Fast Inverted Indexes with On-Line Update

Charles L. A. Clarke* Gordon V. Cormack Forbes J. Burkowski

Dept. of Computer Science
University of Waterloo, Waterloo, Canada, N2L 3G1

Technical Report CS-94-40
November 23, 1994

Abstract

We describe data structures and an update strategy for the practical implementation of inverted indexes. The context of our discussion is the construction of a dedicated index engine for a distributed full-text information retrieval system, but the results have wider application. Retrieval operations require a single disk access per query term. The on-line update strategy guarantees the consistency of on-disk data structures. Index compression integrates smoothly.

1 Introduction

1.1 Environment

Our general concern is the construction of a distributed full-text information retrieval system. The basic architecture consists of a group of LAN-connected processors, each managing its own separate disk and memory. Individual processors act as either *text servers*, storing documents and servicing requests for portions of these documents, or as *index engines*, identifying the portions of documents that match client-generated search criteria. To external clients, the group of machines appears to be a single large information retrieval system. A front-end processor, the *Marshaller/Dispatcher*, coordinates the activities of the group of processors, interacting with client applications, dispatching queries to the index engines and text servers, marshalling query results and returning the results to clients. Figure 1 provides a schematic overview of the architecture.

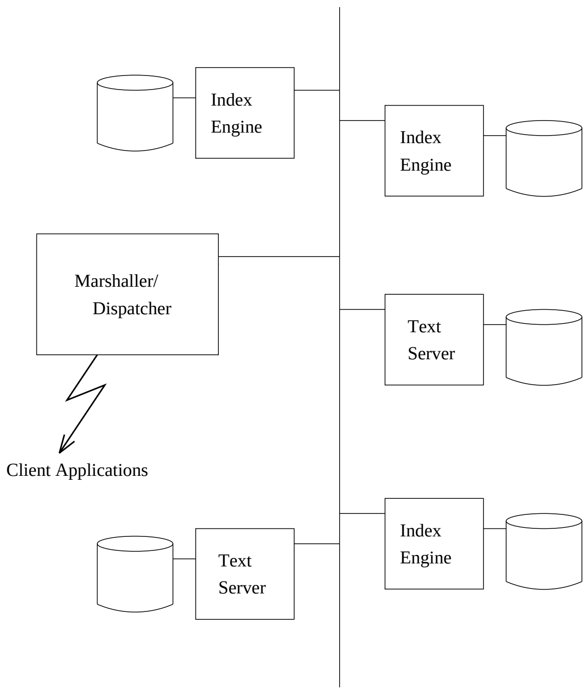

Figure 1: Architecture of the retrieval system.

*Email: claclark@plg.uwaterloo.ca

Design aspects of distributed full-text information retrieval systems have been the subject of earlier research. Burkowski [1] studied the division of processors into text servers and index engines, with the conclusion that such a split can implement a system which provides better overall response time than a system in which each processor acts as both text server and index engine. Tomasic and Garcia-Molina [12] examine the allocation of index terms from a set of documents to index engines. They conclude that all indexing of terms for an individual document is best allocated to a single processor.

1.2 Inverted Indexes

Our specific concern is the data structure design and update strategy used by the index engines. The basic data abstraction implemented by an index engine is an inverted index [7]. File structures based on inverted indexes are standard for implementing information retrieval systems [5, 6, 11, 10]. An inverted index is a function that maps index terms into positions in documents where the terms occur. Index terms are typically words, but may include document markup tags and other structural information of importance to database clients.

Figure 2 presents a simple (but widely used) realization of the inverted index data abstraction. The *dictionary* maps terms into a pair of offsets into the *postings file*. Between these start and end offsets in the postings file is a sorted list of *postings*, positions within the database where the term occurs. Both the postings file and the dictionary are large enough to require disk storage. Using this realization, a mapping of a given index term into its postings list consists of a binary search of the dictionary (requiring $O(\log w)$ disk accesses, where w is the number of index terms in the dictionary) followed by a single access into the postings file.

