DYNAMIC COLLECTIONS IN INDRI

TREVOR STROHMAN

Abstract. Text search engines have historically been designed for unchanging collections of documents. While this is fine for many applications, a growing number of important applications in news, finance, law and desktop search require indexes that can be efficiently updated.

Previous research into supporting dynamic collections revolves around incremental methods. Incremental systems are optimized for adding large batches of documents to an existing index. These systems do not generally allow for queries to run while an incremental update is taking place.

This work presents recent changes to the Indri search engine to support dynamic collections. Unlike previous incremental systems, Indri does not require large batch sizes to achieve efficient indexing performance. Indri is also designed to be as concurrent as possible, allowing queries to run while documents are added to the system.

1. Introduction

While full-text information retrieval systems have been available for decades, they were not commonly used by the public until the advent of the world wide web. Today, almost everyone is a regular retrieval system user, and full-text search is being used for a wide variety of applications, including the web, news, and now desktop search.

Searching desktop files and e-mails presents a difficult challenge for most information retrieval systems. Users expect that e-mails will be indexed the moment they arrive, and that files will be indexed the moment they are saved to disk. In addition, these desktop users will not tolerate occasional outages in retrieval capability just because an index is being updated. Ubiquity is both a blessing and a curse–users now see the utility of full-text search, but now they will not do without it for even a moment.

However, updating a full-text index is a time-consuming process. Modern information retrieval systems create indexes from documents using inverted lists. An inverted list contains a list of documents that contain a particular word, and perhaps additional information (such as the locations of a particular word in documents). Even a short document could contain 100 unique words; indexing it requires updating 100 inverted lists. If these inverted lists are stored on a disk, that translates into a 100 disk seeks, which take more than 1 second to complete on modern desktop hardware. Longer documents could take much longer than that. During an update, it is not clear how a system might handle new queries other than to make them wait.

We bypass these restrictions in our recent modifications to the Indri retrieval system. Indri is a new retrieval system developed as a part of the Lemur project [SMTC05]. The Indri system supports structured queries using the language modeling framework, as proposed by Metzler, Lavrenko and Croft [MLC04]. It also supports structured document retrieval, as required for question answering applications. Although the first version of Indri did support incremental indexing, modifications in the latest version allow for true real-time indexing of individual documents.

We identified three key requirements for an information retrieval system that handles dynamic document collections. First, such a retrieval system should allow documents to be added and deleted from the collection quickly, and at any time. Here, quickly is defined as how long a desktop user might be willing to wait for indexing to complete, which means nearly instantaneous response time for a single document. Second, the system should allow queries to execute at any time, even if documents are being added to the collection concurrently. Third, in order for the system to be useful in practice, the system should be competitive in indexing and retrieval performance to systems that are not dynamic.

The latest version of Indri has met these objectives, by using the following design principles:

- Memory structures – hold data in memory as long as possible in order to avoid disk seeks
- Lock isolation – the system holds exclusive locks to data for as little time as possible
- Read only structures – the system uses read-only data structures when possible to reduce the need for exclusive locks
- Background I/O – the system interacts with the high latency disk subsystem in background threads so that indexing operations can be low latency
- Multiversion structures – when long running operations must access data that changes, allow multiple versions of data to exist in the system simultaneously in order to reduce the need for exclusive locks

In the rest of the paper, we discuss how Indri used these principles to achieve our objectives.

2. Related Work

The database community has spent decades researching techniques for allowing concurrent access to changing data. Ramakrishnan and Gehrke [RG00] provide a overview of database principles in general, while Gray and Reuter [GR93] offer an in-depth look at transaction processing systems.

While the problems in maintaining concurrent access to database data and document data are similar, there are some important differences. In database systems, users are particularly concerned with atomicity of operations. In the classic example, a bank patron initiates a transfer of d dollars from a savings account to a checking account. If d dollars are first removed from savings, there will be an instant where the patron's total balance appears to be d dollars too small. Instead, if the d dollars are first added to the checking balance, there will be an instant where the patron appears to have d more dollars than she actually has. In both cases, the accounts are in an inconsistent state after the first account is modified, and consistency is restored after the second account is modified. A major component of concurrency research in databases is ensuring that the database always appears to be in a consistent state, even after a system failure.

