion of documents, fast queries, and to scale well to large document collections and many disks.

1 Introduction

As the costs of processors, main memories, and disks have fallen, full-text indexing has become an increasingly popular tool. These costs trends have also encouraged the storage of increasing numbers of documents on-line. As a result, there is renewed interest in efficient document indexing techniques.

The underlying index structure for most document retrieval systems is the *inverted list* [6]. 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, etc.) In a full text index, every word occurring in documents (minus perhaps some *stop* words) has an inverted list. The size of the inverted lists for a full text index varies from perhaps 25% to 100% the size of the text document database itself.

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 among systems. Boolean systems support

*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.

†IBM Almaden Research Center. e-mail: shoens@almaden.ibm.com

‡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

boolean expression conditions such as "(cat and dog) or mouse." Proximity systems support 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 locates documents that maximize the weighted sum of occurring words. All of these query models are well served by inverted lists.

Traditional information retrieval systems, of the type used by libraries (e.g., Stanford University's FOLIO 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 build a B-tree or hash table that maps each word to the locations of its list on disk.) Periodically, e.g., every weekend, new documents are added to the database and a new index is 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 unsatisfactory. 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 in memory small numbers of documents for each in-place index update. The in-memory batch 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.

To address these issues, in a previous work [7] we proposed and experimentally analyzed a new dynamic dual-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. Our previous work also studied 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 as an in-place update or to migrate all or parts of it, and how much free space to leave at the end of a list. Finally, we did 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 indexed according to our algorithms. Our experimental system generates the exact sequence of disk block updates that each policy produces and executes them on real disks. Based on the resulting disk layout, we also computed disk space utilization and estimated query performance.

In the previous work, we used real text data and measured performance on real disks. While this approach has the advantage of being grounded in reality, it has some disadvantages. First, a given text collection has specific characteristics such as size and type of indexing. Our collection was relatively small (686 MB). We only studied an abstracts index of our text collection where duplicate occurrences of a word in a document are dropped. Our synthetic document generator will permit us to study larger text collections of documents. As we shall see, some effects in our system can only be observed with larger text collections. In addition to abstracts indexes, we will study the effect of full text indexing of those collections. Second, in real experiments, we are dependent on the specific hardware characteristics of our test system. Performing experiments with real disks takes a long time – a typical graph can take a day of computation time to produce. And those results depend on a disk with specific characteristics such as seek time and transfer rate. Our simulations take only a few minutes to run, permitting us to explore a much larger range of parameters. In addition, our simulation is calibrated to the results of our real disks, so we are sure that the simulation is accurate. Our synthetic disk simulation permits us to vary multiple parameters of the I/O architecture to study a parameter's effects.

In summary, the new contributions of this paper are:

- a method for generating synthetic documents that are representative of a real document collection,
- a performance evaluation of our algorithms to show the effects of data structure tuning, disk performance, index scale, type of indexing, and striping, and
- a comparison between our dynamic approach of building incremental indexes and a static approach typically found in existing systems.

The next section describes the dual-structure index and algorithms introduced in our previous work. The next two sections describe our experimental design and results. We wrap up with related work and conclusions.

2 Dual-Structure Index and Algorithms

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. Our objective is to update the disk incrementally 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 [9]). This presents a dilemma for the in-place update of inverted 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 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.

2.1 Buckets and Short Lists

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 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.)

2.2 Long list data structure

Once a short inverted list L for a word w overflows from a bucket, L is written out to disk and the directory is updated with the existence of w and the location of L. On a subsequent update, an in-memory list L' for a word w is directly appended to L without accessing a bucket. The append is accomplished as follows. If L' fits in the reserved space at the end of L, then L' is added as an in-place update. That is, the last block of L is read, L' is concatenated to the tail of L and the result is written out as an in-place update. If L' does not fit in the reserved space at the end of L, then L' is appended to L in a new region of disk. That is, L is read from disk, L' appended to L, and the result written to a new region of disk with a 10% reserved space at the end of the new region. The old version of L on disk is then freed to be reused. The value of 10% was determined to give good performance by experimental analysis [7]. The long list management policy we have described here is called "whole, limit = y, proportional allocation" in [7]. This is the only policy we consider here. Others are described and compared in [7].

Figure 1 illustrates the four different situations for our dual structured index. For this example (and for this example only), we use two simplifications: we use a single bucket for all words and we assume one posting exactly fits one disk block. The bucket has a capacity of 10 postings.

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

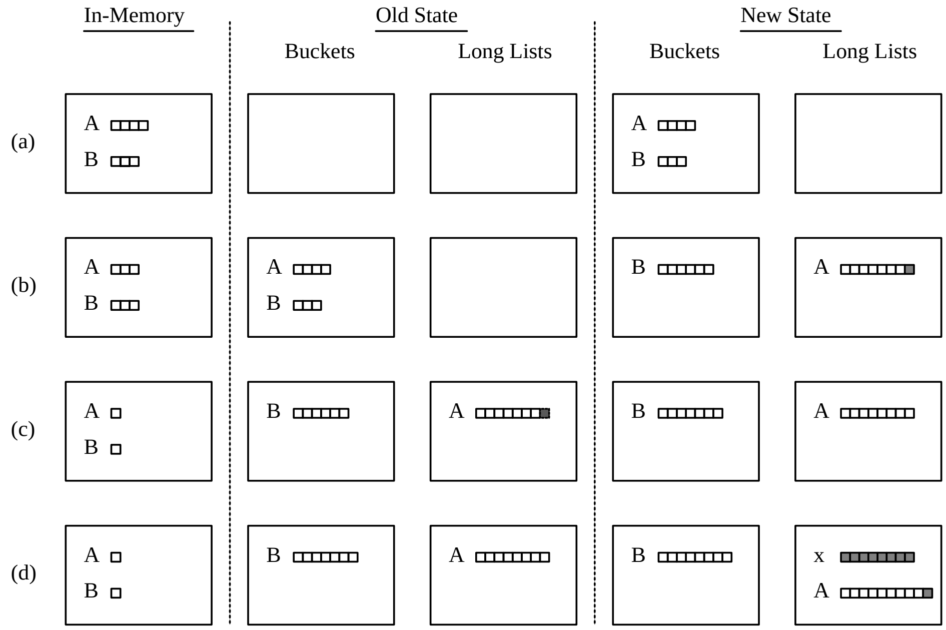

Figure 1: Each row, labeled (a) through (d) is an example of the dual structure index. Each column of large boxes represent the contents of the different data structures as indicated at the top of the column. "A" and "B" represent words and "x" represents an free inverted list which can be reused for any word. The small empty squares represent postings and the small shaded squares represent space reserved for postings. Only a single bucket with a capacity of 10 postings is used in this figure.

Each row, labeled (a) through (d) is an example of the dual structure index. The in-memory column represents the new inverted lists generated by a batch of documents. The old-state column represents the state of the disk before the new lists are added; the new-state column is the result after insertion. In Figure 1 (a), two words for a total of seven postings are inserted into the empty bucket. (Postings are represented by small empty squares.) In Figure 1 (b), six postings for two words are inserted into the bucket. The bucket now contains 11 postings, so a word must overflow since the bucket capacity is 10 postings. The longest short list is chosen, thus the word "A" with seven postings is chosen over the word "B" with six postings. The short list overflows into the long list data structure and it is written to disk with a 10% reserved space, which is one posting (the free posting is represented by a shaded square). In Figure 1 (c) two words with a single posting each are added. The posting for the word "B" is added to the short inverted list for "B" in the bucket. The posting for the word "A" is added as an in-place update to the long inverted list for the "A" word. Finally, in Figure 1 (d) two words with a single posting each are added. The posting for the word "B" is again added to the short inverted list for "B" in the bucket. The posting for "A" cannot fit in the reserved space, so the long inverted list is moved to a new location with both the extra posting appended to the end and with new reserved space. The old long list for "A" is freed for subsequent reuse by any word, as indicated by the "x" symbol.

