An Inverted Index Implementation Supporting Efficient Querying and Incremental Indexing

Ajith Nagarajarao, Jyothir Ganesh R., Abhishek Saxena
{ajith, jyothir, asaxena}@cs.wisc.edu

May 6, 2002

Abstract

We have implemented an inverted index as a part of a mass collaboration system. It provides the facility to search for documents that satisfy a given query. It also supports incremental updates whereby documents can be added without re-indexing. The index can be queried even when updates are being done to it. Further, querying can be done in two modes. A normal mode that can be used when an immediate response is required and a batched mode that can provide better throughput at the cost of increased response time for some requests. The batched mode may be useful in an alert system where some of the queries are can be scheduled. We have implemented generators to generate large data sets that we use as benchmarks. We have tested our inverted index with data sets of the order of gigabytes to ensure scalability.

1 Introduction

The inverted index is a data structure that is widely used to support efficient querying on a large text corpus. The index associates sets of documents with tokens. Each document in the corpus is represented in a typical index by a postings entry. A postings entry is an ordered pair of the form <LRID, metadata>. The LRID contains the id of the document that is represented by this postings entry. The ordinate, metadata, is used to store information pertaining to this document such as the frequency with which the token appears in the document, the offset in the document at which the token appears, etc. Searching can then be done on the basis of tokens by looking up the desired token in the inverted index and returning the set of postings associated with the token.

In addition to supporting searching on the basis of tokens, inverted indexes must also provide mechanisms to update the postings associated with a token. Postings, in general, may be added or deleted. Further, the metadata for a posting may be updated. All these updates will have to ultimately be reflected in the inverted index. In a typical inverted index, the number of updates is very small compared to the number of queries and can be clustered and merged with the index either at periodic intervals or when the number of updates crosses a predefined threshold.

In the remainder of this report, we will describe in detail our implementation of the inverted index. We start by describing in section 2 the related work done by others. In section 3, we describe the architecture of our inverted index, the searching and updating functions we have implemented and the benchmarks we have generated and used. The actual implementation is discussed in section 4. The results we obtained for varying sizes of data sets are explained in section 5. In the last section, we conclude by talking of features that could have been added to our inverted index given sufficient time.

2 Related Work

A lot of work has been done on inverted indexes and various implementations of this data structure have been suggested by many different authors. In this section, we will describe related work done by others and their relevance to our project.

In their work on compression and fast indexing, Moffat et al. [1] describe certain compression techniques for inverted indexes as well as data and show that the compression performance does not degrade with increase in the size of the data sets and that the response time can still be bounded. However, they make the assumption that the database is static and use Huffman codes to compress it.

Brown et al. [2] propose an incremental indexing technique in which they describe the support they provide for updates to the inverted index data structure by building it on top of a persistent store. They use the data management facilities of the object store to achieve this goal. A similar proposal [3] describes an architecture with a process dedicated to applying updates to the inverted index and making sure that all the shared data structures are maintained in a consistent state.

File based inverted indexes typically use the filesystem, support provided by operating systems. Knight and Hamilton [4] describe one such implementation that uses the unix filesystem. The advantage of these systems is that they are reliable and are optimized for rapid retrieval of stored data. A careful use of filesystem support can lead to good response times.

The Google search engine [5] is a very widely used search engine for querying on hypertext data. It uses a wide variety of data structures to support efficient querying. Each document in Google is identified uniquely. Information about documents is kept in a fixed width ISAM called the document index. Google features a lexicon that is kept completely in main memory. Google also incorporates a page rank feature to rank the search results by order of relevance.

Tomasic et al. [6] describe a dynamic dual datastructure used to implement inverted indexes. The index then dynamically separates long and short inverted lists and attempts to optimize each type of list separately.

Inverted files and signature files are two ways to implement indexing. Zobel and Moffat [7] compare these two methods and conclude that inverted indexing in an inherently superior method as they take less time to evaluate typical queries and also take less space.

Lucene [8] is an on going, open source search engine project implemented in the Java programming language. It is designed to support incremental indexing and aims to support applications that can search mail, online documentation and websites.

Various optimizations for inverted index maintenance are discussed by Cutting and Pederson [9]. In particular, they introduce two optimization techniques: the merge update which they show to be better than straight forward block updates and pulsing which reduces the amount of space required without reducing performance.