In this work, we ensure that document insertions and deletions are atomic; no user of the system should see a document that is partially inserted or partially deleted. However, we do not allow multiple documents to be added or deleted in one atomic transaction. This is not a major restriction, since documents rarely depend on each other the way database records do.

However, we still borrow some ideas from the database community. Asynchronous I/O and fine grained locking are employed in most modern database systems. We also use the idea of multiversion concurrency [BG83].

The information retrieval community has not neglected dynamic collections. However, research has focused on 'incremental' systems, which add large batches of documents to an existing index, instead of systems which can efficiently add a single document. Also, this research does not consider the ability to continue query processing while documents are added to an index.

Brown, Callan and Croft [Bro95] investigate a method for efficiently adding documents to an existing index. They make a distinction between small inverted lists (less than 8K) and large inverted lists (greater than 8K). When a small inverted list needs to grow, it is copied into a larger, contiguous region of the inverted file. However, long inverted lists are not moved; instead, a forwarding pointer is added from the old inverted segment to a new segment. This opens the possibility for inverted lists to be linked lists of data throughout the inverted file. When building an index in 7 batches, they find that query performance decreases by about 6%. The cost of using small batch sizes is high; it takes approximately 8 times as long to index a 64MB batch of text versus a 1MB batch of text in their model.

In a more recent work, Lester, Zobel and Williams [LZW04] compare three update strategies for indexes: in-place, re-merge or rebuild. The in-place strategy is similar to that used by Brown, except the linked-list optimization is not employed; all inverted lists are contiguous, and existing data must be copied if not enough free space remains to write additional new data. In the re-merge strategy, the new batch of documents is built into a separate index, then merged with the existing index. The rebuild strategy involves throwing away any existing index structures and building a new index from scratch. The authors find that unless the batch size is small (under 100 documents), re-merging is the most efficient update method. However, the authors have not investigated pre-allocation strategies or strategies for handling large lists differently, as in Brown, Callan and Croft's work.

In the smallest batch size reported (10 documents), Lester, Zobel and Williams report an update time of approximately 7 seconds in the best case (in-place) against a 1GB index. While this is quite fast compared to the other strategies presented, this is not ideal for single document updates.

The work presented here is most similar to the Lucene search engine [Cut]. In a normal configuration, Lucene writes data to disk in segments, much like a traditional batch indexer. However, data can be queried from these segments as soon as they are written–a segment merge step is not necessary. This is somewhat similar to the linked list method of Brown, Callan and Croft. However, there still is the overhead of writing data in batches; to get good performance, many documents must be written to disk together in one batch.

If faster response time is required, Lucene offers a `RAMDirectory` class that allows indexes to be created in memory. Adding documents to a `RAMDirectory` is very fast, since no disk I/O occurs in the process. As soon as a document has been added to a `RAMDirectory`, it can be queried by an `IndexWriter` object. This solves the problem of document batches for small collections. However, for collections that are

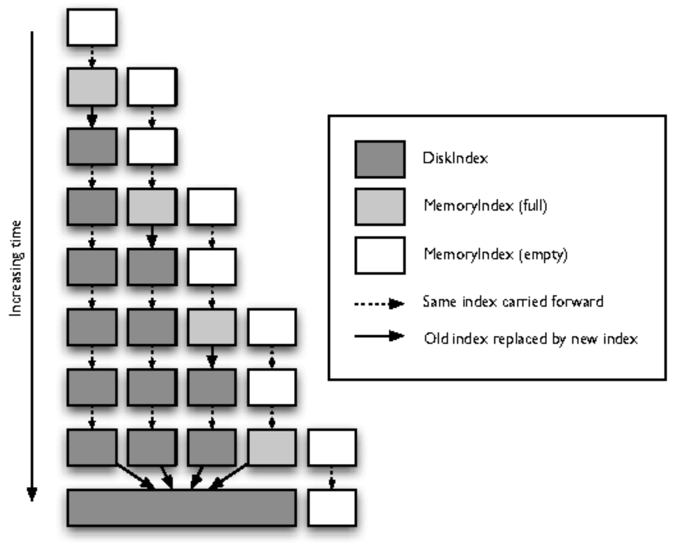

Figure 1. This figure shows a natural progression of `index_states` as documents are added to a `Repository`.

larger than the memory of the machine, in-memory indexing will not be suitable. Data must then be written to disk, and it is up to the user to keep track of where the indexed data is located–some will be on disk, some will be in memory.