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

Variable	Default Value	Description
Profile		See Table 2
$f(x)$		See Equation 1
Type	Abstracts	Type of indexing of documents
TotalPosting	151 Million	Total postings in all updates
Updates	200	Number of updates
Documents	2124	Documents per update
UniqueWords	350	Unique words per document
StopList	0	Number of words in stop list

Table 1: The variables used to control the synthetic generation of documents and the associated default value of each variable.

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 by packing multiple very short inverted lists into a disk block. Long lists are kept contiguous to insure good query performance.

3 Experimental Design

This section describes our experimental framework, which consists of three parts. The first part is the synthetic document generator. The output of the generator is given to a simulation of the algorithms and data structures described in Section 2. The result of this simulation is a sequence of disk operations. This sequence is synthetically simulated by our disk simulation. The result of the disk simulation is a set of timings for incremental updates.

3.1 Batch update generator

The synthetic document generator creates batches of updates that we use to drive our algorithm simulations. The document generator uses an arbitrary profile of word-occurrence frequences, represented by variable *Profile*, an estimate function f of vocabulary growth, and some document characteristic variables. If the *Type* of index is *Abstracts*, then one posting per unique word per document is generated. Otherwise, *Type* is *Full* and one posting per word occurrence per document is generated. Table 1 summarizes the parameters used.

3.2 Design of the Generator

The synthetic document generator operates as follows. Since the growth in the vocabulary of the database does not go to zero over time (because new words are constantly introduced to the database with each update e.g., family names, company names, misspellings, etc.), two types of words are generated. One are words that are already in the index and the other are synthetic words that are new to the index.

First, the simulator constructs a *synthetic final index* (SFI) representing the final index at the end of all batch updates based on three arguments:

- a *Profile* of a word-occurrence distribution, which is a set of words and the number of times each word occurs,
- the total postings *TotalPostings* in the synthetic final index, and
- a function $f(x)$ that, given a parameter x representing the number of postings, $f(x)$ estimates the number of unique words a document collection of size x.

To construct SFI, the generator proceeds by first basing the distribution on *Profile*. It then estimates the number of new synthetic unique words in the synthetic distribution by computing $f(\mathit{TotalPostings}) - \mathit{TotalPostings}$. The base distribution is expanded to accommodate the new words by assigning 1 posting to each new synthetic word in the distribution, giving SFI.

Next, random documents are created by the generator by selecting *UniqueWords* random words from the SFI. The probability that a word is chosen is proportional to its frequency distribution, e.g., if the word "cat" occurs x times in the SFI, then its probability of being chosen is $x / \mathit{TotalPosting}$. *Documents* random documents are generated and then a synthetic batch update is constructed from the documents. The generator repeatedly creates new random documents and constructs synthetic batch updates until *Updates* batch updates are generated. The form of each synthetic batch update is a set of words and the number of times that each word occurs. This corresponds to the lengths of the inverted lists in an actual implementation of an information retrieval system.

Some information systems index all occurrences of a word, rather than just the count of the occurrences in each document. Indexing all occurrences supports efficient query proximity operators such as finding terms adjacent to one another or within some specified number of words of each other. To determine how effective our algorithms are for this class of systems, we model the index for a full text indexing system. This is indicated to the generator by setting *Type*=Full. First, we assume that the position information for each posting would take the same amount of space as the count information, so the size of individual postings is unchanged. Second, we model the increase in the number of postings by indexing every occurrence of a word. Third, actual systems often limit the increase in postings for this kind of index by using a "stop list" of very frequent terms that are presumed to offer limited search discrimination. To model the effects of a stop list, we eliminate the top *StopList* most frequent words from each synthetic batch update.

3.3 Determining Parameters

For our experiments, we supplied arguments to our batch document generator to emulate the characteristics of a collection of 64 days of NetNews articles that we collected in our previous work. Some statistics for this document collection are described in Table 2. Our collection is based on all the articles at our local newsfeed. We then eliminated all nontext articles and articles of short length [7].

To provide the proper parameters to our synthetic document generator, we follow these steps.

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 2: 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.

1. Estimate the growth in the vocabulary of the text document database.
2. Estimate the number of documents in a batch update.
3. Estimate the number of unique words in a document.
4. Verify that the generator is accurate.

In Figure 2, the data points graphs the size of the vocabulary measured at the end of each batch update. The x-axis is the number of postings and ends at about 48 million postings, the total number of postings in our database. We use postings as the axis instead of the batch update number because we expect words to be introduced into the vocabulary on a per posting basis instead of on an arbitrary division of documents into batches. The y-axis in the figure is the number of words in the vocabulary. Each data point is the size of the vocabulary for our experimental NETNEWS collection measured after the given incremental update. The figure shows the continued introduction of new words into the index for each incremental update.

To predict the size of the vocabulary as the number of postings grows beyond 48 million postings, we generate a function $f(x)$ by a curve fit to the data. To determine the form of f, we start with the function $x \ln x$ (analytically derived in [1]) and then add all combinations of lower order terms. The result of fitting the data to the equation $a + bx + c \ln x + dx \ln x$ using [8] is the equation

$$f(x) = -436739.2 + 0.09494688x + 35499.8 \ln x - 0.004679559x \ln x. \quad (1)$$

(We also tried curve fits for every subexpression of the function. The function above had the lowest least-squares error of all the curve fit equations.) Figure 2 shows a close agreement between the growth in the vocabulary size and the function f. However, the curve fit equation has a limited range of utility since it eventually produces negative numbers as x approaches infinity (due to a negative value of d – the constant for the dominate term of the function). We limit the scaling

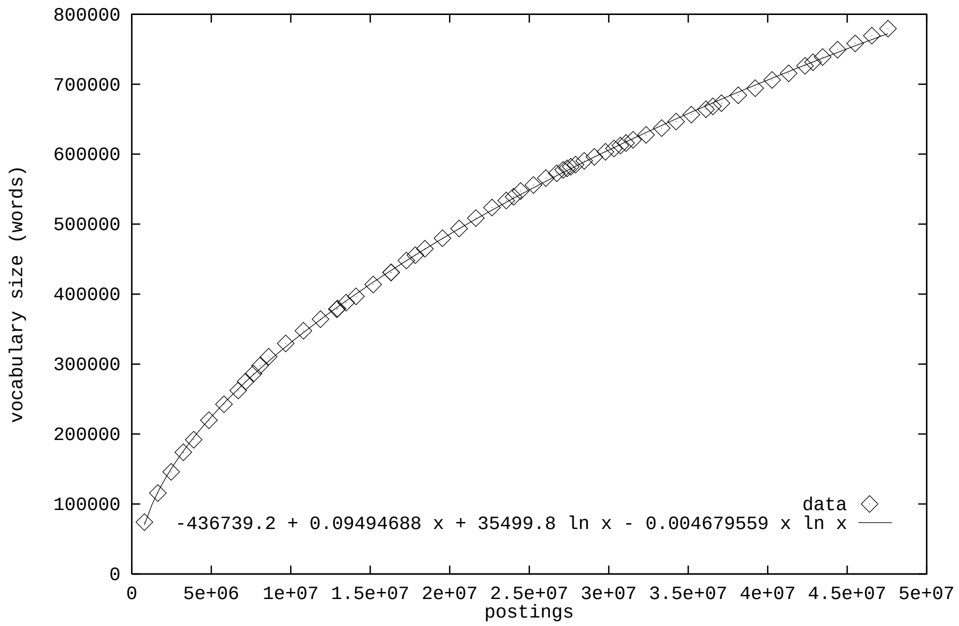

Figure 2: Comparison of the synthetic vocabulary size function f to the actual growth in vocabulary size. Each data point is generated by counting the vocabulary size of the index after each incremental update.

of our database to just over 3 times, or *Updates*= 200, the number of postings in the real text collection. Within this range the curve fit function behaves reasonably.