In their work on searching large lexicons, Zobel et al. [10] describe how to use an in-memory lexicon to search a compressed inverted index for partially specified terms. They show that it is possible to get an effective compromise between speed and space. Using their approach, they show that these types of queries are much faster than brute force searches and require less memory compared to other pattern-matching data structures.

Compressing inverted indexes typically leads to an increase in CPU time required to respond to a query. However, Zobel and Moffat [11] describe a technique of storing an internal index in each compressed inverted list that can be used to reduce the processing time required.

Berry et al.[12] explore the use of underlying association between words in their work on latent semantic indexing.

The use of a relational database system to hold an inverted index is described by Putz [13].

Use of large data generators for benchmarking xml is described in [14].

3 Inverted Index

3.1 Architecture

Our token index can be depicted schematically as shown in Figure 1. At the topmost level, the token index is responsible for handing search queries as well as updates.

For search queries, this module sends the token to the Searcher/Postings Manager. Updates, in turn, are passed on to the update manager. The main functionality of the searcher/postings manager is to handle search queries. The lexicon module is located in main memory and maintains associations between tokens and their corresponding postings. The lexicon can be queried to obtain the postings corresponding to a token.

For update queries, the Tokenindex module sends the update to the Update Manager. This module is

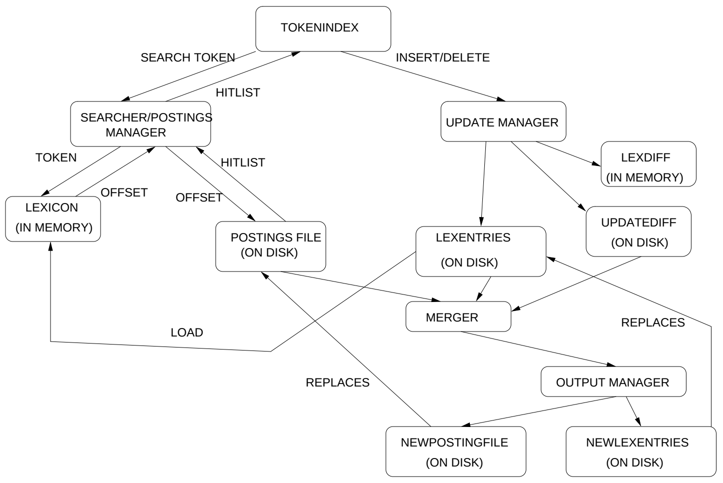

Figure 1: Architecture of the Token Index

responsible for maintaining all the updates and ensuring that information does not get stale by merging the updates with the token index. It does this by maintaining three data structures viz., the lex diff, the update diff, and the lex entries. It then dispatches these data structures to the merger module whenever the updates need to be merged.

The merger module is responsible for performing the actual merging. This module uses the data structures provided by the update manager to merge the various files that hold the update information into the token index. This module is also responsible for resolving conflicts and eliminating duplicates.

The Output Manager module is responsible for ensuring that the new lexicon and postings file are written in the correct format. This module abstracts the internal format of these files and provides a uniform interface for reading from and writing to these files.

3.2 Searching

Retrieving the documents for a token involves the use of the lexicon and the postings file. The lexicon is essentially a hash table mapping tokens to the offsets in postings file where the postings list for the token begins. The searcher looks up the lexicon to find the appropriate offset in the postings file, seeks to that offset, and returns an iterator to the postings list which begins at that offset.

3.2.1 Lexicon

Having the lexicon fit in memory can lead to more efficient searches, but this consumes a lot of main memory. As observed in [5], this is a price worth paying considering that, today, machines with gigabytes of memory are easily available.

Fitting a large lexicon in memory needs specialized data structures to save space. Generic data structures provided by libraries are too space inefficient for this purpose. For example, the hash table provided by the java class libraries takes about 2GB of memory to fit ten million tokens assuming that the average length of token to be six characters.