In this work, we show how Indri also uses in-memory indexes in order to remove the need for batch indexing while still allowing for immediate access to documents. We also show how Indri automatically decides when data should be written to disk, and when merges should occur, while still allowing query processing to continue.

3. Approach

3.1. Memory Structures. Indri uses two types of indexes; the `MemoryIndex` and the `DiskIndex`. As is probably clear, a `MemoryIndex` resides in RAM, while a `DiskIndex` resides on disk. Both indexes are capable of handling queries, but only a `MemoryIndex` can accept new documents. A consequence of this is that all Indri index structures on disk are immutable; they may be deleted, but they cannot be modified.

Most information retrieval systems build a single index on disk that contains all indexed documents. While Indri can do this, sometimes it may have many different indexes on disk at once. Therefore, we use the term *repository* to refer to a collection of indexes and related structures on disk.

When creating a repository from a set of text documents, Indri adds incoming documents to the active `MemoryIndex`. Whenever a repository is open in write mode, there is an active `MemoryIndex` ready to receive documents. For small collections, indexed data is not written to disk until the repository is closed. However, for larger collections, the `MemoryIndex` will grow larger than the user-configurable memory limit. At this point, the `MemoryIndex` will be written to disk. At this time, a new `MemoryIndex` will be created, and will serve as the active index.

Building an index this way is similar to how many information retrieval systems operate. Documents are generally added to in-memory structures one at a time, and at some point a memory (or document) limit is reached, at which time the memory structures are flushed to disk. In batch systems, these disk and memory structures are not self-contained, and need some kind of post-processing before they can be used in the query system.

It is convenient to have many separate indexes on disk, because it is more efficient to construct many small indexes than it is to construct a single large index. This is true because construction of a large index typically happens by merging the data from many small indexes (or disk structures). In order to add documents as quickly as possible to the system, it is advantageous to simply make many small indexes. [LZW04]

However, it is much faster to query a single large index than many small indexes. The primary reason for this is disk seek overhead; the number of disk seeks needed to evaluate a query is linear in the number of indexes. Therefore, for a query-heavy workload, Indri will try to reduce the number of `DiskIndex`es by merging some of them together.

3.2. **Lock isolation.** In order to meet users' expectations of a responsive system, Indri must run queries and add documents quickly. While the previous version of Indri was already an efficient batch system, it would be easy to add locks to data structures such that queries or document insertions could block for long periods of time. In order to keep fast response time, it is necessary to make sure exclusive locks are held for only very short periods of time.

We reduce exclusive lock ownership times by only allowing the active `MemoryIndex` to change. Therefore, this is the only structure in the system (apart from some caches) that requires an exclusive lock. Since its size is limited by available memory, and since all of its contents are in memory, even complex queries against the active index should complete quickly. Of course, lock ownership times can be decreased by reducing the memory limit.

An exclusive lock is also required to add documents to a `MemoryIndex`. In order to keep ownership times down, we make sure that each document is completely parsed before the lock is acquired. The lock is only held while a single document is added to the index, then it is released so that queries can be processed. Most web and news documents can be indexed in less than 1/100 sec., so queries are not likely to have to wait long for execution.

In both the query and the document insertion case, no disk I/O is done while the exclusive lock is held. This significantly reduces the likelihood that the owning thread will be descheduled while holding the lock.

3.3. **Read-only structures.** In order to achieve only short periods of exclusive locking in a system that must handle large volumes of data, we have made most of the data in the system immutable. This is true primarily of data on disk. If data on disk was not immutable, threads would have to acquire a lock and hold that lock during disk I/O, which would lead to high variance in lock duration.