To estimate the behavior of documents in a batch, we assume that the number of documents per batch and the number of unique words per batch are about equal to the averages of the 64 known batches, so we let each batch have *Documents*= 2124 documents and each document have *UniqueWords*= 350 unique words. To compute *TotalPosting* for our SFI, we linearly scale the experimental final index of about 48 million postings for 64 batch updates to predict *TotalPosting*= 151 million postings for 200 updates. Using f, the SFI has an estimated total vocabulary of about 1.26 million words. The resulting synthetic text collection represents 430,000 documents occupying 2.1 GB.

To verify that the batch generator is accurate, we compare the vocabulary size of the function f with the vocabulary size produced by the generated documents. Figure 3 shows that the batch update generator initially overestimates the vocabulary size. This is due to the fact that the selection of a word for a batch update is over the entire range of words in the SFI. Eventually the generator underestimates the vocabulary size. This indicates that not every possible word was chosen for some batch update. However, we believe the generator is good enough to qualitatively compare the behavior of databases scaled to larger sizes.

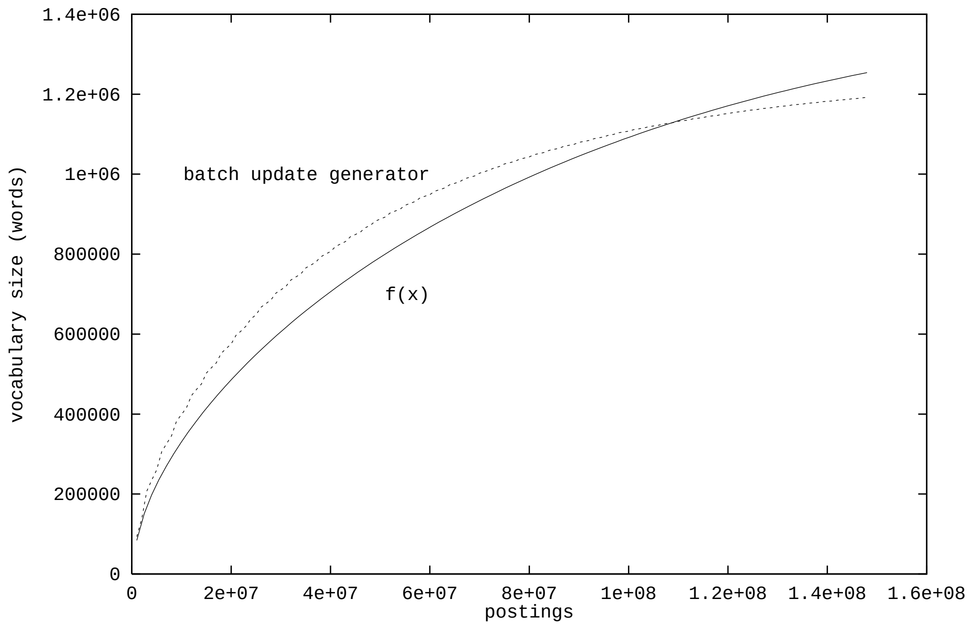

Figure 3: Comparison of the batch update generator to the curve fit function.

Variable	Default Value	Description
$Ndisk$	4	Number of disks
$Buckets$	4500	Number of buckets
$BucketSize$	6500	Postings in each bucket
$BlockSize$	4096	Bytes in long list blocks

Table 3: Variables controlling the dual structure index model

3.4 Dual-structure index simulation

To study the behavior of the dual-structure index, we wrote a simulation of the algorithms and data structures described in Section 2. The simulation reads a list of document batches and writes a trace of the disk I/O operations required to update the bucket data structure and the long lists. At the end of each batch update, we simulate the flushing of all data structures to disk. More detail on the simulation is available in [7].

In comparing our simulation to 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 our simulation does using the document representation generated by the synthetic document generator. Thus, our simulation generates exactly the same sequence of disk requests as an actual information retrieval systems, but the *contents* of the lists differ. In our simulation the contents of the disk requests are empty, in an actual information retrieval system, the contents are inverted lists.

In our model, the bucket structure is striped over all available disks. We also assume that the

Variable	Default Value	Description
Seek	20.0	Seek time (ms)
ReadBlock	3.8	Read a block (ms)
WriteBlock	4.0	Write a block (ms)

Table 4: Variables controlling the disk model

buckets all fit in memory so no disk I/O's are simulated to read the buckets.

Parameters to the simulation are described in Table 3.

3.5 Synthetic Disks

Our synthetic disk model replaces execution of traces on disks with the evaluation of an analytic function. It reads the disk I/O trace and predicts how long it would take to perform the inserts for a given number of disks with a given performance. This model can accommodate various seek times and data rates, as well as varying numbers of disks run in parallel.

Our model is based on an average time to read or write the first block of a request, plus an additional charge for each subsequent block in the request. This model takes into account the relatively large time to seek to a block and the relative efficiency of reading or writing many blocks at a time. Read and write data rates are modeled separately, because real disks exhibit significantly different rates (especially optical disks).

Table 4 lists the variables used in the synthetic functions. Let x be the number of blocks to read or write. For read operations, we use $Seek + ReadBlock \cdot x$ as the number of milliseconds required for the read operation. For write operations, we use $Seek + WriteBlock \cdot x$, as the number of milliseconds required for the write operation. These parameters were derived from measurements of three disks on a single controller for workloads like those used in this study. The synthetic disk model reads a disk trace, computes the time taken by each disk using the read and write formula, and then reports the maximum of these times as a prediction of the total time need to execute the disk trace by actual hardware.

To verify the accuracy of the model, we compared the actual times encountered in running the disk traces on real hardware to those predicted by the synthetic disk model. The results are shown in Figure 4. Two sets of curves are shown. Each set compares real to synthetic. The top set is the time taken for each batch update when no additional reserved space is allocated at the end of a long inverted list. The second set is when 10% reserved space is allocated. We see from the graph that the synthetic disk model does a good job of estimating the time taken by real disks in performing the disk traces.