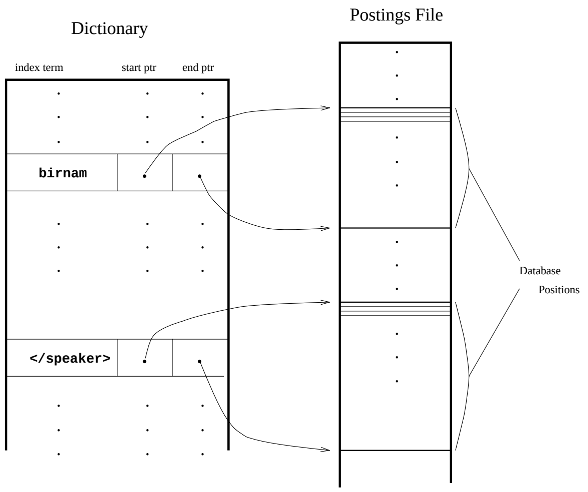

Figure 2: Simple realization of an inverted index.

1.3 Practical Issues

In an operational environment there are a number of practical issues to be considered when implementing inverted lists.

Retrieval Response Retrieval operations far outnumber update operations. Querying the retrieval system is the primary operation used by external clients and response time is of utmost importance. The mapping of an index term into its postings list must require as few disk accesses as possible. Ideally, a single disk access would be sufficient to translate any term, independent of the size of the dictionary and postings file, and independent of the size of the postings list for the particular word.

Update Throughput Updates are usually additions of new documents. Occasionally, deletion of documents, and addition and modification of indexing may be required. Since update is primarily a maintenance function rather than an external client service, update throughput, not response time, is of importance. The simple file structures of figure 2 require a complete rebuild to apply updates.

Index Compression Compression will increase the amount of dictionary and postings data that can be stored on available disk [14]. Since compression and decompression techniques operate by linearly processing a range of data, this property creates a potential decrease in retrieval response time if random access into the data is limited.

Consistency The database must be maintained in a consistent state at all times. For example, if a failure occurs during update of the postings file, the dictionary must not be left pointing to an incorrect range of postings.

1.4 Existing approaches

Most discussions of inverted indexes for information retrieval assume that the file structures are static — created by an initial database load operation and not modified thereafter. These file structures generally require multiple disk accesses for term translation [6, 10].

Discussions of updatable inverted indexes generally adopt an append-only model of update. Tries, hashing and the ubiquitous B-Tree can all be used to implement an updatable dictionary. An updatable postings file can be implemented using a variety of free space management techniques. The postings for a particular term may be maintained in chained buckets. A new bucket is added to the chain each time a document append causes a bucket to overflow. Alternately, postings may be stored in contiguous extents with free space left after each extent. If the extent overflows, the postings list is copied to a larger extent.

Cutting and Pedersen [4] examine in detail the use of B-Trees to implement an efficiently updatable inverted index. Using their optimizations to the basic B-Tree file structure, a retrieval operation can require as little as one disk access, but may require more, depending on the caching strategy used and the branching factor of the B-Tree. Updates are buffered in main memory and applied in large batches. Postings may be compressed, but the ability to do delete operations is then sacrificed. Maintenance of database consistency during block pointer updates is not discussed.

Burkowski [2] discusses free space management for the postings file and proposes an organization that groups postings into subsets and leaves free space for appending new postings at the end of each subset. Postings for several different index terms may be combined in a single subset. Postings for different index terms are identified by unique markers assigned at build time. An append that overflows available free space triggers a complete rebuild of the postings file. Retrieval operations cannot be processed while a rebuild is taking place. During a rebuild, free space usage since the last rebuild is used as a predictive model for free space allocation in the rebuilt postings file. Translating an index term requires at least two disk accesses: one (or more) into the dictionary, and one into the postings file to retrieve the appropriate subset. Update operations other than append can only be performed during a rebuild.

1.5 Our approach

In the remainder of the paper we present data structures that efficiently realize the inverted index data abstraction and permit the continuous on-line application of updates without significantly disrupting retrieval performance. Accessing the inverted index for a term requires a single disk access. Update is an on-going background process, re-writing the database on an on-going basis. Updates are maintained in main-memory data structures until they can be applied to disk. The update process is occasionally checkpointed. If a processor failure occurs, the update process may be restarted at the last checkpoint. The integration of caching and index compression is straightforward.

