Low Latency Index Maintenance in Indri

Trevor Strohman
strohman@cs.umass.edu

W. Bruce Croft
croft@cs.umass.edu

Center for Intelligent Information Retrieval
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

ABSTRACT

There has been a resurgence of interest in index maintenance (or incremental indexing) in the academic community in the last three years. Most of this work focuses on how to build indexes as quickly as possible, given the need to run queries during the build process.

This work is based on a different set of assumptions than previous work. First, we focus on latency instead of throughput. We focus on reducing index latency (the amount of time between when a new document is available to be indexed and when it is available to be queried) and query latency (the amount of time that an incoming query must wait because of index processing). Additionally, we assume that users are unwilling to tune parameters to make the system more efficient.

We show how this set of assumptions has driven the development of the Indri index maintenance strategy, and describe the details of our implementation.

1. INTRODUCTION

One of the biggest successes of information retrieval research has been the development of efficient full-text indexing systems. The compressed inverted list index allows for quick retrieval of documents from very large corpora. Web search engines have put this technology to public use, and now hundreds of millions of people use full-text search every day.

Most research into text indexing assumes that the collection of documents to be searched is unchanging. However, it is difficult to imagine many useful scenarios where this is a reasonable assumption. Web documents, news feeds, e-mails, corporate documents, and even library book collections are constantly changing. Even if the documents themselves are static, most collections of documents grow over time, and the newest documents are often the most relevant. Handling this changing nature of data is an important area of information retrieval research.

The process of updating information in an existing index is called both index maintenance and incremental indexing in the literature. This is a small but active field of research. However, the evaluation metrics used in most papers is on the throughput of the indexing process. We claim that indexing throughput is not the critical factor in many real scenarios. Modern information retrieval systems can handle as much as 50GB per hour of new text data on a single machine [4], and other than the web, few data sources generate new or changed text at that rate.

By contrast, consider a search engine storing data from a particular newswire. The TREC[1] AP89 collection consists of 84,678, for an average of 261 new news stories a day. This collection is small by current standards, and the rate of information change cannot be considered taxing for any modern system. Raw indexing throughput is not the most important measure for this application. We argue that latency is a better measure–the amount of time taken between the appearance of a document on the wire until it is able to be searched, and the amount of time that incoming queries must wait while the indexing process for a new document is taking place.

We have built an incremental indexing system that focuses on this latency problem. The remainder of this paper focuses on describing our system and relating it to previous work. We do not have an evaluation of our system at this time, but we hope that this paper will spark additional work in the future.

2. RELATED WORK

While there has always been practical commercial interest in incremental indexing, academic work on this topic has been sparse. A small pool of researchers considered this topic between 1990 and 1995, then publications nearly stopped. In the past three years, new research has arrived to build upon and update the assumptions in earlier work.

Cutting and Pedersen [7] present the first incremental indexing work that we consider here. The authors use a variety of methods to store posting lists so that they may be dynamically updated. In the first method, they store postings (word and document location pairs) directly in the B-Tree, sorted first by word and second by document location. This straightforward approach is used more recently by the MySQL database engine for full-text search [1]. This

[1] http://trec.nist.gov

OSIR'06, August 11, 2006, Seattle, Washington, USA.
ACM .

Disk Model	Transfer rate	Seek time	Year	Size
Seagate ST1980	3.55 MB/s	9.9 ms	1994	1GB
Seagate ST3146854FC	80 MB/s	3.5 ms	2004	147GB

Table 1: Performance figures for two Seagate hard disk models produced in the last decade.

method is simple and conceptually clean, but leaves room for improvement in both space and speed. The authors improve on space utilization by storing the word only once, instead of in each posting. Then, they improve on speed by using an external heap file to store list data instead of storing the data within the tree itself.

Tomasic, Garcia-Molina and Shoens [14] focus on the storage allocation policy in the inverted file. As in Cutting and Pedersen's approach, the inverted file is a heap that requires an allocation policy. Clearly the individual lists are expected to grow over time, but leaving large gaps in the file for extra postings wastes space and time (since the extra file space implies longer seek times between relevant data regions). The authors explore three allocation policies: constant, block, and proportional. In the constant case, each inverted list update operation reserves a constant amount of extra space for new postings. The block strategy is similar, except the extended list is forced to end on a block boundary. The proportional strategy increases leaves some percentage of the total length of the list as empty space; this means that longer lists have more room to grow. Additionally, the authors consider three update policies; new, whole and fill. The new strategy writes new postings to a new location, effectively making each inverted list a linked list of segments. The fill strategy is similar, except each linked list segment is forced to be the same length. Finally, the whole strategy requires that each inverted list be copied at every update, so that lists remain contiguous. Not surprisingly, the authors find that the *new* strategy is quicker for updates, while the *whole* strategy is preferred for queries.

