Efficient Search in Large Textual Collections with Redundancy

Jiangong Zhang and Torsten Suel
CIS Department
Polytechnic University
Brooklyn, NY 11201, USA
zjg@cis.poly.edu, suel@poly.edu

ABSTRACT

Current web search engines focus on searching only the most recent snapshot of the web. In some cases, however, it would be desirable to search over collections that include many different crawls and versions of each page. One important example of such a collection is the Internet Archive, though there are many others. Since the data size of such an archive is multiple times that of a single snapshot, this presents us with significant performance challenges. Current engines use various techniques for index compression and optimized query execution, but these techniques do not exploit the significant similarities between different versions of a page, or between different pages.

In this paper, we propose a general framework for indexing and query processing of archival collections and, more generally, any collections with a sufficient amount of redundancy. Our approach results in significant reductions in index size and query processing costs on such collections, and it is orthogonal to and can be combined with the existing techniques. It also supports highly efficient updates, both locally and over a network. Within this framework, we describe and evaluate different implementations that trade off index size versus CPU cost and other factors, and discuss applications ranging from archival web search to local search of web sites, email archives, or file systems. We present experimental results based on search engine query log and a large collection consisting of multiple crawls.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms

Algorithms, Design, Experimentation.

Keywords

Search engines, inverted index, redundancy elimination, index compression, query execution.

1. INTRODUCTION

With the rapid growth of the web, more and more people use web search engines as their primary means for locating relevant information. Such search engines are built on large clusters of hundreds or thousands of servers, and employ numerous published and proprietary performance optimizations in order to support thousands of queries per second on billions of web pages. Current search engines focus on providing access to the most recent snapshot of the evolving web. In order to optimize the freshness of their index, search engine crawlers continuously retrieve new web pages, add them to their document set, and then either periodically or continuously update their inverted index to reflect the new data. If the URL of a page has been previously crawled, then the engine typically replaces the old with the newer version, and any information contained only in the older version becomes unavailable.

While most users are primarily interested in current pages, there are many cases where a search over all previous versions would also be of interest. As one important example, the Internet Archive has collected more than 85 billion web pages over the last decade in an attempt to archive and preserve the information on the web. It is currently not feasible for the archive to offer full-text search on the entire data for a large user population, due to the high cost of processing a query on such a data set. One reason is that current indexing and query processing techniques, when applied to say ten successive crawls of the same set of URLs, result in index sizes and query processing costs roughly ten times that of a single crawl. Of course, completely identical versions of a page could be simply removed from the index. However, in many cases the new version of a page is different from, but still substantially similar to, the previous one, and it would be desirable to be able to exploit this redundancy to decrease index size and increase query throughput.

A large amount of recent research by networking, OS, and data compression researchers has focused on the problem of efficiently storing and transmitting redundant data collections. This includes work on differential compression techniques, e.g., for storage and versioning file systems [22, 13, 26, 17, 10], file synchronization techniques such as *rsync* [37], and redundancy-eliminating communication protocols for web access or other tasks [19, 25, 27, 34]. Thus, we know how to store and transmit the redundant document collections themselves in a highly efficient manner. However, almost all current techniques for dealing with inverted indexes on such collections, including current index compression techniques, are unable to exploit redundancy across version boundaries (one exception is the very recent work in [7, 16] which we discuss later).

In this paper, we focus on this problem of how to efficiently index and search collections with significant redundancy. We propose a complete framework for indexing, index maintenance, and query processing on such collections. Our main focus is on archival collections, such as the Internet Archive, that contain several similar versions of the same document. In addition, there are several other scenarios that can be addressed with our approach. General broad-

WWW 2007, May 8–12, 2007, Banff, Alberta, Canada.
ACM 978-1-59593-654-7/07/0005.

based web crawls as well as site-specific collections also tend to have a significant amount of redundancy, particularly within large sites [12]. The same is true for email archives where replies often quote from previous messages in the thread. In these cases, our framework also reduces the index size, though by a more moderate factor. It can also support highly efficient updates in the case where old versions are replaced by new versions, and in the case where a remote index is updated across a network. Finally, there are interesting possibilities for applications to desktop search and indexing of versioning file systems that retain old versions of all files.