The work described in this paper is part of the Waterloo Multi-User Multi-Server Very Large Text Database Project (the MultiText Project). The primary goal of this project is the creation of a prototype distributed full-text information retrieval system. Where exposition is simplified and no generality is lost, we use the concrete data structures of the MultiText Project in our discourse.

2 Interfaces

The index engine ignores document boundaries, treating the text in the database as one continuous sequence. Document boundaries are treated the same as any other structural element. A position in the database is a single positive integer. The granularity of this position — whether it refers to a character, word or to an entire document — depends on the needs of the query language used by the retrieval system. The text structure model used by the MultiText project allows multiple terms to be indexed at a particular location. This property is crucial to schema-independent retrieval [3].

An index engine is responsible for implementing two classes of operations: retrieval operations and update operations.

A retrieval operation is a request to solve a query, expressed in some query language. Inverted indexes easily implement the boolean-algebra-based languages used by most commercial text retrieval systems [11, 13]. Inverted indexes are also appropriate for implementing the schema-independent heterogeneous structured text query language used by the

MultiText project [3].

Each query solution is a pair `(start, end)` indicating a range in the database that satisfies the query. Each retrieval operation may result in one or more solutions of this form.

An update operation is a request to add or remove indexing. An *add indexing* operation has the form:

```
Add (position, term)
```

Which indicates that the specified term occurs at the specified position in the database. A *remove indexing* operation can take one of three forms:

1. `Remove (start, end, term)`
   Removes indexing of the specified occurrence of the term in the range specified by the start and end positions.

2. `Remove (*, *, term)`
   Removes all indexing of the specified term.

3. `Remove (start, end, *)`
   Removes all indexing in the specified range. This operation may be used to remove an entire document from the index engine.

The add operation and the first two remove operations change the postings list for a single index term and are referred to as *local update operations.* The third remove operation may affect many postings lists and is referred to as a *global update operation.* It is worth noting here that all the update operations are idempotent, they may be applied one or more times with the same effect. As an aside, the retrieval operation implemented by the text server is:

```
lookup (start, end)
```

which returns the text associated with the range.

3 Resources

A primary concern is the management of the index engine's memory resources: disk storage and main memory RAM.

M_{D} = Quantity of disk storage (words)
M_{R} = Quantity of main memory (words)

We have $M_{\mathrm{R}} \ll M_{\mathrm{D}}$ where the size difference is typically a factor in the range 16 to 256.

The quantity M_{R} does not include the memory required for the operating system and its internal data structures, or for the index engine application software. To permit control over these resources, we disable all operating system paging, swapping and other memory management. The operating system is used only to provide address translation, physical I/O to the disk, network access and multiprocessing. Disk storage may consist of several physical disks. We assume striping if this is the case.

A word must be big enough to hold:

- any database position.
- any index term
- any integer in the range $[0, M_{\mathrm{D}}]$

A 32-bit word size is sufficient to index approximately 20GB of (uncompressed) English-language text. A 64-bit word is sufficient for all currently conceivable applications. A index term is represented by a word-sized *term symbol.* This mapping of index terms to term symbols is a global function implemented by the Marshaller/Dispatcher. Ordering of the term symbols corresponds to the lexical ordering of the index terms.

4 Data Structures

Before examining the on-line update algorithm, we examine the static organization of the index engine data structures as they appear between cycles of the update process.

The vast majority of disk storage is allocated for *index blocks.* Together, the index blocks make up the *index*, which combines the functions of the dictionary and postings file of figure 2. The index contains both index terms and postings. An index block is of fixed size B. The size of the index is M_{DIdx}. The total disk storage allocated for the index is M_{DIdxS}. Both M_{DIdx} and M_{DIdxS} are multiples of B.

When in use, each index block contains one or more *index entries.* Each index entry consists of a term symbol followed by an occurrence count followed by an occurrence list, indicating positions in the database where the term occurs (see figure 3). The occurrence count is never bigger than the space remaining in the index block. Since an entry is always at least three words, up to two words at the end of an index block may be unused. By using the occurrence counts as relative pointers, we may treat an index block as a linked list of entries. The length of this linked list is at most $\lfloor B/3 \rfloor$.

Overall, the index is ordered first by term symbol and then by database position, and divided into index blocks as appropriate. If an entry for a particular term symbol would not fit in a single index block, it is divided into multiple entries and stored in a range of index blocks.