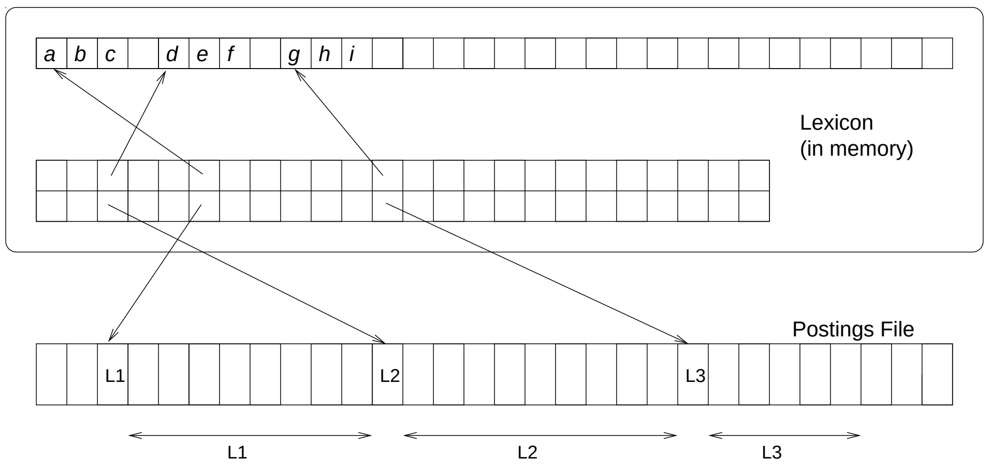

Figure 2: Lexicon

The data structure we used is a variant of the one used in [5], and can fit up to twelve million words of an average length of eight characters in 256MB.

In the lexicon, tokens are represented as null terminated character sequences. The tokens are concatenated together to fit in a character array called the token array. The lexicon maintains a pointer array, an array of pointers to tokens in the C++ version and offsets of tokens in the token array in the Java version. There is also an offset array of same capacity as pointer array, an array of offsets in the postings file. The pointer array, together with the offset array gives a mapping from tokens to offsets.

Given a <token, offset> pair to be inserted into the lexicon, we compute the hashcode of the token. We then compute the hash value based on the hashcode and the capacity of the pointer array. Starting with the hash value as the index in the pointer array, a quadratic probing scheme probes for an empty slot in the pointer array. The token is just concatenated to the current list of tokens in the token array. A pointer to the token or the index of the token in the token array is added to the empty slot found. In the corresponding slot in the offset array, the given offset is added.

Given token to be searched in the lexicon, the hashcode and the hash value of the token is computed as done in for insert. Starting with this hash value as the index in the pointer array, the quadratic probing scheme probes for a slot in the pointer array where the token pointer points to a token matching with the search token. The offset in the corresponding slot in the offset array is returned. If an empty slot is encountered during probing, search immediately returns as unsuccessful.

3.2.2 Retrieving Results

Once the offset for a given token is found, the searcher seeks to that offset in the postings file. The size of the postings list corresponding to the token is stored at that offset, followed by the list itself. The searcher returns an iterator, in the form of an input stream that reads postings, at most as many as given by the size.

The lexicon lookup, being an in memory task, returning an iterator to the postings list for a token requires just one disk read. Retrieving the postings may require more disk reads. Usually, only the top few results are actually retrieved. By keeping the postings sorted based on their rank, all the top postings that fit into a disk block can be retrieved with just one read. While this is the best scheme for single token queries, multiple token conjunctive boolean queries which are very common, require merging the postings lists of two different tokens to retain only the postings found in both. If the docId range is small and the postings lists is expected to be dense with respect to docId's, this can be done efficiently using a bit vector. But, if the docId range is very large, the best strategy is to keep the postings list sorted based on docId and merge them. Since our goal is to scale to tens of millions of documents, we keep the postings list sorted based on docId. A combination of the two schemes can be beneficial. We can store a few top ranked results in the beginning, and the rest sorted based on docId. Merging of the top unsorted results need to be handled specially. This has the dual advantage of getting top results quickly for single token queries and also provides a fast way to merge all the results for conjunctive queries.

3.2.3 Effect of Concurrency and Scheduling

We investigated the effect of concurrent requests on throughput. We expected an increase in throughput even on a uniprocessor. Concurrent search requests from multiple threads will eventually queue up as disk read requests. With many requests queued up at once, the disk scheduler will be able to schedule them in a nearly optimal way. But experiments showed no gain in throughput, possibly because, at a small number of threads, there is not much gain due to scheduling, and at a large number of threads, the gain is offset by context switches.