4 Results

This section describes the results of our study. One of our goals is to analyze how the dual structure scheme scales; hence, we explore its behavior as the number of batches (days of operation) grows

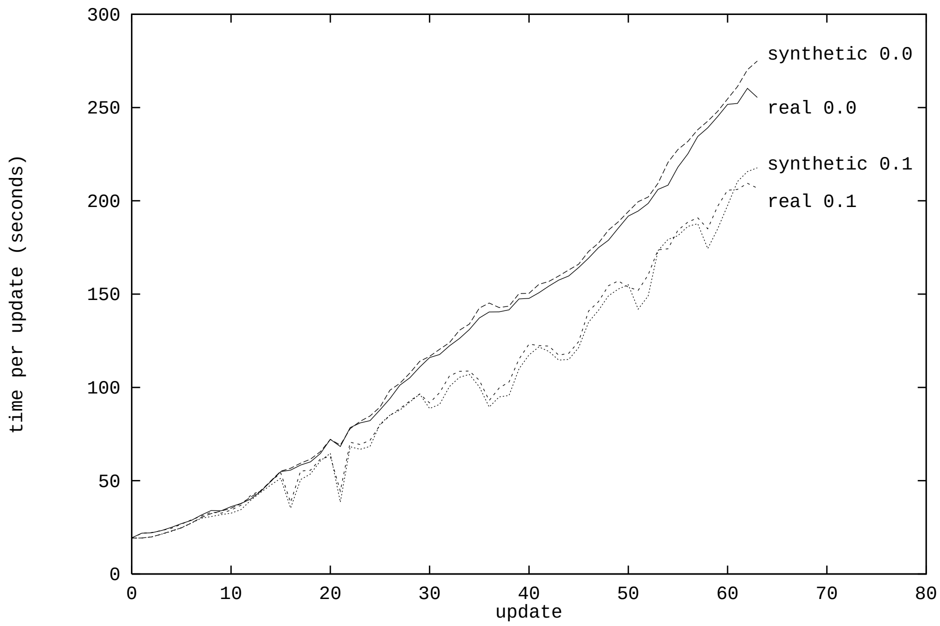

Figure 4: Comparison of the per batch update times of disk traces of real disks to disk traces of synthetic disks. Each line labeled "real" is based on real documents and hardware. Each line labeled synthetic is real documents and synthetic hardware. The additional number for each line is the fraction of reserved space added at the end of each long inverted list.

from the 64 in our real collection to 200, representing over half a year of operation. A second goal is to study the impact of various key system parameters, such as the number and speed of the disks, and the number of buckets.

4.1 Choosing the Number of Buckets

The size of the bucket data structure in our dual-structure index is important. Too little bucket space results in many long lists, slower batch insertions, and increasing internal fragmentation due to the allocation of disk blocks to relatively short lists. Too much bucket space results in internal fragmentation in the bucket structure and a higher fixed cost per batch insertion.

In this section, we apply our synthetic batch generator to examine the behavior of buckets as the index grows. We use the batch update generator and the synthetic disk process to estimate the per batch update time for 200 batch updates. Figure 5 shows this situation for several different sizes of the bucket data structure. (We vary the number of buckets and keep the size of each bucket constant.) Examining the curves for the first 20 updates, we see that as the bucket data structure grows in size, the update time per batch is higher. This is due to the need to write out a larger bucket data structure. Over the entire set of updates, however, the larger bucket data structure eventually has the lowest per batch update time since the smallest number of long lists exist for the index with the largest number of buckets. The spikes in the curves correspond to an unusual amount of moving of lists for that update. This is to be expected since each batch update is of a fixed size. This means that many of the long lists are growing at a constant rate and consequently

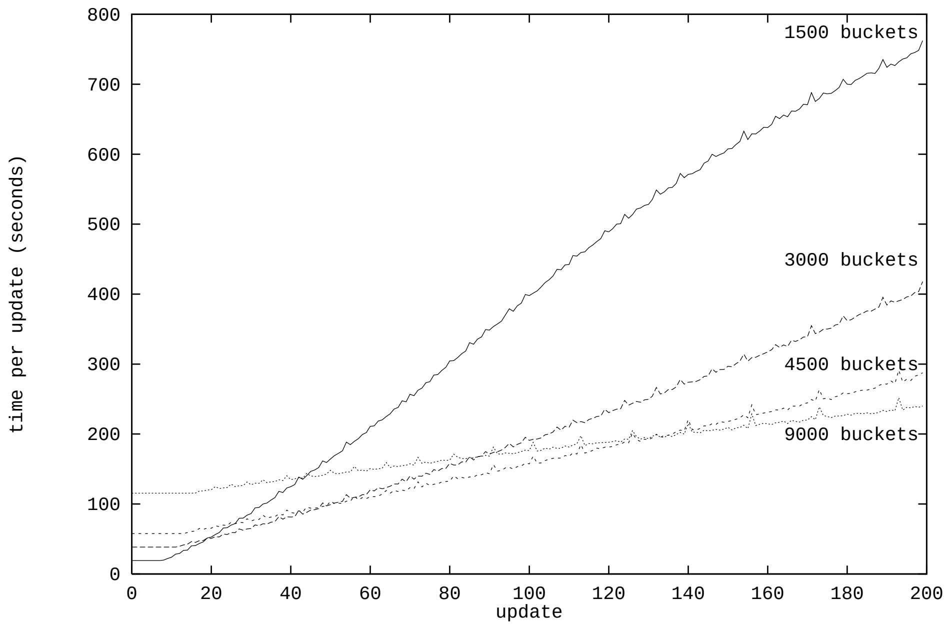

Figure 5: The update time per batch for the dual scheme for various bucket sizes.

they all overflow the 10% reserve space at the same time. The slight bend in the curve for 1500 buckets is due to the flattening out of the vocabulary size for the index as it grows.

For the same situation as the previous figure, Figure 6 shows the space occupied by the index for each of the bucket sizes. The jump in the space occupied in each curve is due to the update in which the largest lists overflow from the bucket data structure. Since each of these very long lists has a 10% reserved space, a tremendous amount of space is reserved for these lists. We see that the slope of the curve for the 1500 bucket scenario is higher than the slope for the 9000 bucket scenario. Eventually, these curves will converge to points separated only by the size of the bucket data structure. This is due to the bucket data structure filling up with words containing short lists with only a few postings.

4.2 Tuning the bucket data structure

So far we have assumed a fixed-size bucket space. Our dual structure algorithm works well in that it readily adapts to the available space. However, after many updates, shorter and shorter lists become long lists. This suggests that the bucket data structure requires *expansion*, i.e., an increase in the memory committed to the bucket data structure. For expansion, the buckets are grown in memory, and when they are written to disk at the end of a batch, they are written to an expanded disk area. This raises a host of issues—should there be more buckets or larger buckets? periodic or continuous expansion? a threshold value to trigger expansion? based on the size of the list or the fraction of the index in buckets, or some other criteria? We believe these issues are important. However, whatever expansion scheme is chosen, it is not critical for the issues we study in this paper.

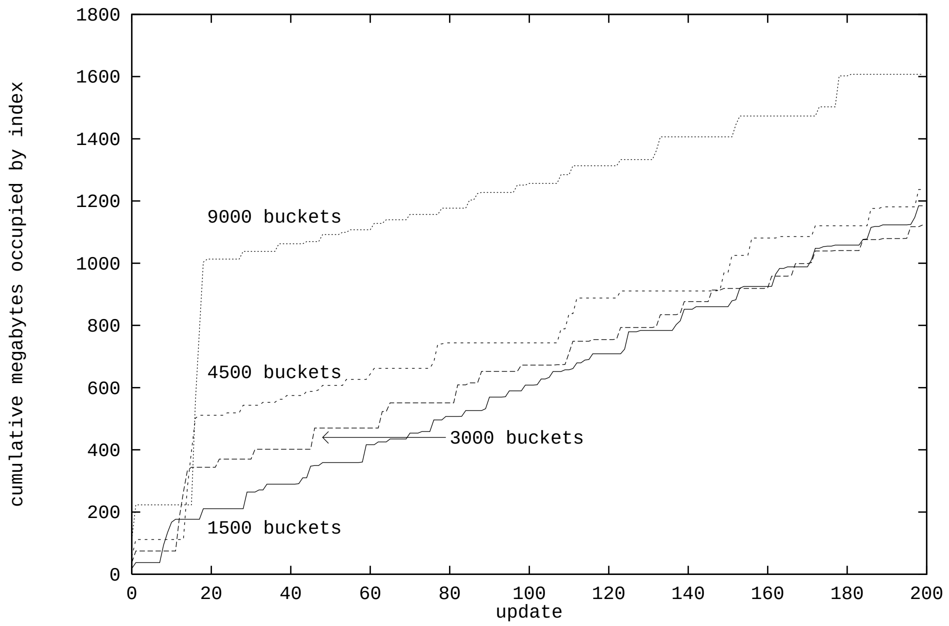

Figure 6: The update time per batch for the dual structure index with for various bucket sizes.