Disk space allocated for index blocks is treated as a circular array. The index may start at any point in the circular array and does not completely fill it. The dynamic update algorithm operates by modifying the index according to an update list and writing the modified index into the unused portion of the array, releasing storage used by the unmodified index as it is modified and written back (figure 4).

Main memory is used for three purposes:

1. Working storage for retrieval operations (M_{RCache}). This storage is largely a cache of index blocks for solving queries, but also includes memory for queries and partial results.

2. Storage for the *index map* (M_{RMapS}). Each entry in the index map describes a block in the index storage.

3. Working storage for the update algorithm (M_{RUpdate}). This storage buffers updates until they are applied to disk.

The index map contains a two-word description of each block in the index, giving $M_{\mathrm{RMapS}} = 2M_{\mathrm{DIdxS}}/B$. The index map itself has size M_{RMap}, with the relation

$$\frac{M_{\mathrm{DIdx}}}{M_{\mathrm{DIdxS}}} = \frac{M_{\mathrm{RMap}}}{M_{\mathrm{RMapS}}}$$

The first word of each index map entry contains the first term symbol indexed in the block, and the second word contains the first posting for this term symbol indexed by the block. A binary search of the index map allows the determination of the range of index blocks containing the postings list for any particular term. This postings list may then be read with a single disk access. If we are interested only in postings within a limited range of values, either because of restrictions placed by the client or because of earlier partial query results, keeping the first posting in the index map assists in applying this restriction, particularly if the indexing for the term is divided across multiple index blocks.

Space must be allocated on the disk for the storage of a non-volatile copy of the index map. A small amount of additional space must be allocated for a *configuration block* containing parameters describing the disk layout: the start and end locations of the updated and yet-to-be-updated segments of the index. Consistency of the file structures requires that a write of the configuration block be atomic. This space is allocated to a single physical block on the assumption that a write of a physical block will either correctly overwrite the block or will not change the block.

5 Update Application

Updates are buffered in main memory until they can be applied to disk. A background process continuously cycles through index storage applying updates and re-writing the index. Update throughput is thus a function of the size of the main memory buffer and the period of an update cycle.

The organization of the index makes update application a simple process. Modified index blocks are written into the free portion of the index storage without affecting the consistency of the index itself. The free portion of the main memory index map is modified in parallel.

At any point in time, the index blocks are grouped into two segments: an updated segment and a yet-to-be-updated segment. The index map parallels this structure. During a retrieval operation the appropriate segment is selected before performing the binary search to determine the actual blocks to retrieve. In the rare case of an access to the term that stretches across the segments, two accesses to disk are required.

When all available free space is consumed with updated index blocks, or at any other time deemed appropriate, the index is checkpointed and a range of index blocks is freed. Checkpointing consists of five steps:

1. Update the non-volatile copy of the index map. This involves changes only to portions of the map that are currently free.