The basic ideas underlying our approach are almost embarrassingly simple, and we now describe them very briefly. First, we use content-dependent string partitioning techniques, e.g., *winnowing* [30] or Karp-Rabin partitioning [20], to split each document into a number of *fragments*, say, 10 to 25 on average. The main characteristic of these techniques is that similar files will have many fragments in common; this has been previously exploited by OS and networking researchers to save storage and transmission costs on redundant data sets [13, 19, 22, 25, 27]. We then simply index these fragments instead of the complete documents, i.e., each distinct fragment is assigned a fragment ID and the index contains references to fragments rather than documents. The result is a much smaller index for collections with significant redundancy. We then design modified algorithms for *document-at-a-time* query processing that efficiently stitch the fragments back together during evaluation of a query. Finally, by also identifying each fragment by a hash of its content, we can support extremely efficient updates on such indexes, both locally and across a network.

While the basic ideas are simple, there are various details that complicate matters. In particular, we consider several different *fragment sharing policies* that trade off index size versus the CPU cost of query processing and the size of certain auxiliary data structures: In some cases, we may limit elimination of identical fragments to different versions of a page or pages within a site, while in other cases we may want to detect duplicate fragments across the entire collection. While some of the ingredients in our approach have been previously employed in several scenarios, we believe that our general framework still makes a significant and novel contribution. As mentioned, redundancy elimination through partitioning of data into blocks has been used to reduce transmission costs and storage sizes for redundant data sets; see, e.g., [13, 19, 22, 25, 27, 37]. The most relevant previous results on textual indexing are the *Landmarks* approach [23], which focuses on efficient index updates when a new version of a document replaces an older one, and the very recent work in [7, 16] on searching redundant collections. The *Landmarks* approach in particular strongly influenced our approach.

The remainder of this paper is organized as follows. In the next section, we provide some necessary background. Section 3 discusses related work in more detail, and Section 4 summarizes the contributions of this paper. Our general framework is introduced and discussed in detail in Section 5. Section 6 presents a preliminary experimental evaluation of our framework using real query logs and web data. Finally, Section 7 provides some concluding remarks.

2. TECHNICAL BACKGROUND

We now provide some basic technical background on text indexing, search engine query processing, and redundancy detection through partitioning. In the following definitions, we assume that we have a document collection $D = \{d_0, d_1, \ldots d_{n-1}\}$ of n web pages that are stored on disk. Each document is uniquely identified by a document ID (docID); in the simplest case the documents are just numbered from 0 to $n-1$. For now we do not distinguish between different documents and different versions of the same document. Let $W = \{w_0, w_1, \ldots, w_{m-1}\}$ be all distinct words that occur anywhere in the collection. Typically, almost any text string that appears between separating symbols such as spaces, commas, etc., is treated as a valid word (or *term*) for indexing purposes.

Indexes: An *inverted index* I for the collection consists of a set of inverted lists $I_{w_0}, I_{w_1}, \ldots, I_{w_{m-1}}$ where list I_w contains a *posting* for each document containing w. Each posting contains the ID of the document where the word occurs (docID), the number of occurrences in this document (frequency), the (byte- or word-based) positions of the occurrences within the document (positions), plus possibly other information about the context of each occurrence (e.g., in title, in bold font, in anchor text). In this paper, we assume that each posting is of the form $(did, f, p_0, \ldots, p_{f-1})$. The postings in each inverted list are usually sorted by docID, and stored on disk in highly compressed form. Thus, Boolean queries can be implemented as unions and intersections of these lists, while phrase searches (e.g., "New York") can be answered by looking at the positions of the two words. We refer to [38] for more details.