Since disk structures do not change once they are written, they can be read without locks. As a consequence of this locking strategy, Indri is now able to take advantage of multiprocessor systems. Most of the query code path requires only read locks, meaning that query processing is highly parallel, although it is likely to be limited in parallelism by the disk subsystem. Even the indexing process is somewhat parallel, since parsing and indexing can take place simultaneously.

3.4. **Background I/O.** If I/O cannot happen in the code path of queries or indexing, it must happen asynchronously. Indri handles this by running two threads at all times:

- `RepositoryMaintenanceThread`
- `RepositoryLoadThread`

These threads are associated with a particular `Repository`. If more than one `Repository` is open, each repository will have a pair of these threads.

The `RepositoryLoadThread` has two jobs. First, it maintains load average statistics for both queries and document additions. These load averages are much like a Unix process load average; they indicate the number of queries executed and documents added over the last 1, 5, and 15 minutes. This helps the system make decisions about when to write data to disk.

The second job of the `RepositoryLoadThread` is to check on the memory usage of the system. If the system is using significantly more memory (25% more) than the user-specified limit, the `RepositoryLoadThread` suspends all document indexing until the memory usage falls. This keeps the system from crashing in the case when documents are added to the system in batch. For most conceivable real time applications, like news feeds, the system should never run out of memory.

The `RepositoryMaintenanceThread` writes index data to disk. Since this is the only thread that can write index data to disk or delete data from disk, no complicated locking is necessary in this thread. This thread wakes up approximately 5 times a minute and checks on the current memory usage of the system. If the system is using too much memory (as measured by the user-specified memory limit), the thread will begin writing data to disk.

As discussed before, it is not always advantageous to add another `DiskIndex`, since many small `DiskIndex`es are slower than one large `DiskIndex` at query time. Because of this, the maintenance thread checks the query and document load before writing to disk. If the query load is high with respect to the document load, the maintenance thread will favor merging indexes together instead of writing a new one.

3.5. **Multiversion structures.** Since an Indri `Repository` may contain more than one index, Indri maintains a structure called an `index_state`. This structure maintains pointers to all indexes currently in the `Repository`.

Data is written to disk in two scenarios:

- A `MemoryIndex` has filled past its memory limit
- There are too many `DiskIndex`es, and they need to be merged

In both cases, Indri is writing data to disk that is already available in the system in some other index structure. Therefore, the `index_state` needs to change to reflect that the data has moved.

One way to solve this problem would be to protect the `index_state` structure with a readers/writers lock. However, this approach could lead to poor performance in a heavily loaded system.

Consider the case where Indri is running on a parallel system, and an Indri user is running very complicated queries that each require 10 minutes of processing to execute. Suppose that the user submits these queries on two separate threads, so that there are always two of these queries running. Suppose the execution of these queries is staggered by 5 minutes; queries start on thread A at 0, 10, 20, 30, 40 and 50 minutes past the hour, while queries begin on thread B at 5, 15, 25, 35, 45, and 55 minutes past the hour.

Suppose we attempt to lock the `index_state` structure for writing just before a new hour begins. We would keep thread A from starting its next query while we wait for thread B to complete a query, and one processor would be left idle for 5 minutes. This is clearly undesirable.

To avoid this, Indri may maintain more than one `index_state` structure at a time. All new work (queries, document additions) can use the new `index_state` structure, while old jobs continue to use the old `index_state` structure. When there are no users of the old `index_state` structure left, it is deleted.

In the example, this would mean that thread A would start a new query using the new `index_state` while thread B would finish its query using the old `index_state`. When thread B finishes its query, the old `index_state` would have no users, and would therefore be deleted.

4. Document Deletion

Indri includes marked deletion support. Marked deletion is a weak form of deletion where deleted documents are simply hidden from the user. The data for these documents is not removed from the inverted lists, direct lists, or compressed collection. Also, the counts for the terms in this document remain in the corpus statistics.