Brown, Callan and Croft [2] investigate similar approaches to those of Tomasic et al. [14]. The authors modify the INQUERY retrieval system to store its data in the Mneme object store [11]. This abstraction of the storage layer is similar to more recent work, such as de Vries et al. [8]. Brown et al. store inverted lists in this disk-based object store; small lists are segregated from large ones, and small lists are allocated in power-of-two sized blocks. Large lists are not necessarily stored sequentially, but may be stored in a linked list of blocks. The results are largely similar to Tomasic et al. [14].

Clarke et al. [5] present a system which, in contrast with the others shown so far, explicitly discusses query activity while new documents are added to the collection. Unlike the work shown in this paper, this method still adds documents to the index in batches, but these updates are committed quickly to the index, so that pauses in query operations are as short as possible.

The four papers mentioned here represented the state-of-the-art for index maintenance until recently. However, these papers were written in the mid-1990s; since that time, computer technology has changed drastically.

The disk models shown in Table 1 are a small sample of the disks available on the market, but they represent the progress that drive makers have achieved over the last decade. Seek times have dropped by approximately 60%, but transfer rates have increased by as much as 20 times. Just looking at these figures, it seems that disk seeks have become significantly more costly relative to disk transfers.

This comparison is not completely fair, because average seek time is computed as the average time to seek between two random locations on the device. Since the new devices are bigger, the 'average' seek time represents a jump over much more data. In the 1994 disk, for example, an average seek might cross 500MB of data, while on a newer disk it might cross 70GB of data. Therefore, we expect that if the same size document index was used on both drives, the seek time difference would be more dramatic.

A more subtle change is in the amount of time necessary to read the data of the entire disk. For the 1994 drive, the entire contents of the drive can be transferred in about 4 minutes. The 2004 drive requires 30 minutes for the same task. If we assume that collection sizes keep pace with disk sizes, we see that transfer rates are not keeping up with increasing data size.

More details on the changes in retrieval system performance can be found in Zobel, Williams and Kimberly [15]. The important factor is that research that relies on disk performance must be revisited in order to maintain relevance.

Lester et al. [10] revived interest in the area of index maintenance with a 2004 study on the relative advantages of three strategies, in-place, re-build and re-merge. The in-place strategy is similar to Tomasic's *whole* strategy with a proportional allocation policy. The re-build strategy simply rebuilds the entire collection, while re-merge merges new postings into an existing index, forming a new index. The authors find that for updates of less than 10000 documents, both incremental strategies are better than rebuilding the index. Furthermore, for the smallest updates (under 100 documents), the in-place strategy is faster. However, for these small updates, the update time per document approaches 1 second.

Especially because of the issues in transfer time, a new class of update algorithms has been discussed in the literature recently. This strategy, called *geometric partitioning* by Lester et al. [9], does not force inverted lists to be merged together when postings are flushed to disk. Instead, new postings are flushed to disk in entirely new indexes, called *partitions*. Queries acting upon this data must check each partition for inverted list data, so there is a query speed penalty for maintaining too many partitions. Therefore, these partitions can be merged together to form larger partitions that are more efficient to query. Since merging is costly, an efficient system must balance the needs of query processing and efficient indexing in choosing its merging policy.

The name geometric partitioning refers specifically to a merging policy developed by Lester et al. [9]. For a system that can hold b document pointers in memory and some positive integer parameter r, the i^{th} partition is limited in size to br^i pointers. For example, if $r = 3$, the first disk partition is limited to holding 3 times as many documents as the system memory can hold; if this partition grows beyond that size, its data is merged into the second partition (which is limited to 9 times the system memory). By keeping this exponential distribution of partition sizes, the total number of indexes is kept small while still making the common case (merging in-memory data into the first partition) fast.

Many authors ([3], [4], [9], [12]) have studied the parti-

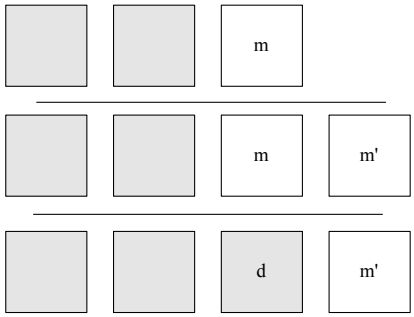

Figure 1: Three steps in the process of adding a new memory partition. First, a new partition m' is added. Next, m is written to disk asynchronously. Next, d is made available for new queries. Finally, when m is no longer in use by any query, it is deleted.