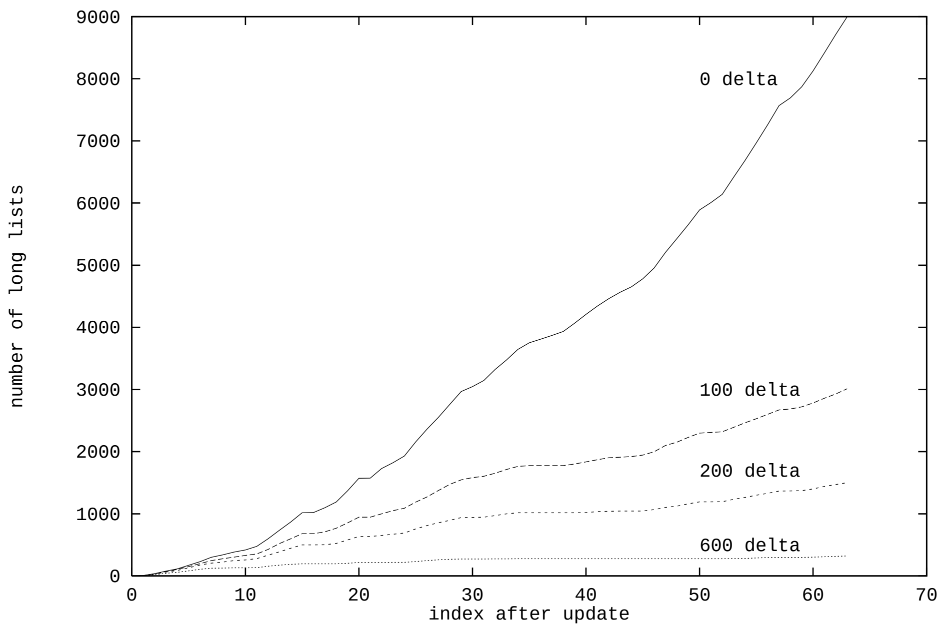

Figure 7: Controlling the creation of long lists by increasing the bucket size. The x-axis is the update number. The y-axis is the number of long lists in the index at the end of the update. As updates are processed, the bucket size is determined by the function $BucketSize + dx$ where d is the label associated with each curve and x is the update number.

For completeness, we briefly consider one possible expansion here. We limit the number of long lists by incrementally increasing the amount of memory dedicated to buckets as the index grows. Figure 7 shows the number of words with long lists at the end of each update. Let x be an update number. Each curve corresponds to the function $BucketSize + d \cdot x$ (where d is the *delta* associated with a curve). For instance, the second to top curve is labeled 100, so the size of the buckets at each update x is $6500 + 100x$. The graph demonstrates that the number of words with long lists in the index can be held essentially constant as the index grows by allocating a constant increase in the size of the bucket data structure.

The memory cost of increasing the memory dedicated to the bucket data structure is high. With a delta of 0, the total size of the bucket data structure is $Buckets(BucketSize + d \cdot x) = BucketTotal$ or $1500 \cdot (6500 + 0 \cdot 63) = 9,750,000$ words and postings. At the end of the last update, the bucket data structure holds 19.8% of all the words and postings. Assuming that a word or a posting needs 6 bytes to store on average, the bucket data structure is 10.9% of the size of the raw text of the database. With a delta of 100, the total size of the bucket data structure is $1500 \cdot (6500 + 100 \cdot 63) = 19,200,000$ or almost double the size of the bucket data structure with a delta of 0. We use a delta of 0 for the remainder of the paper.

4.3 An Alternative Scheme for Comparison

For the remaining experiments we will describe in this section, it is useful to contrast our dual-structure index to a simple "old master/new master" index structure, which we refer to as the *alternative* scheme. This alternative scheme is what current information retrieval systems typically use. In this scheme, the postings for each word are stored contiguously and there is no free space allocated between the lists for different words. On each batch update, the entire existing database is read sequentially, the updates in memory are applied, and the database is written out. Note that the database need not be read completely into memory, but can be processed in convenient-sized chunks. In our comparison, we assume that the time to perform an update in the alternative scheme is mostly disk I/O time and that the I/O time to write the database can be completely overlapped with the time to read it.

The alternative scheme as described above requires about twice as much disk space to store the index, since a copy of the old and new index must exist simultaneously. It is possible to design schemes that break the index into several large pieces to reduce the storage overhead. We assume in our simulation of the alternative scheme that such designs will have little effect on the batch build time if the pieces are kept reasonably large (e.g., 8 to 32 megabytes).

Since the alternative scheme's update time is controlled by disk data rate, we assume that the index is laid out so that the index can be read in parallel from half the disks and written in parallel to the other half.

The alternative scheme offers a couple of advantages. First, since there are no gaps between the lists for different words, there is no space overhead due to internal fragmentation. The postings for words are stored contiguously, so they can be read quickly for queries. Finally, all of the disk I/O during an update is sequential, which typically yields a factor of ten faster data rate than random I/O.

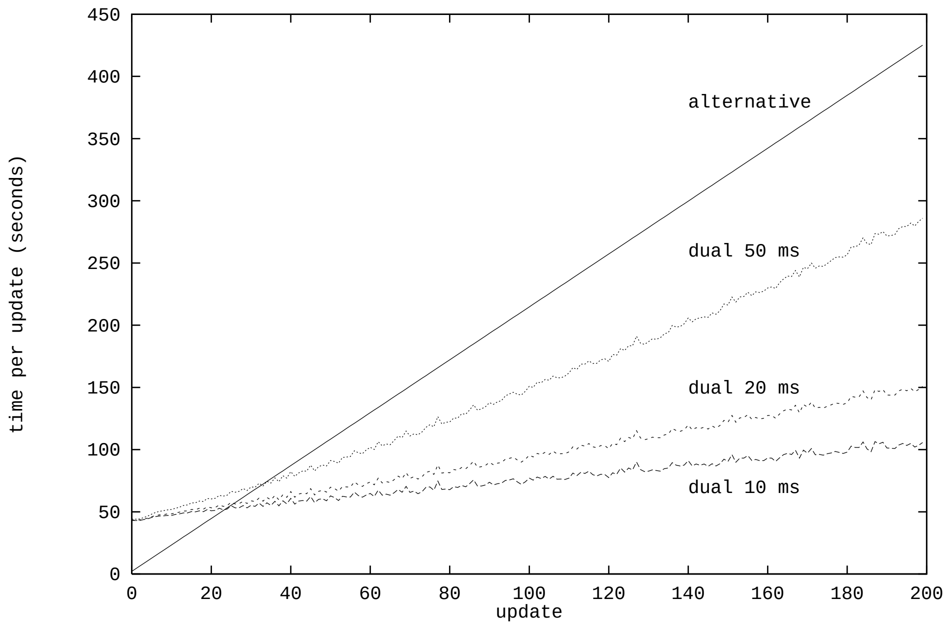

Figure 8: Comparison of varying seek times

On the negative side, the alternative scheme has to re-read the entire database, even when applying a small batch update. We expect the index build time of the alternative scheme to suffer as the database grows. These features will become apparent when we present our performance results in the following sections.