Ranked Queries: We define a query $q = \{t_0, t_1, \ldots, t_{d-1}\}$ as just a set of terms (words). For simplicity, we ignore issues such as term order in the query, phrase searches, or various other options, though our techniques can be adapted to all of these. The most basic way to perform ranking of results in search engines is based on comparing the words (terms) contained in the documents and in the query. More precisely, documents are modeled as unordered bags of words, and a ranking function assigns a score to each document with respect to the current query, based on the frequency of each query word in the page and in the overall collection, the length of the document, and maybe the context of the occurrence (e.g., higher score if term in title or bold face). Formally, a *ranking function* is a function F that, given a query $q = \{t_0, t_1, \ldots t_{k-1}\}$, assigns to each document d a score $F(d, q)$. The system then returns the, say, 10 documents with the highest score. Commonly studied classes of ranking functions include the *Cosine* and *Okapi* measures, but current search engines use many other factors in addition to simple term-based ranking. Our techniques are largely orthogonal to these issues.

The most important observation for our purposes here is that a ranked query can be processed by traversing the inverted lists for the query terms, and computing the score for each document encountered in the lists, based on the information stored in the postings plus usually some additional statistics stored separately. For various reasons, many web search engines prefer to return documents that contain all (or almost all [6]) of the query terms; in this case, it suffices to only compute the score of any document whose docID is in the intersection of the relevant inverted lists.

Document-at-a-Time Query Processing: The cost of query processing is usually dominated by that of traversing the inverted lists. For large collections, these lists become very long. Given several billion pages, a typical query involves hundreds of MB to several GB of index data which, unless cached in memory, has to be fetched from disk. Thus, list traversal and intersection computation has to be highly optimized, and should not require holding the complete lists even temporarily in main memory. This is usually done using an approach called *document-at-a-time (DAAT) query processing*, where we simultaneously traverse the relevant lists from start to end and compute the scores of the relevant documents en passant [6, 21].

We later need to adapt this approach to our new index structure, and thus we now provide some more discussion; additional details are available in the appendix. In the DAAT approach, we main-

tain one pointer into each of the inverted lists involved in the query, and move these pointers forward in a synchronized manner to identify postings with matching docIDs in the different lists. At any point in time, only one posting from each list is considered and must be available in uncompressed form in main memory. Another advantage of the approach is that it allows us to implement optimizations that skip over many elements when moving forward in the lists [24], while hiding all details of the index structure and compression method.

Content-Depending File Partitioning Using Winnowing: A significant amount of research in the networking, OS, and data compression communities has focused on eliminating redundancies in large data sets by partitioning each file into a number of blocks and then removing any blocks that have previously occurred. This is usually done by identifying each block by a hash of its content; if we choose the blocks to be large enough, we can limit the number of hashes such that they can be kept in main memory for many scenarios. One problem is how to perform the partitioning. If we simply partition each file into blocks of fixed size and store their hashes, then we would be unable to detect many repeated blocks due to alignment issues. (E.g., if one file differs from another only by a deleted or inserted character at the beginning, none of the blocks would likely match.) In some cases, this can be resolved by checking for all possible alignments between current and previously seen blocks [37, 34, 32], but in other scenarios this is infeasible [22, 25, 13, 27, 19].

For such cases, several techniques have been proposed that partition a file in a content-dependent manner, such that two similar files are likely to contain a large number of common blocks [20, 30, 29, 35]. Among these, we focus on the more recent *winnowing* technique proposed in [30], which appears to perform well in practice. Given a file $f[0 \ldots n-1]$, the process runs in two phases:

(1) First, we choose a hash function H that maps substrings of some fixed small size b to integer values, say for b around 10 to 20. We then hash each of the $n-b+1$ substrings $s_i = f[i \ldots i+b-1]$ in f, resulting in an array of integer values $h[0 \ldots n-b]$ with $h[i] = H(s_i)$.