Suppose we have a document set A, and a subset $B \subset A$. We create an index I by adding the documents from A to I, then deleteing those documents in B from I. We create a similar index I' by just adding the documents in $A - B$ to I'. Index I will take more disk space than I', because it still contains all the document data from B. Furthermore, the query results will be slightly different when querying these two indexes, since I will have slightly different corpus statistics than I'. For these reasons, the deletion feature should be used with caution when doing research with Indri.

For practical use, though, deletion can be very useful feature. While delete by itself is perhaps not especially useful, it can be used to update documents (delete the old version, then insert the new copy). This can be useful in desktop search or in newsfeed search where article corrections may arrive after the erroneous version of an article has been indexed.

To mark documents as deleted, we use a simple bitmap where a bit is set when a document is deleted. Any bits not in the bitmap are assumed to not be set; therefore, if no documents have been deleted, the bitmap file will be an empty file. The file is only extended until it contains the last non-zero bit.

At query time, each document is checked against the bitmap before being scored. Only documents that have not been deleted can be scored.

5. Conclusion

Indri is now capable of making indexed documents available for query immediately after they are indexed, which typically takes a small fraction of a second. The penalty for fast, concurrent access to newly-indexed documents is low enough that Indri has no special batch or incremental mode. With this system, we can index web data at approximately 15GB/hour, including storing a compressed copy of each document.

With this level of performance, we have achieved the goal of making a retrieval system suitable for news filtering and desktop search applications. We believe this is the first open-source system to have this capability.

6. Acknowledgements

This work was supported in part by the Center for Intelligent Information Retrieval and in part by Advanced Research and Development Activity and NSF grant #CCF-0205575 . Any opinions, findings and conclusions or recommendations expressed in this material are the author(s) and do not necessarily reflect those of the sponsor.

Documents	25,205,179
Total terms	22,811,162,783
Unique terms	51,208,878
Frequent terms	224,840
Index size	90G
CompressedCollection size	123G
Uncompressed text size	431G

Figure 2. Statistics about the GOV2 collection

7. Appendix

In the course of adding dynamic collections support to Indri, a large part of the indexing infrastructure was changed. Some of the major changes to indexing had little to do with dynamic collections support, and therefore did not fit well into this paper. This appendix contains information about the Indri changes not directly related to dynamic collections.

7.1. **Disk Index Structures.** This section gives a broad overview of the files in a typical Indri repository, and what data these files contain. Since Indri was built to handle large collections of text, we have included the file sizes for these file structures in an index of the TREC GOV2 collection (see Figure 2).

The core index structures are stored in a `DiskIndex`, which contains inverted lists, direct lists, document statistics, and field information (Figure 3). The largest files are the direct file and the inverted file, since both grow roughly linearly with the number of words in the collection. The rest of the files contain aggregate statistics about terms and documents.

The `CompressedCollection` stores the full text of each document in the collection, as well as term boundary information from the parser (see Figure 4). This information is most often used to show snippets of text at retrieval time, so the term boundary information is useful for highlighting. Even though each document is compressed individually, the compression ratio here is generally quite good; we see 50% to 80% reduction in size when the text is compressed. The `lookup` file allows convenient random access into the `storage` file in order to find document data. The optional forward lookup files allow particular bits of a document (such as the document title) to be found efficiently without needing to decompress the entire document. The optional reverse lookup files allow documents to be found by some identifying information (like a URL) instead of the Indri-assigned numeric documentID.

The `Repository` directory contains both a `CompressedCollection` directory and a directory of `DiskIndex`es (see Figure 5). It also includes a deleted documents bitmap, described earlier in this paper.