4.4 Disk Performance

In this section, we show the effects of disk performance on index building time. In our measurements on a real system, we used disks that took about 20 milliseconds to read or write the first block and could sustain a data rate of about 1 megabyte/second. We considered these disks to be "medium speed." We also modeled optical disk-like parameters (slow seek, somewhat slower read data rate, significantly slower write data rate) and current best-performance magnetic disks (faster seek and data rate).

To compare seek times, we modeled four disks with 10, 20, and 50 millisecond times to read or write the first block plus the data rates that we measured on our real disks. The results of this comparison are shown in Figure 8. There is a strong component of seek time. Initially, most of the I/O operations are for rewriting the buckets and are strongly sequential. As the trace progresses, the average size of the data of an I/O operation falls to only a few blocks as more long words are created. Since the average number of blocks written for each long word is small, the average size of the data of an I/O operation falls and the seek time begins to dominate the total I/O time required to add a batch to the data base. Since the number of seeks is proportional to the number of long words, the faster seek time disks have a smaller slope of increased update time as batches are added.

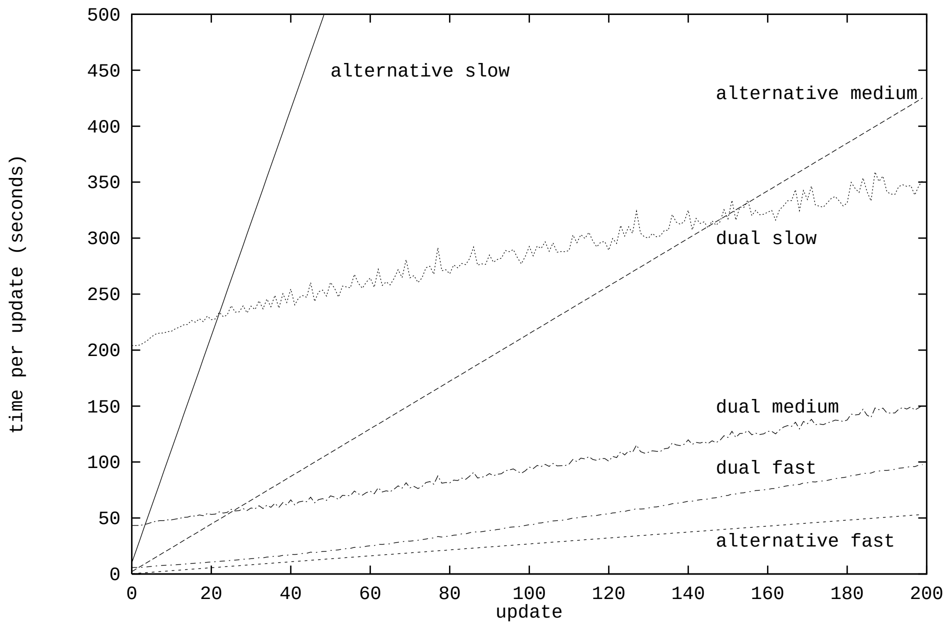

Figure 9: Comparison of varying data rates

The alternative scheme is not affected by seek time; a single line is shown for four 1 MB/second disks. The alternate scheme is clearly more time consuming than the dual structure one, even with slow disks, except when the index is very small. Because of its high cost, the alternate scheme would probably not be used after each relatively small batch as we have done here. Instead, larger batches would be constructed. The update cost would still be very high, but it would not be paid as frequently.

To compare the effects of data rate, we model disks with 20 millisecond seeks as seen on our real disks and used a variety of data rates, as show in Figure 9. The slow transfer curve has a data rate based on optical disks (660 KB/sec reads, 210 KB/sec writes). The medium transfer curve has a data rate based on our disks (1 MB/sec to read and write). The fast transfer curve has a data rate based on current high end disks (8 MB/sec to read and write).

For the dual-structure index, two effects are visible from the figure. First, the slower data rate device exhibits far larger variations in update times. The spikes in the update times are due to update batches that move many long lists. Second, faster transfer times significantly reduce the update times. This variation is mostly due to the reduced time to rewrite the buckets on each disk. That difference is seen on the time required to process the first batch.

The time to perform a batch update in the alternative scheme is the database size divided by the data rate. Therefore, higher data rates result in a flatter line.

Note that the best build performance in this set is given by the alternative scheme. This is due to the combination of a high data rate and only "nominal" seek performance. The poor seek performance hurts the dual structure index but does not affect the alternative scheme. Should technology trends provide disks with significantly faster data rates but only somewhat faster seek times, the dual structure index should be tuned with a larger bucket data structure. As seen

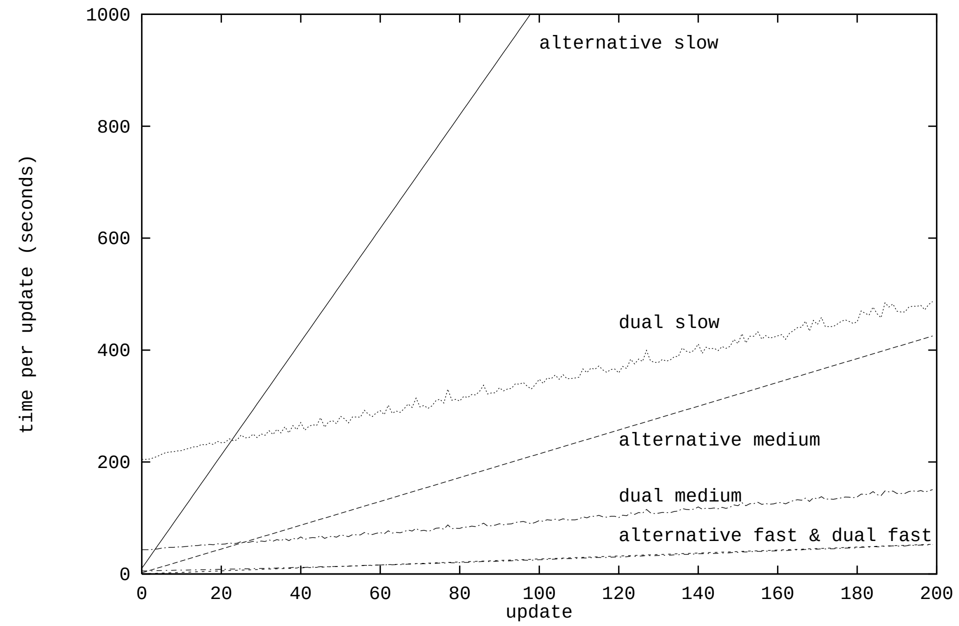

Figure 10: Comparison of varying disks

earlier, a larger bucket structure increases the amount of sequential I/O per update but decreases the amount of seeking.

To provide a complete picture of the differences in index time caused by disks of various performance, we compare various specific devices directly. The slow disk curve represents optical drive parameters (50 millisecond seek, 660 KB/sec read, 210 KB/sec write). The medium disk curve represents our disk parameters (20 millisecond seek, 1 MB/sec read and write). The fast disk curve represents the latest generation disk parameters (10 millisecond seek, 8 MB/sec read and write). The results are shown in Figure 10. For the dual structure index, the effects of the previous two studies are combined. The fastest seek time provides a more gradual slope coupled with the fastest data rate giving the lowest base update time. For the alternative index, seek time is irrelevant and the results from the data rate study are repeated.

Again, note that for the fastest disk drives, the alternative scheme and the dual structure index produce nearly identical performance. The fast disk has 8 times the data rate but only 2 times the seek performance as our nominal drives.