(2) We now choose a larger window size w, say $w = 100$ or more, and slide this window over the array $h[0 \ldots n-b]$, one position at a time. For every position of the window, we now use the following rules to partition the original file f:

   (a) Suppose $h[i]$ is strictly smaller than all other values $h[j]$ in the current window of size w. Then cut f between $f[i-1]$ and $f[i]$.

   (b) Suppose there are several positions i in the current window with the same minimum value $h[i]$. If we have previously cut directly before one of these positions, then no cut is applied in this step. Otherwise, cut before the rightmost such position.

It is shown in [30] that if two files have a common substring of size at least $w+b+1$, then they are guaranteed to have at least one common block. The maximum size of a block is w, while the expected size, assuming a random hash H, is $(w+1)/2$. The partitioning can be performed highly efficiently by using a *rolling* hash function H, i.e., a function such that $H(s_{i+1})$ can be computed directly from $H(s_i)$ and $f[i+b]$. The entire process is illustrated in Figure 2.1.

Index Updates: Finally, we need some background on efficient schemes for updating inverted indexes. Consider a new page that has been crawled and needs to be added to the index. The primary performance challenge here is that the typical page has several hundred distinct words, and thus a naive scheme for disk-resident indexes would require several hundred changes in different locations on disk. A number of optimizations for index updates have been proposed [9, 36, 11]. If a very high rate of updates needs to be supported, then the best approach appears to be based on periodically pushing updates to the index. That is, when new documents arrive in the system, we first add them to a second, memory-based index structure. To keep this structure from growing beyond memory size, we periodically merge it (or parts of it) into the main index on disk. If the main index is very large, it may be preferable to use a multi-level scheme where the data in memory is first merged into a slightly larger index structure on disk, which itself is periodically merged into an even larger structure, etc. To process a query, we perform separate accesses into each of the index structures (including the one in memory) and then merge the results. This approach has provable performance bounds and is used in a number of systems (e.g., Lucene and the text index of MS SQL Server).

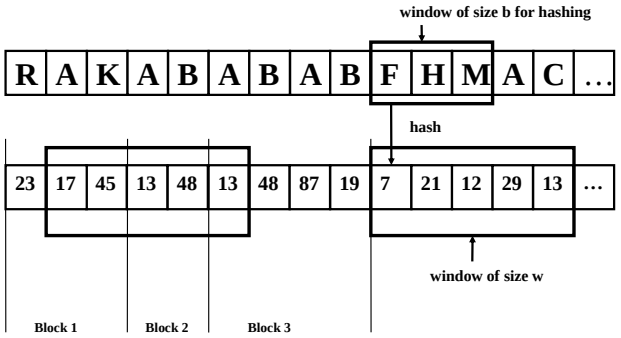

Figure 2.1: Example of the winnowing approach on a file. A small window of size $b = 3$ moves over the file to create a sequence of hashes. A larger window of size $w = 5$ moves over the hashes to determine block boundaries.

Many details depend on the types of updates we need to support, e.g., addition of new documents, deletions, replacement of old versions by newer versions, or addition of newer versions. Note that if we do not store positions in the postings, then a new version of a page that differs very slightly from the previous one may only require updates to a few postings. If positions are stored as well, then a single word added to the beginning of the page may result in updates to all postings. This challenge was addressed by the *Landmarks* approach in [23], which we discuss in more detail later. In general, when a document is added, deleted, or modified, this results in a sequence of insert, delete, and update commands on individual postings that are first buffered in memory and then periodically applied to the larger disk-based structures. The performance of such an update scheme is determined by the number of such commands that are generated, as this determines the frequency and amortized cost of the periodic merges into the disk-based structures. In our later experiments, we will use this as our measure of efficiency.

3. DISCUSSION OF RELATED WORK

We now provide some pointers to related work and discuss the most closely related previous work in more detail. For basics of search engine architecture we refer to [5, 28, 14]. For background on indexing, ranking, and query execution in IR and web search engines, see [2, 3, 38, 39]. Document-at-a-time query processing is described and evaluated, e.g., in [21].