2. Disable query access.

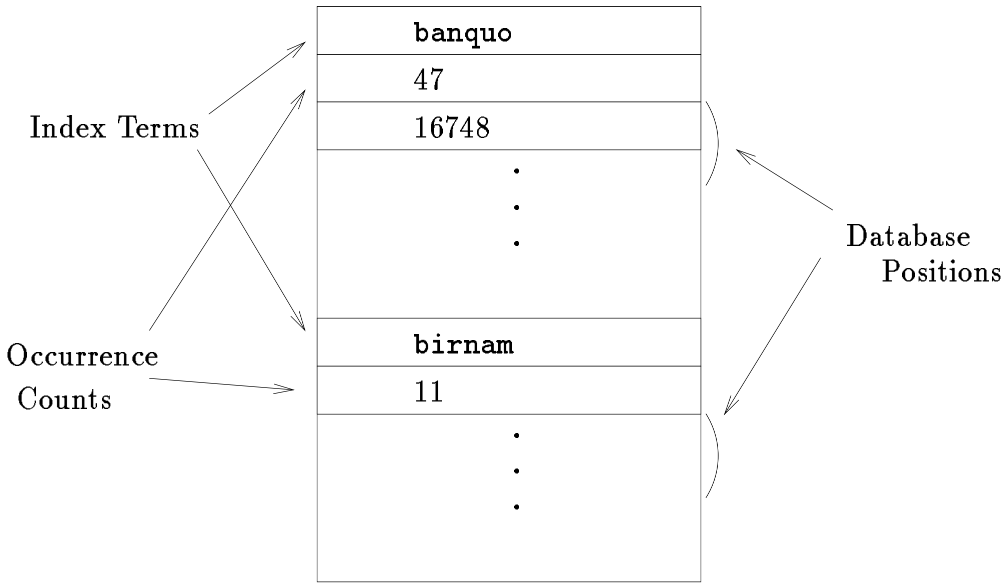

Figure 3: Index block organization.

3. Invalidate cached indexed blocks that are about to be freed.

4. Re-write the configuration block with new file structure parameters.

5. Enable query access.

Steps 3 and 4, which are performed with query access disabled, proceed quickly as they involve only minor modifications to main memory data structures and a single write to disk. A failure before step 4 leaves the index in a consistent state corresponding to the previous checkpoint. A failure after step 4 leaves the index in a consistent state encorporating the updates.

6 Update Management

We assume that update operations are used by an external maintenance agent responsible for maintaining the overall state of the database. The maintenance agent may run on the Marshaller/Dispatcher and may multiplex many sources of updates into a single source. For the purpose of query operations, updates should take effect as soon as they are received from the maintenance agent, but latency in applying the updates to disk is acceptable. The maintenance agent is acknowledged when outstanding updates are applied to disk.

Updates are buffered in main memory until they are applied to disk. During this time, we use the collection of unapplied updates as an auxiliary database, modifying the results of retrieval accesses into the main index. When buffered in memory, we assume that information relating to the update will occupy four words: three words for the update parameters and one word to encode the operator type and the information necessary for acknowledging the update.

Global operations and local operations are maintained in separate data structures, the *global update list* and the *local update list* respectively. We require identical properties for both data structures:

- Sequential access to updates in sorted order. The global list is sorted by start range and scanned once per index term during an update cycle. The local list is sorted by index term and then by database position and scanned once per update cycle.

- Insertion and deletion.