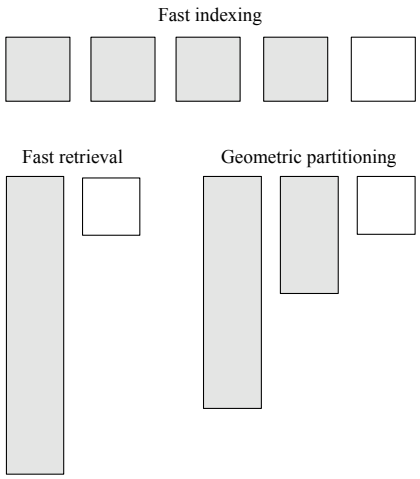

Figure 2: Three possible partition merging policies. The 'fast indexing' policy avoids merging partitions as much as possible; this method approximates a traditional batch indexer. The 'fast retrieval' policy eagerly merges new data, making a single large disk partition. The 'geometric partitioning' strategy finds a balance between the other two policies.

tioning strategy with good results. However, previous work has focused on throughput instead of latency of operations. In this work, we consider latency as our first goal.

3. IMPLEMENTATION DETAILS

3.1 Overview

Our implementation is a part of the Indri search engine [13]. The implementation described here has been in the toolkit since the 2.0 release.

Indri is a component of the Lemur toolkit for Information Retrieval [6]. The Lemur toolkit is an open-source system built to enable Information Retrieval research using language modeling. Indri extends upon the goals of the Lemur toolkit by adding new structured document retrieval functionality and a flexible query language. In addition, Indri is built to be used by both researchers and industrial users.

One common use of Indri is to index news feed data. Like most search engines on modern hardware, it is not difficult for Indri to keep up with the volume of data generated by a news feed. However, it is desirable that new news articles be made available immediately. Furthermore, the addition of a new news article should not be able to halt query processing for a noticeable period of time. This set of requirements drove our implementation of a concurrent version of Indri.

3.2 Index design

As in the work of Lester et al. [9], Indri uses index partitions to maintain high query and indexing throughput. In many previous systems, data stored in memory was not considered fully indexed, and data needed to be flushed to disk in order to be queried. This is not the case in Indri; the first partition is stored in RAM, and can be queried directly. This allows Indri to support high-speed indexing and query processing for smaller collections without ever requiring disk seek delays.

For collections that grow larger than memory can hold, Indri writes partitions to disk. Once a partition has been written to disk, the data will never be changed–it may be merged into another partition and then deleted, but never changed. This allows disk data to be read without the need for locks.

In memory, term statistics and vocabulary are stored in a hash table. On disk, vocabulary information is stored in two B-Trees, with one dedicated to frequent terms and the other dedicated to infrequent terms. The frequent terms tree (consisting of terms that appear more than 1000 times in the collection) is small enough that it always fits easily in memory. The infrequent terms tree is larger than system memory for large collections. The inverted list information for all terms is stored in a single file, sorted by alphabetical order of term. The postings in each list are stored in document order.

To keep index latency as low as possible, we do not let disk I/O block indexing operations. All new documents are added to the in-memory partition. A monitoring thread checks periodically to see if the in-memory partition m has grown too large. If so, a new memory partition m' is created to accept new documents (Figure 1). Then, m is written to disk in a background thread. While m is being written to disk, it remains available for query processing. When the disk write completes, m has an on-disk representation, d, which will be used in place of m for all future queries. However, m may still be in use by some long-running queries. When all of those queries have completed, m is safely deleted.

In other systems, it is common for the parsing and indexing functions to be intertwined. Each word is parsed from the document, then added immediately to the index. In Indri, adding documents to the memory index requires a lock to prevent a query processing thread from seeing inconsistent data. Therefore, to keep index lock duration down, Indri parses each document completely (while not holding index locks) before adding document data to the index.

3.3 Query processing

Query processing is made more complicated by the existence of many index partitions. In Lester et al. [9], in-

verted lists for each query term are materialized in memory by copying data from each fragment into memory, then processing the combined list. Indri works differently; each partition is treated as a separate index. In the first phase of query processing, term statistics (such as term frequency) are collected from all partitions. In the second phase, the query is processed separately on each index partition, starting with the oldest data. When all partitions have been visited, query processing completes.

Since all data on disk is immutable, no special locking is necessary to access it. Indri allows many queries to execute simultaneously on this read-only data. When query processing reaches the most recent data (in the in-memory partition), the query processor must acquire a lock to ensure that the memory partition does not change while query processing occurs. The query processor acquires a read lock, so that other queries can operate on the in-memory data simultaneously. While this read lock does cause an increase in index latency, this latency can be reduced arbitrarily by decreasing the size of the in-memory partition.