Inverted Index Compression: There are a large number of techniques for inverted index compression; see [38, 39] for an overview. One simple and popular scheme called *var-byte*, evaluated in [31],

encodes each posting, frequency, or position in a variable number of bytes. This allows much faster decompression than many bit-aligned schemes such as Golomb, Rice, Gamma, or Delta coding that in turn achieve a smaller compressed size. However, very recent bit-aligned schemes in [1, 15] manage to outperform var-byte in terms of both compressed size and decompression speed. In our experiments, we will use var-byte as well as the recent Simple-9 scheme from [1]. Our approach here is orthogonal to the choice of compression, and can be used in combination with any of these techniques.

We note that improvements in compression rate can also be obtained by assigning docIDs to pages based on similarity and then applying appropriate local coding models [4, 33]. However, these techniques give only fairly moderate improvements, and do no effectively exploit redundancies when pages are very similar. The issue of assigning appropriate docIDs (or in our case, docIDs and fragment IDs) will also come up in our approach.

File Partitioning and Redundancy Elimination: A number of networking, OS, and data compression researchers have studied the problem of eliminating repeatedly occurring blocks of data from large data sets; see, e.g., [37, 27, 34, 32, 22, 25, 13, 19]. In some cases [37, 32, 18, 26], it is possible to use blocks of fixed size, but many other scenarios require the use of content-dependent partitioning techniques such as [20, 30, 29, 35]. We use the *winnowing* technique in [30], which according to our experiments performs well in practice in terms of the trade-off between the amount of redundancy detected and the number of blocks that are created. Our initial goal in this work was to study bandwidth-efficient index updates over a network, and we were particularly influenced in this direction by *rsync* [37] and the *Low-Bandwidth File System* [25].

We also note that a lot of work in the web community has focused on detecting plagiarism, near-duplicate pages, and phrase-level duplication between different documents; see, e.g., [12, 8]. In fact, the winnowing technique was initially proposed for such purposes rather than for eliminating redundancy for performance reasons. (Our description of this technique in the previous section was adapted to this new scenario.)

Index Updates: A number of researchers have studied the problem of efficiently updating inverted index structures [9, 36, 11, 23]. As mentioned, current state-of-the-art methods generate posting updates that are only periodically merged into the disk-based structures, rather than directly applying changes on disk.

Most relevant to our work here is the *Landmarks* approach in [23], which focuses on the case where an old version of a web page is replaced with a updated, but often very similar, version. If positions are stored in each posting, then a naive approach would have to update almost all postings when a new version arrives, due to changes in alignment. The approach in [23] avoids this by expressing positions relative to certain *Landmarks* in the page, rather than as absolute offsets from the start of the page. When an update occurs, posting updates are only generated for those areas of the page that have actually changed, and the position information for the *Landmarks* is updated to account for changes in the offsets from the start of the page. We note that the *Landmarks* approach can be seen as an implicit partitioning of a page into blocks or fragments, one for each Landmark. The work in [23] looked at several heuristics for selecting the *Landmarks*, but did not consider the above partitioning techniques. One main insight in our work came when we realized the relationship between file synchronization techniques such as *rsync* and the index update approach in *Landmarks*.

However, there are also a number of differences to our work. The work in [23] focuses on the scenario where old versions of pages are replaced by newer ones, and does not consider the case of archival collections and collections with redundancies between different URLs, which is the main focus of our work. We use content-dependent partitionings and identify each resulting fragment by a hash. Our approach also supports efficient updates over a network, and does not require access to the complete outdated version during updates. In summary, we believe our framework provides an elegant generalization and significant extension to the work in [23].