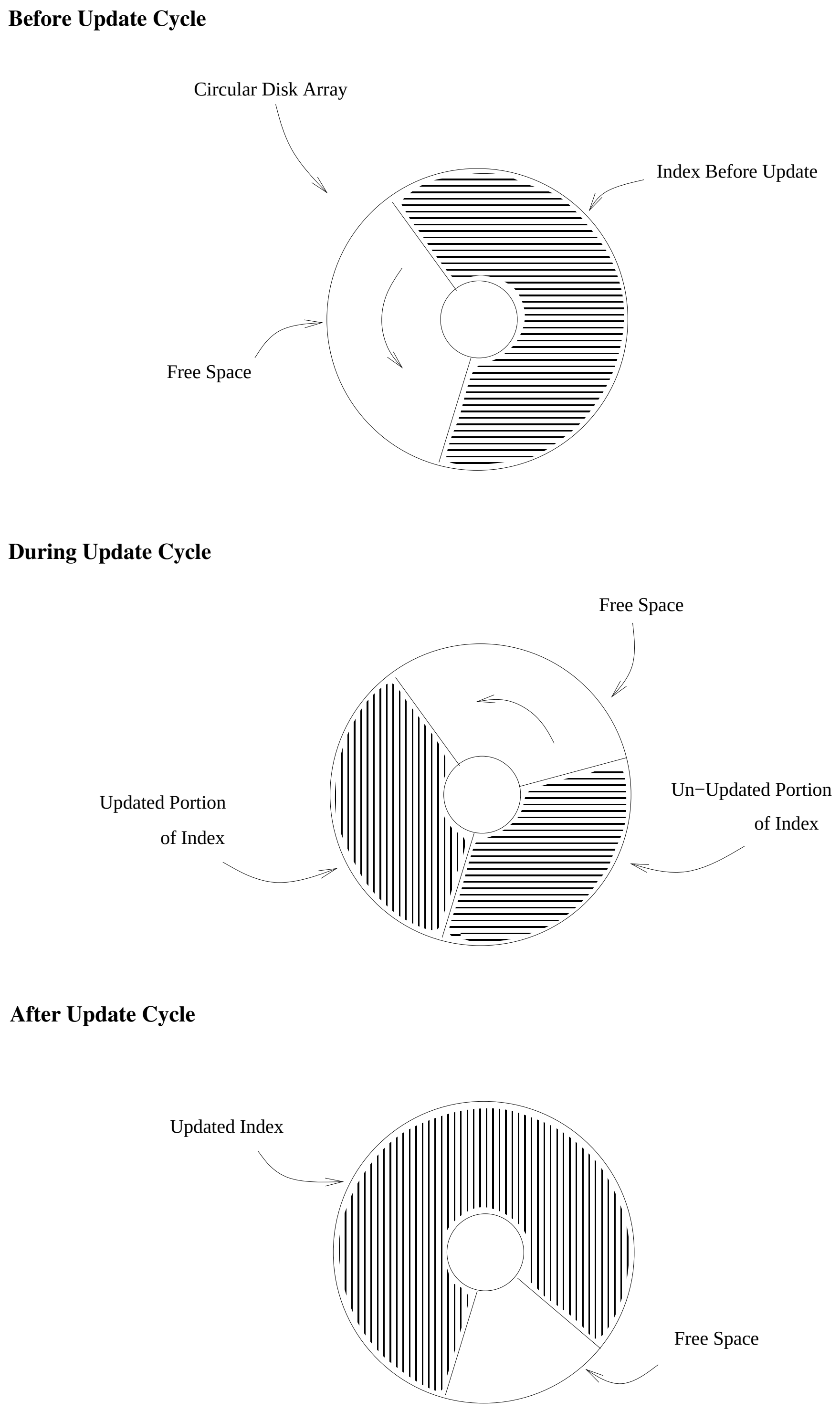

Figure 4: Overview of an update cycle.

- Search. Results from querying the main index will be modified by searches in the local and global update lists.

It must be possible to perform these operations concurrently — sequential scan is an on-going process, updates are continuously arriving from the maintenance agent, retrieval requests are constantly being processed for clients.

The update process acknowledges and deletes updates at each checkpoint. It is possible that a failure after a checkpoint but before updates are acknowledged might result in the maintenance agent having an incorrect view of the index engine's file structures. Idempotency of the update operations ensures that these updates could be re-applied without harm.

Skip lists [8, 9] provide a simple and nearly ideal implementation for the update lists. For the purposes of sequential access, skip lists are effectively linked lists. Insertion, deletion and search are all $O(\log n)$ operations in the average case. Concurrent access to skip lists is simple and efficient. Overhead for pointers is in the range of one to two words per buffered update (depending on an implementation parameter). Total storage overhead for a buffered update is thus at most six words. Data structures other than skip lists may be used, but Pugh's discussion of concurrent maintenance of data structures [8] provides a strong argument for skip lists.

Global operations represent deletions of ranges of text; local operations represent changes to individual postings lists. Adding a document requires many local operations, but deleting the same document requires only a single global operation. For these reasons, global operations tend to be few in number and local operations tend to be relatively many in number. Global operations must be buffered in main memory, but if there is no requirement for updates to take immediate effect there is no need to buffer local updates in main memory. Instead, the local updates may be sorted by the maintenance agent and presented in mulitple batches to the index engine over the course of an update cycle. If this technique is used, disk access is no longer a factor limiting maximum update throughput.

Word Size	64 bits
M_{D}	2.1 GB
M_{R}	51 MB
M_{DIdx}	1 GB
M_{DIdxS}	2 GB
M_{B}	64K
M_{RIdx}	256 KB
M_{RIdxS}	512 KB
M_{RUpdate}	14 MB
M_{RCache}	36 MB

Figure 5: Retrieval system parameters.

7 Implementation Measurements

The update strategy described in this paper has been implemented as part of the MultiText project. However, we do not at this time have experience with the strategy on a large scale under production loads.