We also investigated scheduling the searches at a higher level. In an alert system, unlike in a typical search engine, not all queries may need immediate responses. Alert based queries that are known long before we need to generate a response allow us to schedule the processing of a search request. This lets us gain throughput possibly at the cost of response time. We queued many requests, grouped them based on the postings file that contains the results, and further sorted each group based on the offsets in the files, and then processed the reordered requests. A good disk space allocator allocates close pages in a file closely on the disk, so ordering the requests in the way mentioned above leads to better scheduled set of disk reads. This increased the throughput considerably.

3.3 Merging

Updates occur at a much lesser frequency when compared to searching. They are expensive and so are batched and merged with the token index. Merging may take place periodically or whenever the number of updates exceeds a certain predefined value.

The update manager is primarily responsible for ensuring that all the updates are reflected correctly in the token index. The update manager receives update messages from the token index and creates three new data structures. Each update message contains the token, the posting and a flag that specifies whether the action to be performed on the posting. If the flag is set, the posting is to be added to the token index. If the flag bit is not set, the posting is to be deleted from the token index.The lex diff data structure is contained in memory and ensures that duplicates are not added to the lexicon on the disk. The update diff data structure which is stored on the disk specifies all the posting updates corresponding to a particular token. This data structure also serves as a write ahead log as all updates are stored persistent on disk before the actual updates are performed. Upon receiving update messages, the update manager also writes this information to the lexicon that is stored on the disk with a null pointer for the offset field of the lexicon entry to indicate that the postings for this particular token haven't yet been updated.

During the merging phase, the merger uses the data structures created by the update manager. For each token in the lexicon on disk, the merger scans the update diff data structure to see if there are any update entries corresponding to this token. It also retrieves all the postings corresponding to this token from the postings file. If there are no entries, all the postings corresponding to this token are sent to the output manager to be written without any modifications. If there are postings corresponding to this token, the corresponding actions are performed. If the action specifies the addition of the posting, the posting is inserted into the postings stream sent to the output manager. If the action specifies the deletion of the posting, the corresponding posting is removed from the stream of postings sent to the output manager. To ensure efficiency, the update diff data structure and the postings file are kept sorted so that merging can be done by simply comparing the corresponding front entries as in a merge sort. The sorting algorithm is a stable one to ensure that updates are applied in the same order in which they took place. To ensure that the changes are reflected in the lexicon, the merger computes the pointer to the first posting for each token and writes out a lex entry for the token using this pointer.

The output manager is responsible for writing the new lexicon and postings files. Its primary function is to abstract the internal formats of each of these files and to provide a uniform interface for reading and writing to the lexicon and the postings file.

After the merging is done, the new lexicon and the new postings file replace the old ones so that the updates are subsequently reflected.

3.4 Benchmarks

We obtained the search performance numbers by running search queries against a word list and postings files generated by specially built random data generators. A word list generator generates word list of a specified size. Given this word list, a postings file generator generates single or multiple postings file(s) in the format we use for postings files. Parameters that can be specified include number of documents and their distribution with respect to word categories. In order to obtain a distribution similar to that of a real corpus as well as to keep the generation simple and fast, we tuned the generator to follow a bucketwise zipf distribution. That is, we classified all the tokens into a few groups. As we go from the first group to the last one, the size of the group increases exponentially, while the number of documents each word occurs in decreases exponentially.

The number of groups classified into depended on the word list size. For the word list of size ten million, for instance, we divided the words into six groups. The first group containing just a hundred words as a whole occurs in around 99% of the documents. The second group contains nearly ten times as many words, but each with frequency a tenth of that of a word in the first group. The sixth group contained nearly 90% of the words in the word list, but were very rare.

Though a real corpus is the best one to use to obtain a very good distribution of word occurrences, generated data has many advantages over a real corpus. If we use a real corpus, we will have to build the inverted index before testing searching. As the development of the searching and index building modules had to go on in parallel, the generators let us incrementally test the performance of searching even before the index building part was completely developed. This let us come up with optimizations and incorporate them as we developed. Moreover, index building is a slow process. Even if we had the index building modules ready, it would take very long to build an index measuring a few gigabytes. This was unacceptable since we were forced to build the index often because of frequent changes in postings file format to accommodate for addition of new features and optimizations as a part of incremental development process. The generators have advantages when we have to study the effect of word occurrence distribution on performance, as the size and distribution are just parameters to the generators.