Indexing of Redundant Content: Very recent work in [7, 16], published while our work was in progress, describes alternative approaches for indexing text collections with redundancy. The basic idea is also to avoid repeated indexing of content that is shared between different pages or different versions of a page. However, the approaches taken are somewhat different from ours. In [7], similar documents are organized in a tree structure where each node is a document with some private and some shared content, and each node inherits its ancestors' shared content. In [16], common parts of different versions of a document are identified by solving a multiple sequence alignment problem. The approaches in [7, 16] also use adaptations of standard DAAT query processing. They do not consider incremental updates, but require that the collection is fully available during indexing. There are interesting opportunities for future research that combine ideas from these approaches with our own.

4. CONTRIBUTIONS OF THIS PAPER

We study the problem of indexing and searching large web page collections with redundancy, and propose a new and general framework that results in significant savings in the size of the inverted index and the performance of query processing. In particular, our contributions are:

(1) We propose the use of content-dependent partitioning techniques, in particular the *winnowing* approach in [30], to avoid repeated indexing of content that is shared between several pages or several versions of a page. This is done by partitioning each page into a number of fragments and then independently indexing each fragment using standard techniques.

(2) We propose modifications of document-at-a-time query processing algorithms that can efficiently utilize such fragment-based indexes. We consider several sharing policies between different pages, and show how to adapt query processing for these policies.

(3) We discuss several application scenarios for our framework, and provide efficient update mechanisms for local indexes and index updates over a network.

(4) We perform a preliminary experimental evaluation of our framework based on search engine query traces and more than 6 million web pages extracted from several crawls. Our results show benefits in index size, query processing speed, and update cost.

5. OUR FRAMEWORK

We now describe our new framework in detail. We first describe the various data structures in our setup and the basic steps during indexing and index updates.

5.1 Basic Setup

In the following, we use the term page to refer to distinct documents or information items identified, e.g., by a URL. We use the term *version* to refer to different versions of the same page. Thus, if a crawler visits the same one million URLs ten times, we have one million pages and ten million versions. We use the term *fragment* to

refer to a block of data produced by applying a content-dependent partitioning technique to a page. Pages are identified by a docID, while fragments are identified by a fragID.

Basic Indexing Process: To index a new version of a page, we first partition it into a number of fragments. However, before running winnowing, we first remove all HTML tags from the page and then hash each word in the resulting text page to an integer value; in fact, it suffices to hash to an unsigned char. We then run winnowing on the resulting unsigned char array, where each character corresponds to a single word – this guarantees that partitioning aligns with word boundaries. By using values of w between 100 and 200, we obtain fragments containing 50 to 100 words on average; thus the typical web page is divided into about 10 to 20 fragments.

We then compute an MD5 hash over the content of each fragment, and check a global table to see if a fragment with this hash has been previously indexed. If yes, then we do not index this fragment. Otherwise, we assign a unique fragment ID (fragID) to this fragment, and add for each term in the fragment a posting of the form $(fragID, f, p_0, \ldots, p_{f-1})$, where f is the number of occurrences of the term in this fragment and the p_i are the offsets of the occurrences from the start of the fragment. We will later discuss how to best generate fragIDs, as this impacts query processing and index size. For now, we observe that these postings can be treated by the index just as normal postings, and that the average gap between consecutive fragIDs in the inverted lists increases while the average values of f and the p_i decrease, relative to the docIDs, frequencies, and positions in a standard index.

Finally, for each page, we maintain a data structure that keeps track of the different versions of the pages and which fragments each version consists of. This structure will be stored in compressed form, and is updated whenever we add a new page or version to the index (even if a page consists completely of already existing fragments).

Data Structures: Search engines typically contain three major data structures that are needed for query processing:

- the inverted index, consisting of inverted lists sorted by docID and mapped into one or several large files,
- a dictionary structure which stores for each distinct term in the collection a pointer to the start of the inverted list for this term, plus useful statistical information such as the number of documents containing the term, and
- a page table which stores for each docID the length of the corresponding page in words, other useful information such as the Pagerank value of the page, and possibly the complete URL of the page.