The most unusual aspect of our approach is the continuous re-organizational cycle through the index. In order to convince ourselves that this strategy is feasible for use in a production environment, we have performed experiments to understand the possible performance impact of continuous update under a heavy retrieval load. We assumed the worst case: continuous random query accesses with no hits in the cache. Each query access reads a single index block. We varied the update cycle time and examined its effects on the sustainable query access rate. To provide baselines for the measurements we determined the maximum sustainable access rate that could be achieved with no concurrent update and the minimum possible update cycle time with no concurrent query access. Parameters of the retrieval system for the experiments are given in figure 5. Not all of these parameters are strictly relevant to the experiment; the values for M_{RCache} and M_{RUpdate} should be taken as nominal. The experiments were run on a dedicated DEC Alpha 2000-300 running OSF/1 V1.3 with a 2.9 GB Seagate ST43400N SCSI disk.

The results of the experiment are shown in figure 6. The irregular spacing of data points along the horizontal dimension is a result of our inability to finely control the update cycle period in the current implementation. With an update cycle time of as little as 45 minutes, query performance degrades by only 23%. With an update cycle time of 3 hours, query performance degrades by only 7%.

We use the nominal value of 14 MB for M_{RUpdate}. Of this quantity, 2 MB is used for merging updates — index blocks are read and written by the update process in groups 1 MB in size. The remaining 12 MB is used for buffering updates. Since each update requires six words (and each word is 8 bytes), this 12 MB allows 256 K of updates to be buffered. Applying 256 K of updates over a 3 hour update cycle gives an update throughput of 24 updates/sec.; if applied over a 45 minute update cycle the update throughput is 97 updates/sec. For comparison, the reader should consider the maximum query access rate and consider the throughput of applying each update individually to a B-Tree file structure.

8 Further Issues

8.1 Index Compression

Index compression integrates smoothly into the scheme. Each index block is individually compressed to a variable-length segment. The index map references compressed blocks rather than fixed-size blocks. We add a word to each entry in the index map that indicates the offset of the compressed block in the index. Blocks are decompressed as they are brought into the cache or read by the update process. Since all blocks are cyclically re-written, compression does not hamper update.

8.2 System Considerations

This paper has concentrated on the design of the index engine. We look briefly at a few relevant aspects of the remainder of the system.

The text server translates a range in the database into the associated text. While the details differ, we organize the text server using data structuring principles similar to those used in the index engine.

While our data structures efficiently implement the inverted index data abstraction, they do not efficiently implement queries that are based only on the dictionary. Generally, these queries consist of identifying terms that match specified patterns. In systems that separate the dictionary from the index, this type of query can be satisfied by a dictionary search. Besides its other duties, the Marshaller/Dispatcher is responsible for handling these dictionary-based queries by maintaining a separate dictionary database of all words in the system. In addition, the Marshaller/Dispatcher is responsible for implementing a term thesaurus.

8.3 Wider Application

While our exposition has been in the context of a distributed information retrieval system the data structures and update strategy have wider applicability. Inverted indexes are used in applications other than information retrieval. Even if file structures are static and update is not a requirement (in the case of a CD ROM, for example) our data structures provide an efficient realization of inverted lists. The update strategy has applicability to other databases with similar update characteristics, with the text server being a ready example.

9 Conclusions

The data structures presented in this paper efficiently realize the inverted index data abstraction. A retrieval operation requires a single disk access in all but rare cases. The update strategy provides high throughput with little impact on retrieval performance. The file structures may be compressed to increase the size of index that can be stored on available disk. Although discussed in the context of a distributed full-text information retrieval system, the results of this paper have applicability to any use of inverted indexes and any database with similar update characteristics.

Acknowledgements

The Multi-User Multi-Server Very Large Text Database project is funded by the Government of the Province of Ontario through its Information Technology Research Centre. The Natural Sciences and Engineering Research Council of Canada and the University of Waterloo provided additional financial support.