Filename	Size (GOV2)	Description
`directFile`	38G	Numeric representation of each document in the collection, useful for query expansion
`documentLengths`	97M	Length of each document in words, 4 bytes per document
`documentStatistics`	481M	Offset of each document in the directFile, document length in the directFile, document term length, number of unique terms in each document
`field`n	-	Inverted extent list for field number n
`frequentID`	7.7M	`BulkTree` storing the mapping from termID to termString and term statistics for frequent terms
`frequentString`	7.3M	`BulkTree` storing the mapping from termString to termID and term statistics for frequent terms
`frequentTerms`	6.6M	List of $termID, termString$ pairs, not stored in a tree, used at index merge time
`infrequentID`	1.8G	`BulkTree` storing the mapping from termID to termString and term statistics for infrequent terms
`infrequentString`	1.8G	`BulkTree` storing the mapping from termString to termID and term statistics for infrequent terms
`invertedFile`	49G	Inverted lists for each term in the collection
`manifest`	428b	XML file storing important collection statistics and configuration information

FIGURE 3. Files composing a `DiskIndex`

Filename	Size (GOV2)	Description
`lookup`	498M	`Keyfile` (B-Tree) that stores the mapping between documentID and offset into `storage`
`manifest`	70b	XML file storing configuration information for this `CompressedCollection`
`storage`	123G	Stores compressed version of each document in the collection, along with byte offsets for each word in each document, and various document metadata added at index time.
`forwardLookup`n	-	`Keyfile` (B-Tree) that stores the mapping between a documentID and a metadata string
`reverseLookup`n	-	`Keyfile` (B-Tree) that stores the mapping between a metadata string and one or more documentIDs

FIGURE 4. Files composing a `CompressedCollection`

Filename	Size (GOV2)	Description
`index`	Directory	Contiains zero or more `DiskIndex` instances, in numbered subdirectories.
`collection`	Directory	Contains a `CompressedCollection` instance.
`deleted`	0b	Bitmap containing a list of all deleted documents.
`manifest`	102b	XML file storing configuration information about the collection, including index counts, stemmer and stopword information.

FIGURE 5. Files composing a `Repository`

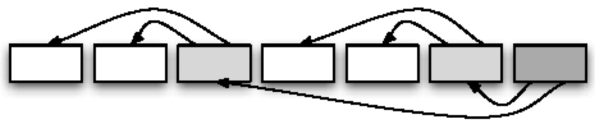

FIGURE 6. Construction of a BulkTree. Notice that leaves are written in order, while internal nodes of the tree are written only when all of their children have been written.

7.2. **Vocabulary Growth.** It has been suggested that vocabulary growth in a collection follows the function kn^b, where k and b are constants, and b is close to 0.5, and n is the number of documents in the collection [BYN99]. This formula suggests a very slow vocabulary growth rate once the collection becomes large. In particular, we expect to find $k(n^b - (n-1)^b)$ unique words in the n^{th} document. For $b < 0$, this quantity becomes arbitrarily small as collection size grows.

This model has shown to be a good fit for smaller collections, and for collections (like news) that are considered to be clean.

However, this model neglects the common use of unique identifiers in documents. As an example, many legal documents are stamped with a unique Bates number which needs to be accessible from the retrieval system. Corpora that use such identifiers will necessarily have $\Omega(n)$ vocabulary growth.

For large collections, linear vocabulary growth can cause extremely large vocabulary structures that become unmaintainable. In the first release of Indri, the vocabulary was stored in a single B-Tree. As the vocabulary grows, this tree becomes so large that it is impossible to cache it in memory efficiently, causing disk seeks that slow down the system.

In the latest version of Indri we recognize two classes of terms. The frequent class of terms includes those words that are broadly used in text, and are likely to be seen in topical queries. The infrequent class of terms includes misspellings and machine generated unique identifiers; these terms are necessary to store, but are unlikely to be used in queries except for very specific information needs.

Indri considers terms that appear in more than one thousand documents to be frequent terms. While this may not be the perfect constant to use for small collections (where very few terms will then be marked frequent), this is not important from a performance standpoint because we expect the total vocabulary size to be easily manageable with current hardware.

As an example, for the GOV2 collection (see Figure 2), we find that there are 224,840 frequent terms, but 51 million infrequent terms. The B-Tree that holds the frequent terms is 7.5MB in size and therefore is easily cacheable. By contrast, the infrequent term B-Tree is 1.8GB, which is larger than the installed memory on most systems at this time.