4 Implementation

The implementation of the token index was done in both the C++ and the Java programming languages. We'll describe both the implementations in detail in this section.

The linear hash based lexicon depends on minibase for the storage and buffer management layers. This was written purely for evaluation and comparison purposes and is not integrated with the rest of the system. The driver takes tokens from a tokens file, and offsets from an offsets file and loads the linear hash table inserting <token, offset> pairs. Keywords to be searched are randomly chosen from the tokens file and searched for. Searching finds the offset from the linear hash table, seeks to that offset in the postings file, and reads the first few results.

The C++ implementation of the inverted index consists of generators for benchmarking and the actual implementation itself. One of the generators generates tokens, and the other generates postings file and offsets. The lexicon loads from tokens file and the offsets file. It also provides interfaces for inserting a <token, offset> pair, and for searching for a token returning the corresponding offset. A module to search the results provides the interface for the inverted index to the user. This module uses the lexicon to find the offset, seeks into the postings file and fetches the first few results. This also provides an interface to queue requests and process them. These methods can be used in batch processing of queries. The search tester is the driver program which selects keywords to be searched for from the tokens file, loads the lexicon, searches results.

The Java implementation has modules for searching and updating. The modified modules for searching are also separately provided so that they can be run with the data sets generated and used by the C++ implementation. The problem using the same data set for C++ and Java implementations is that on Intel machines the C++ implementation uses little endian format, while Java universally uses big endian format. These modules, provided separately, have classes to read the offsets file and the postings file in little endian format. A lexicon that can handle only ASCII characters is provided. Restricting to ASCII characters has the advantage of requiring much less memory than its wide character counterpart. However, a lexicon (I18nLexicon) which can deal with international characters is also provided. The search tester drives the search tests, and has the functionality very similar to that of the C++ implementation.

The main Java implementation, has a module that is responsible for fetching the postings list for a given token using the lexicon, a module to merge the current postings list with updates, a module to write the output to a new postings file in a predetermined format, and a module to maintain the list of updates to be merged. All these modules use iterators in the form of input and output streams to read and write entities they are concerned with. The token index is the main class with provides the interface to the user.

5 Results

The following results for our token index were obtained on a machine with 512 MB RAM and a Pentium III processor with a clock speed of 1 GHz. The operating system used was the Linux Operating System with kernel version 2.4.18. The version of Java used was 1.3.1.

Searching involves an in-memory lexicon lookup which takes a few μs and a seek and read at the offset found by the lookup. As we fetch only the top results in out experiments, this can be done in one read. So we can expect the search response time to be approximately equal to the time taken for a read. The response time for our test with the large scale data set indeed corresponds to disk read time. However, response times for the medium and small scale tests were much lower than that. This is because, the postings file being small, is largely cached in memory. We also observed during the test runs that a few initial runs showed larger response times finally stabilizing at the number shown. This evidence confirms the explanation that the small response times were due to caching. Further, since the disk is the bottleneck, there is negligible difference between the C++ and the Java version in the large scale test. However, in the medium and small scale tests, the response time is not totally dependent on the disk seek time, so Java version is slower than the C++ version.

Description	Scale		
	Large	Medium	Small
# Distinct Tokens	10,000,000	1,000,000	100,000
# Documents	20,000,000	2,000,000	200,000
Lexicon Size	223 MB	22 MB	2 MB
Postings File	5.25 GB	440 MB	35.3 MB
C++	8.9 ms	1.6 ms	120 μs
Java	8.9 ms	1.9 ms	250 μs

Table 1: Search Response Times

Description	Scale		
	Large	Medium	Small
W/o Scheduling	104/s	625/s	8333/s
With Scheduling	178/s	1176/s	14184/s

Table 2: Effect of Scheduling on Throughput

As explained earlier, sorting based on file offsets can give significant improvement in throughput. We did expect a considerable improvement for the large scale test. For the medium and small scale tests, we did not expect an improvement as large as we saw. The reason is still unclear.