Another important variable in the configuration of a system is the total number of disks. As the number of disks grows, more I/O's can occur concurrently. In particular, the buckets will reside on more disks and can be thus updated in less time. Each long list is still written to a single disk, but more lists can be updated concurrently.

To study the effect of this variable we fix the single disk parameters to those of our current disk drives. The maximum system disk throughput we assumed for this study is 12 MB/second and is well within the capabilities of current workstations. Given a properly spread I/O load, the use of multiple disks simulates a smaller number of disks that support a high data rate and fast seek time. This effect is borne out in this study, shown in Figure 11. As more disks are used, the slope

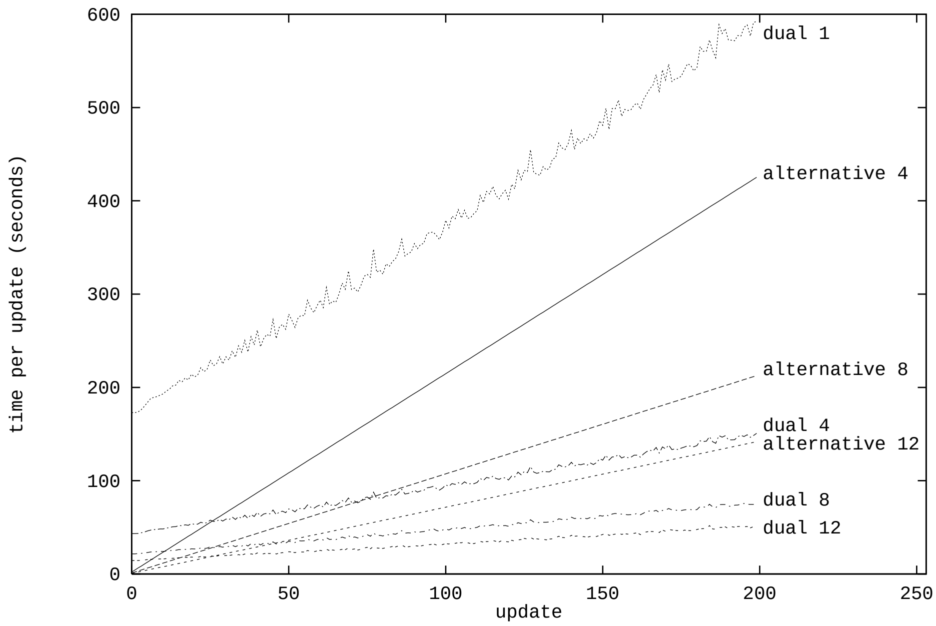

Figure 11: Comparison of varying number of disks.

of the lines for the dual structure index flatten out and the variations in batch time are reduced, as seen in the seek study. In addition, more disks reduce the base time to rewrite the buckets, as seen in the data rate study.

For the alternative scheme, more disks means more aggregate throughput so the lines flatten out accordingly. Note that increasing the number of disks helps the dual structure index more than it helps the alternative scheme. This is because the dual-structure scheme can take advantage of the larger number of seeks per second that can be performed with more drives.

To summarize, we have shown that the time to update the dual structure index is roughly a linear combination of the size of the bucket data structure divided by the aggregate disk data rate and the number of long words time the seek time. Since increasing the size of the bucket structure reduces the number of long words, the dual structure index can be tuned to accommodate the data rate and seek time for the disks used to store the index.

For the alternate scheme, the time is proportional to the database size divided by the aggregate disk data rate. Both of the schemes split well over multiple disks.

4.5 Full text vs. abstract

The following table summarizes the posting volumes seen when indexing one occurrence per document, all occurrences, and all occurrences with a stop list for our synthetic trace of 200 batches:

Type of index	Posting count	Index space
Type = *Abstracts*	148 million	1.1 GB
Type = *Full*	216 million	1.7 GB
Type = *Full*, *StopList*= 20	163 million	1.4 GB

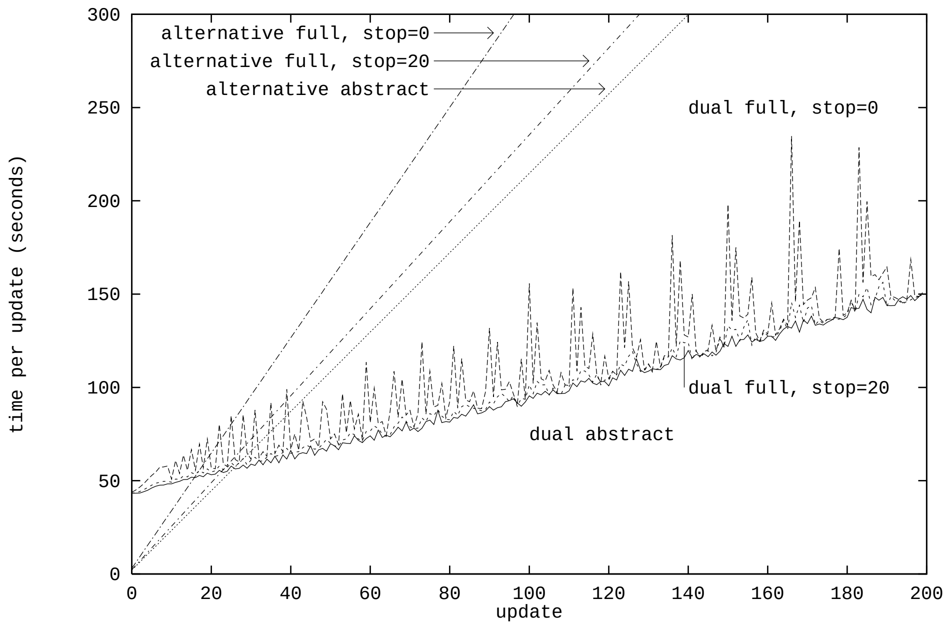

Figure 12: Times per update on synthetic documents comparing indexing one occurrence per term per document, all occurrences of a term, and all occurrences of a term minus a 20 word stop list.

As can be seen from the table, indexing all occurrences adds 46% more postings. The use of a stop list reduces the postings about 25%. (Using a stop list on the one occurrence per document index only reduces postings about 5%.) In addition, indexing all occurrences increases the variation in long list sizes.

Figure 12 shows the time taken to update the dual structure index with the three alternatives. We modeled the disk behavior for four disks with the performance of our real disks. As usual, the large variations in the time per update are caused when many long lists overflow the 10% reserve space and are moved on the same update.

Note that the use of the 20-word stop list removes the wide variations in batch build time. There is still a penalty of 27% additional disk space consumed, but a total build time only 3% larger. The reason that the build time does not increase that much is that the time is dominated by the number of seeks. Since the number of long words updated on each batch is about the same, there is little variation.

4.6 Striping long lists

In this section, we consider the effects of striping long lists across the available disks. Such striping of long lists allows the occurrences of a word to be read in parallel from several drives to reduce query times. However, during index building, striping words across drives will cause a larger number of disk seeks to be performed per batch. As seen earlier, the number of seeks drives the slope of the update line as the database grows. To counteract this effect, we also consider limiting striping to relatively long lists.