Since merging happens asynchronously (as described in the indexing section), long-running queries can continue to execute while merging is in progress. The only point of contention between indexing and querying operations happens in the in-memory partition, where I/O operations are never performed while a lock is held.

3.4 Merging policy

In other recent index maintenance work, authors have proposed particular fixed merging policies. Parameters can be tuned in these policies to produce higher query throughput or higher indexing throughput.

However, in the kind of cases we are considering, we expect the optimal policy to change over time. In a desktop search application, we may find the system flooded with new data as a user downloads a large batch of documents onto the system. When this large batch of documents arrives, it is advantageous to tune the system for optimal indexing performance. However, later in the day, a user may start running queries against the data (while no new data is entering the system). At this point, optimum query performance would be preferred.

Indri attempts to pick a merging policy based on the weighted load average of the system over the previous 15 minutes (Figure 2). Every query is counted, as is every added document. When no queries have been executed in 15 minutes, the system is optimized for document indexing, and the system only merges index partitions when it is close to running out of file handles. Similarly, when no documents have been added for 15 minutes, the system chooses to merge data as often as possible in order to make the index structures optimized for query processing. Load between these two extremes causes the system to find a balance between query speed and indexing speed.

Our performance tuning so far has caused us to choose to merge partitions when:

- More than one partition exists, and
- The system has not thrashed in the last 5 minutes (that is, has not had to halt document additions because the system could not write them to disk quickly enough), and:
  - either:
    $$\frac{\text{documents added per second}}{50 \times \text{queries per second}} < |\text{partitions to merge}|$$
  - or there is very high query load, or
  - or there are very few documents being added.

This formulation is clearly *ad hoc*, and we have not performed a detailed analysis to determine how close to optimal this policy is. We do know that it seems to work well for the kinds of tasks we have tried. It works well enough that we have not seen the need to expose these parameters to the user.

4. CONCLUSION

While most work in index maintenance has focused on maximizing throughput, we considered the problem of reducing latency in a concurrent information retrieval system. Our system, Indri, contains a concurrent indexing and retrieval system built to reduce query and indexing latency, while making intelligent decisions about merging based on workload.

Our goal with this work is to stimulate new thinking about workloads and applications for index maintenance, particularly in evaluation. In the future, we hope to evaluate our system merging policy and latency in order to find ways that our implementation can be improved.

5. ACKNOWLEDGMENTS

This work was supported in part by the Center for Intelligent Information Retrieval and in part by NSF grant #CNS-0454018 . Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the sponsor.

6. REFERENCES

[1] MySQL. `http://www.mysql.com/`.

[2] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192 – 202, Santiago, Chille, September 1994.

[3] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time trade-offs in dynamic information retrieval systems. In *CIKM 2005: Proceedings of the 14th ACM Conference on Infromation and Knowledge Management*, Bremen, Germany, Nov. 2005.

[4] S. Büttcher and C. L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In *ECIR 2006: Proceedings of the 28th European Conference on Information Retrieval*, London, UK, Apr. 2006.

[5] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. Fast inverted indexes with on-line update. Technical Report CS-94-40, University of Waterloo, Waterloo, Canada, 1994.

[6] W. B. Croft, J. Callan, J. Allan, C. Zhai, D. Fisher, T. T. Avrahami, T. Strohman, D. Metzler, P. Ogilvie, M. Hoy, J. Lafferty, J. Brown, L. Si, K. Collins-Thompson, M. Bilotti, F. Feng, and L. Larkey. The Lemur Project. `http://www.lemurproject.org/`.

[7] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[8] A. de Vries, J. List, and H. Blok. The multi-model DBMS architecture and XML information retrieval. *Intelligent Search on XML, Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence (LNCS/LNAI)*, pages 179–192, 2003.

[9] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In A. Chowdhury, N. Fuhr, M. Ronthaler, H.-J. Schek, and W. Teiken, editors, *Proceedings of the ACM CIKM Conference on Information and Knowledge Management*, pages 776–783, Bremen, Germany, Nov. 2005.

[10] N. Lester, J. Zobel, and H. Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In V. Estivill-Castro, editor, *Proceedings of the Australasian Computer Science Conference*, pages 15–22, Dunedin, NZ, Jan. 2004.

[11] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, 1990.

[12] T. Strohman. Dynamic collections in Indri. Technical Report IR-426, University of Massachusetts Amherst, 2005.

[13] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *IA 2005: Proceedings of the 2nd International Conference on Intelligence Analysis*, 2005.

[14] A. Tomasic, H. García-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD 1994: Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 289–300, Minneapolis, Minnesota, 1994.

[15] J. Zobel, H. E. Williams, and S. Kimberley. Trends in retrieval system performance. In *ACSC*, pages 241–248, 2000.