7.3. **Merging.** As Brown *et al.* [BCC94] showed, it is necessary for inverted lists to be stored contiguously on disk for optimum retrieval performance. Because of this, Indri includes the ability to merge many indexes into a single larger index to increase performance.

During the index building process, Indri creates a series of indexes on disk. Each index is self-describing; it has its own inverted lists, direct lists, field lists and vocabulary. During query processing, the query code runs each query against each index in turn, and then merges the results. This is not the most efficient way to process queries–queries would run faster if all the document data was in a single index. In order to process queries efficiently, Indri needs a way to combine many small indexes into one large one. This is what the index merge process does.

Merging would be a straightforward process if all the data could fit in RAM; unfortunately, for large collections, the memory cannot hold all of the data necessary to merge. The complexity of the merge code is due to these memory constraints.

The inverted lists in each index are stored in alphabetical order to make merging simpler. The lists are merged much like the last phase of a merge sort into one long inverted file. In the process, term statistics for each term are collected and stored in B-Tree structures known as `BulkTree`s. A `BulkTree` is simply a B-Tree that is loaded in bulk. All items inserted into a bulk tree must be in sorted order first in order for the bulk loading process to work. Since the keys are inserted in sorted order, the tree can be built from the bottom up, with leaves written first, followed by nodes at higher levels. This process is then approximately linear, instead of the $O(n \log n)$ required for n B-Tree insertions. More importantly, this build process requires no disk seeks and no disk reads, which can dominate B-Tree build time for large trees.

Every term in an index is assigned a number, known as a termID, which is an efficient representation of that word. The termID is used in the direct lists to compactly represent terms. Since our compression scheme for direct lists favors small numbers, it is important that the most frequent terms are assigned the smallest numbers. So, frequent terms are assigned termIDs in descending order by term frequency. The infrequent terms are then assigned termIDs in alphabetical order. We assume that the infrequent terms appear infrequently enough that the size of their termIDs does not matter much, and there is not enough memory available in the merge process to assign them by frequency anyway.

The most difficult part of the merge process is reassigning these termIDs in the direct lists. We want to be able to handle collections with over 100 million terms; using a lookup table to remap the termIDs on a collection this large would require 400MB of contiguous memory, and we would need such a table for each index being merged. We considered this an unreasonable amount of memory usage. Therefore, we use a combination of structures to perform term mapping.

7.3.1. *Array Mapping.* For words that were frequent, we use an array to map their old values to new termID values. Since the frequent terms are assigned small termID values, this is tractable in practice. The GOV2 collection (426GB of web data) contains 224,840 frequent terms, meaning this frequent term array is never larger than 1MB for collections we currently use.

7.3.2. *Hash Table Mapping.* For words that were infrequent, but are becoming frequent, a hash table is used to map from the old termID to the new termID. An array cannot be used here reliably, since the total vocabulary size of the previous index may have been in the hundreds of millions of terms.

7.3.3. *Bitmaps.* If the old termID was not found in the frequent term array or in the hash table, a bitmap is used instead. The bitmap is not particularly time efficient, in that it requires $O(\log n)$ time to resolve a mapping. However, it is very space efficient. It requires approximately 1.5 bits per vocabulary term. For a 100 million term vocabulary, this results in a bitmap of about 20MB. Here we exploit

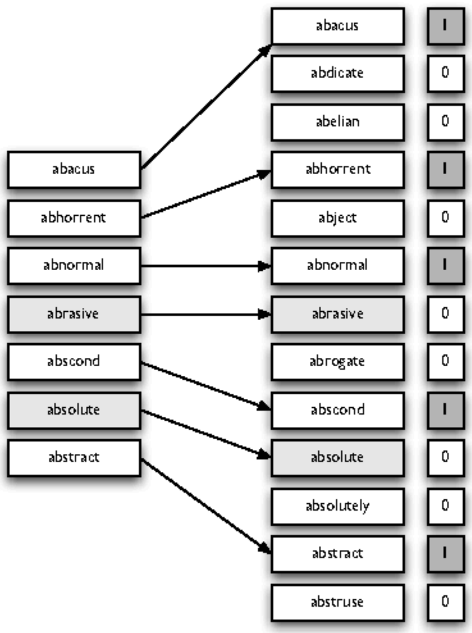

Figure 7. An example of a bitmap that maps terms from an old index (left) into a new index (right). The dark terms are frequent, and are therefore not included in the bitmap.

the fact that these are infrequent terms, so a lookup in this data structure is not the common case.