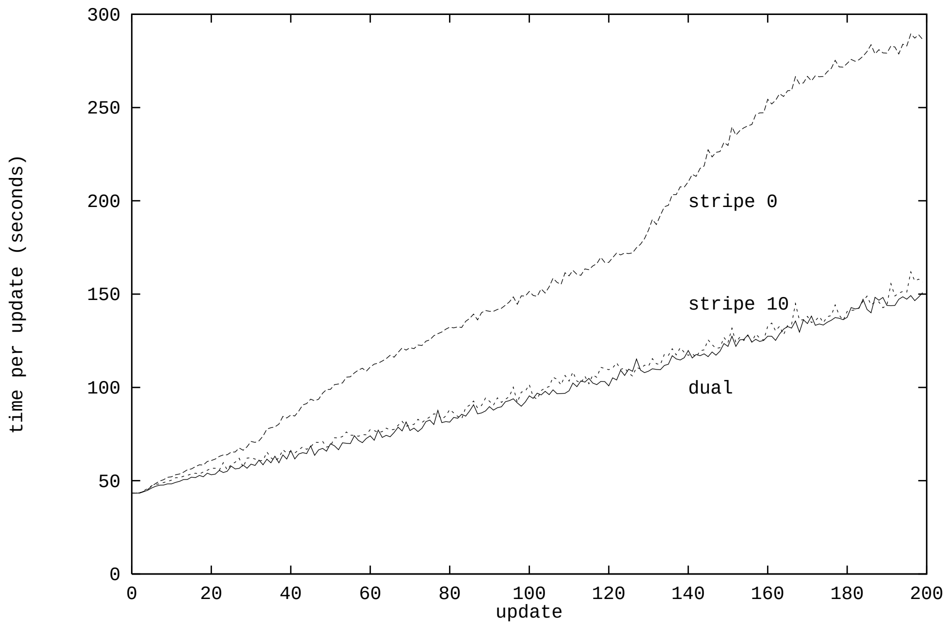

Figure 13: Times per update on synthetic documents comparing the dual structure index with two striped variations

In our striping design, we assume that the allocation of a long list will be split among as many drives as possible. We further assume that the space allocated for each word will be filled in one disk at a time. In that way, the free space at the end of a long list will be concentrated on the minimum number of disks, which will help reduce the number of seeks required during the batch updates.

To limit striping to frequent words, we introduce a new parameters, *MINALLOC*, which gives the minimum number of blocks we will allocate to a stripe. For example, suppose we have four disks and *MINALLOC* is 10. If a total of 20 blocks are required for a long list, two stripes of 10 blocks each will be allocated, rather than four stripes of 5 blocks each. If a total of 15 blocks were required, we would allocate a stripe of 10 blocks and a stripe of 5 blocks to avoid wasting space. As *MINALLOC* is increased, the number of long words subject to striping decreases.

Figure 13 shows the build times for our standard dual-structure index, for the striped index with *MINALLOC* set to zero, and for striping with *MINALLOC* set to 10. As predicted, striping significantly increases the build time slope with *MINALLOC* set to zero. With *MINALLOC* increased to 10, however, the build time penalty for striping vanishes.

Due to space limitations, in this paper we have not studied query processing times. However, since the main advantage of striping is the improved read times of long lists, let us briefly consider this metric.

Without list striping, the postings for a word can be read with a single sequential I/O, in both the dual structure and the alternate schemes. The transfer time is proportional to the number of postings divided by the single disk rate, except when the list is in a bucket (in the dual structure scheme). In this last case, a read of the entire bucket is required.

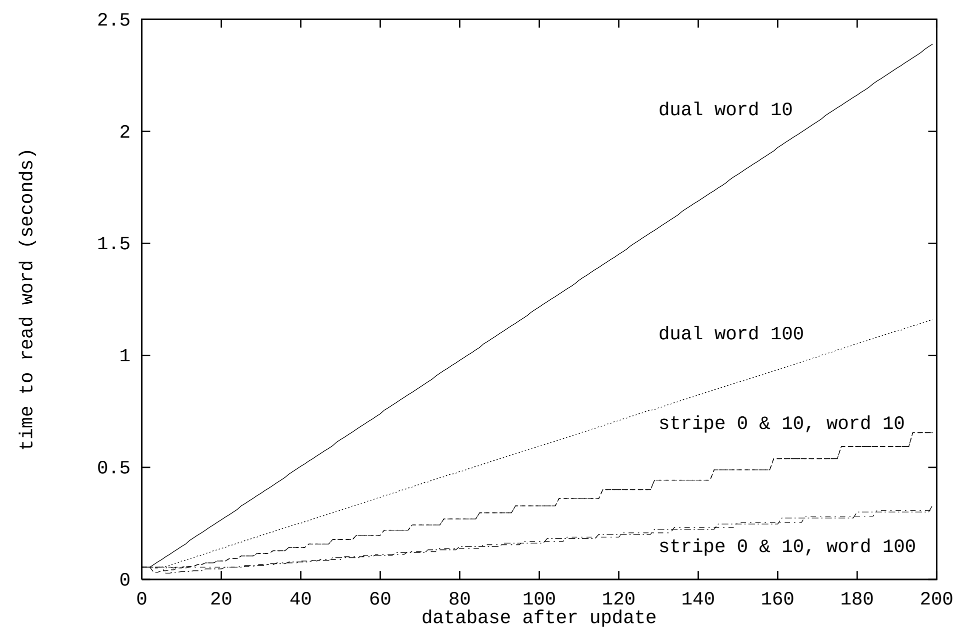

Figure 14: Times to read the 10th and 100th most popular words for the dual structure index and two striped variations

Striping can reduce the transfer time of long lists. To illustrate the potential gains, let us consider two particular words, the 10th and 100th most popular ones. Figure 14 compares the times to read the lists for these words with the variations in index design considered earlier. The striped schemes take advantage of multiple disks and read lists considerably faster than the non-striped schemes. In addition, notice that the read performance with *MINALLOC* set to 10 is the same as with *MINALLOC* set to 0.

A complete evaluation of query performance is beyond the scope of this paper, as it requires knowledge of the distribution of words appearing in queries, the distribution for the number of words, and the query arrival patterns. The last factor is important because striping also increases the number of seeks that must be performed. Thus, under certain high load scenarios, striping can be counterproductive. However, the results we have given here show that, if striping is beneficial for query processing, then an appropriate choice for *MINALLOC* makes the dual structure index updateable in the same time as without striping.

5 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 Section 4 we show that using few, larger, buckets offers better performance. In addition, our scheme dynamically determines a threshold value for determining if an inverted list is stored in a fixed sized structure or a variable length one.

Cutting and Pedersen also described a buddy system for the allocation of long lists. This approach deserves further experimental study since it offers comparable space utilization and it is not clear if it offers better update performance than the methods presented here.

Faloutsos and Jagadish [3] extensively analyze the physical organization of long list. 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 [2] 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, 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 [10] 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.

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 [4]. The scheme is non-incremental. Harman and Candela [5] 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 (Seagate) 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.

6 Conclusion

For dynamic, time critical text document databases, it is important to modify index structures in place, as batches of 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.

We have described a means of creating large synthetic document collections and have described a simple model of parallel disks. The synthetic document model can be tuned to model the statistics of an existing text collection and can be scaled to model varying numbers and sizes of documents. We adjusted our disk model to conform to our existing real disks; the disk model can also be modified to model slower disks, such as optical, or the faster disks that have been recently announced.

We 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. In particular, we showed that the inverted file organization described in this paper stripes across multiple disks well.

We showed the effects of varying the amount of disk space allocated to the bucket data structure and suggested a way to scale the number of buckets as the size of the inverted file grows.

Finally, we showed the effects of applying our inverted file organization to the indexing of each word occurrence as opposed to indexing only counts of occurrences per document. With no stop list, we observed a 13% time penalty for full-text indexing; with a 20 word stop list, the penalty falls to 3%.

Acknowledgments: 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] 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.

[3] 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.

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

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

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

[7] 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.ps.

[8] Stephen Wolfram. *Mathematica.* Addison-Wesley, Redwood City, California, 2nd edition, 1991.

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

[10] 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.