The linear hash based lexicon used the minibase storage and buffer management layers. This scheme used pages to store the lexicon entries. Un-

Description	Scale		
	Large	Medium	Small
In Memory Lexicon	9.6 ms	1.6 ms	120 μs
Linear Hash Lexicon	-	11.5 ms	150 μs

Table 3: Comparison: in memory lexicon vs. linear hash based one

like the in-memory lexicon, the linear hash based lexicon did not have any specialized data structures. This led to considerable space inefficiency. Inefficient space utilization, compounded by various limitations of minibase, was responsible for this scheme not scaling to our large scale test. However, the effects of a generic database style design can be observed in the response time for the medium scale test.

# Documents	100,000	200,000
Postings file	61MB	122MB
1000 docs/cycle	29 mins	78 mins
10000 docs/cycle	9 mins	20 mins

Table 4: Time taken for building index

We built the index from scratch varying the frequency of merging. As the results show, merging on adding 1000 documents can be considerably slower than merging every 10000 documents.

6 Future Directions

Our implementation can handle queries concurrently with updates, but the implementation of handling queries when merging is not yet complete. One of the features our design addressed from its conception was that of writing to a new update file when the update file written is being used for merging. This allows the inverted index to be queried even when merging.

Optimizing the hit lists for space requires the knowledge of the range of LRIDs, and how sparse or dense they are. Using this knowledge, the standard schemes for optimization for space such as storing differences, or other compression techniques can be incorporated. A flexible design makes this easy because changing the read/write methods of postings is all that is required.

Acknowledgments

We wish to thank Prof. Raghu Ramakrishnan for guiding us through the project and Vuk Ercegovac for his time and valuable tips.

References

[1] Moffat,A. and Zobel,J., Compression and Fast Indexing for Multi-Gigabit Text Databases,*Australian Comput. J.*, 26(1):19, February 1994.

[2] Brown,E.W., Callan,J.P. and Croft,W.B., Fast Incremental Indexing for Full-Text Information Retrieval, *Proceedings of the 20th International Conference on Very Large Databases*, Sept.,1994.

[3] Clarke,C.L.A., Cormack,G.V. and Burkowski,F.J., Fast Inverted Indexes with On-Line Update, *Technical Report CS-94-40, University of Waterloo Computer Science Department*, Nov. 1994.

[4] Knight,J.P. and Hamilton,M., A File System Based Inverted Index,*citeseer.nj.nec.com/57570.html*

[5] Brin,S. and Page,L., The Anatomy of a Large-Scale Hypertextual Web Search Engine, *Computer Networks and ISDN Systems, vol. 30, 107-117*, 1998.

[6] Tomasic,A., Garcia-Molina,H. and Shoens,K., Incremental Updates of Inverted Lists for Text Document Retrieval, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, 289-300*, 1994.

[7] Zobel,J., Moffat,A. and Rao,K.R., Inverted Files Versus Signature Files for Text Indexing,*ACM Transactions on Database Systems, Vol. 23, 453-490*, 1998.

[8] Goetz,B., The Lucene search engine: Powerful, flexible, and free, *http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html*.

[9] Cutting,D. and Pedersen,J., Optimizations for Dynamic Inverted Index Maintenance,*Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval, 405-411*, 1990.

[10] Zobel,J., Moffat,A. and Sacks-Davis,R., Searching Large Lexicons for partially Specified Terms using Compressed Inverted Files, *Proceedings of the 19th Conference on Very Large Databases*, 1993.

[11] Moffat,A. and Zobel,J., Self-Indexing Inverted Files for Fast Text Retrieval,*ACM Transactions on Information Systems 14, 4, 349-379*, 1996.

[12] Berry,M.W., Dumais,S.T. and Shippy,A.T., A case study for latent semantic analysis, *citeseer.nj.nec.com/berry95case.html*.

[13] Putz,S., Using a Relational Database for an Inverted Text Index, *Technical Report SSL-91-20, Xerox PARC*, Jan. 1991.

[14] A.R. Schmidt, F. Waas, M.L. Kersten, D. Florescu, I. Manolescu, M.J. Carey, R. Busse, The XML Benchmark Project, *Technical Report INS-R0103, CWI, Amsterdam*, April 2001.