Suppose we are merging indexes A, B and C to create a new index D. Let $IV(A)$ be the infrequent vocabulary of index A, in alphabetical order; similarly let $V(D)$ be the vocabulary of D in alphabetical order. Clearly $IV(A) \subset V(D)$, since D contains every word in A. Since $IV(A)$ and $V(D)$ are ordered sets, it follows that $\forall\, x \in IV(A)$, the position of x in $V(D)$ is greater than or equal to the position of x in $IV(A)$.

We can store this kind of mapping in a bitmap, where each bit position in the bitmap represents an element in $V(D)$, and each set bit in the bitmap represents a term that is also in $IV(A)$.

As an example, consider the following small bitmap:

$$0001010010$$

This bitmap maps a three term vocabulary from $IV(A)$ to a ten term vocabulary $V(D)$. This bitmap contains three mappings: $\{(1,4),(2,6),(3,9)\}$. The mapping $(1,4)$ is represented by the first bit set, which happens to be at position 4.

Certainly this bitmap can represent the mapping between $IV(A)$ and $V(D)$ using $O(|V(D)|)$ bits, but the search time would also be $O(|V(D)|)$, since it would be necessary to scan all the bits to see which ones are set. To make searching faster, we split the bitmap into pieces, where each piece is 32 bytes long. The first 8 bytes of each piece contain two 4-byte integers, f and t. The remaining 24 bytes consitutes bitmap space, where this bitmap is considered to be relative to f and t.

As an example, using the bitmap shown above, the first bit would correspond to a mapping of $(1 + f, 4 + t)$.

While this header adds 50% more space to the bitmap, it can now be searched in $O(\log n)$ time. The f values are searched first using a binary search, followed by a constant time linear scan of the appropriate 24 byte space.

7.3.4. *Term Translator.* The array, hash table and bitmap compose a complete mapping from the old termID space to the new termID space. These three data structures are wrapped together in one object called a `TermTranslator`.

References

[BCC94] Eric W. Brown, James P. Callan, and W. Bruce Croft, *Fast incremental indexing for full-text information retrieval*, VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1994, pp. 192–202.

[BG83] Philip A. Bernstein and Nathan Goodman, *Multiversion concurrency controltheory and algorithms*, ACM Trans. Database Syst. **8** (1983), no. 4, 465–483.

[Bro95] Eric W. Brown, *Fast evaluation of structured queries for information retrieval*, SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval (New York, NY, USA), ACM Press, 1995, pp. 30–38.

[BYN99] R. Baeza-Yates and G. Navarro, *Modern information retrieval*, Addison-Wesley, 1999.

[Cut] Doug Cutting, *Lucene*, http://jakarta.apache.org/lucene.

[GR93] Jim Gray and Andreas Reuter, *Transaction processing: Concepts and techniques*, Morgan Kaufmann, 1993.

[LZW04] Nicholas Lester, Justin Zobel, and Hugh E. Williams, *In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems*, Proceedings of the 27th conference on Australasian computer science, Australian Computer Society, Inc., 2004, pp. 15–23.

[MLC04] Donald Metzler, Victor Lavrenko, and W. Bruce Croft, *Formal multiple-bernoulli models for language modeling*, Proceedings of ACM SIGIR 2004, 2004, pp. 540–541.

[RG00] Raghu Ramakrishnan and Johannes Gehrke, *Database management systems*, McGraw-Hill Higher Education, 2000.

[SMTC05] Trevor Strohman, Donald Metzler, Howard Turtle, and W. Bruce Croft, *Indri: A language model-based serach engine for complex queries*, IA 2005: Proceedings of the 2nd International Conference on Intelligence Analysis (to appear), 2005.