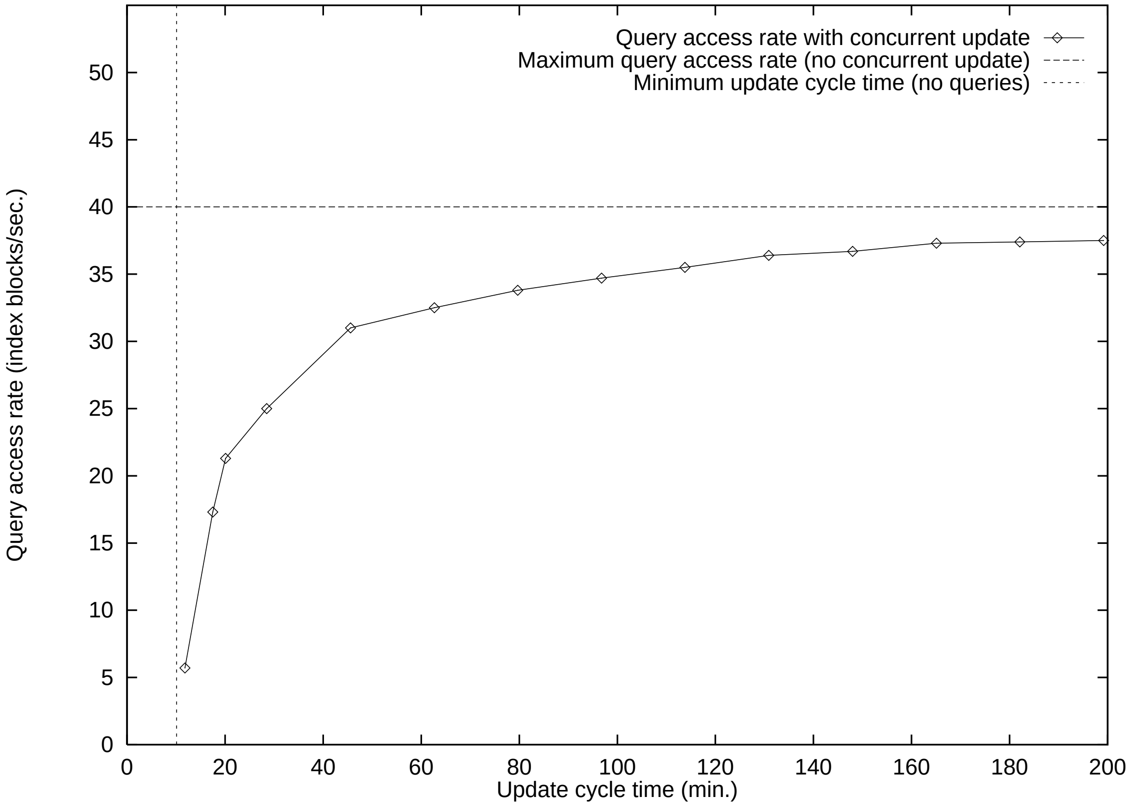

Figure 6: Performance impact of continuous update cycles.

References

[1] Forbes J. Burkowski. Retrieval performance of a distributed text database utilizing a parallel processor document server. In *Proc. 2nd Inter. Symp. on Databases in Parallel and Distributed Systems*, pages 71–79, 1990.

[2] Forbes J. Burkowski. Surrogate subsets: A free space management strategy for the index of a text retrieval system. In *Proc. 13th ACM/SIGIR Conference*, pages 211–226, Brussels, 1990.

[3] Charles L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. Technical Report CS-94-30, University of Waterloo Computer Science Department, Waterloo, Ontario, Canada, N2L 3G1, September 1994. Available by anonymous ftp in `/cs-archive/CS-94-30` on `cs-archive.uwaterloo.ca`.

[4] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proc. 13th ACM/SIGIR Conference*, Brussels, 1990.

[5] Christos Faloutsos. Access methods for text. *Computing Surveys*, 17(1), March 1985.

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

[7] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Massachusetts, 1973.

[8] William Pugh. Concurrent maintenance of skip lists. Technical Report TR-CS-2222, Dept. of Computer Science, University of Maryland, College Park, Maryland, April 1989.

[9] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, June 1990.

[10] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer.* Addison-Wesley, Reading, MA, 1989.

[11] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*, chapter 2, pages 24–51. McGraw-Hill Computer Science Series. McGraw-Hill, New York, 1983.

[12] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *2nd Inter. Conf. on Parallel and Distributed Information Systems*, pages 8–17, San Diego, January 1993.

[13] Steven Wartik. Boolean operations. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, chapter 12, pages 264–292. Prentice Hall, Englewood Cliffs, NJ, 1992.

[14] P. Weiss. *Size Reduction of Inverted Files Using Data Compression and Data Structure Reorganization.* PhD thesis, George Washington University, 1990.