The structures are illustrated in the top half of Figure 5.1. To execute a query, we first use the dictionary to find the start of the inverted lists of the query terms, then compute the top-10 results by traversing the lists and computing scores using the posting information and the additional statistics kept in the dictionary and page table, and finally we use the information in the page table to fetch the actual pages for the top-10 results in order to return meaningful text snippets with the results.

In our framework, we add several new data structures, shown in the bottom half of Figure 5.1, as follows:

- a *hash table* which stores a hash value (e.g., 64-bit of MD5) of the content of each distinct fragment that has occurred in the collection, plus the corresponding fragID (or several fragIDs in some scenarios).
- a *doc/version table* that stores information about a page and its various versions, in particular how many versions there are, which version contains which fragment, and how large each fragment is.
- a *fragment reuse table* that stores in which *other* pages a fragment occurs. For efficiency reasons, it is useful to distinguish between *local sharing*, where a fragment occurs repeatedly in different versions of one page, and *global sharing*, where fragments are reused in versions of other pages. Each fragment has a *primary page*, which is the first page where it occurs. Entries in this table are only created when an already existing fragment is reused in another page.

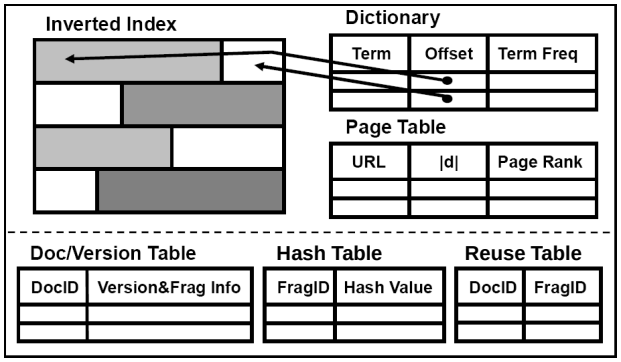

Figure 5.1: Major data structures in our index organization: standard (top) and additional structures in a fragment-based index (bottom).

We note that in some scenarios, we can remove or merge some of the structures, as we discuss later. For example, the hash table is only needed during updates. In terms of memory resources, the largest new structure is typically the hash map which (for large data sets and with proper prefix compression) can be stored in about 9 bytes for each pair of a 64-bit hash and a 32-bit fragID. (We chose 64 bits as this results in a fairly low probability of any hash collisions even for moderately large data sets. Note that if a collision does occur, the result is equivalent to that of a slightly incorrect parsing of the new page with the colliding fragment.)

5.2 Algorithms for Local and Global Sharing

We now consider query processing in our new index organization. In particular, we consider two different sharing policies:

- **Global Sharing:** In this policy, we allow unrestricted redundancy elimination across pages. Thus, if a fragment has previously occurred in any other page, it is not indexed again.
- **Local Sharing:** In this policy, we only avoid re-indexing of a fragment if it has previously occurred in a version of the same page.

Note that there are other interesting choices in between these two, such as allowing fragment sharing only among pages on the same site. Since such pages are more likely to be similar than random pages from other sites, this might achieve most of the benefits of global sharing without some of the costs. We focus here on the two above extreme cases.

Concerning the assignment of fragIDs to fragments, it will be desirable to number fragments in a way such that fragments occurring in the same page are close to each other in the inverted lists. This is easy to do if we have no updates, while a little bit more work is needed in the update case. However, even with updates this is possible if suitable adjustments to the numbering scheme are performed as part of the periodic update operations to the disk-based index that are performed by efficient update schemes. It it sometimes useful to assume that a fragID is actually a pair consisting of

the docID of its primary page and a fragment number; i.e., $(89, 3)$ means that a fragment occurred first in page 89 and was the fourth fragment that occurred first in that page. In terms of actual posting structure, we may either store the fragID as one number or as two smaller numbers, assuming care is taken to use an appropriate index compression technique.

Our query processing can be seen as consisting of three phases, (i) identification of pages that contain all query words (whether or not they occur in the same version), (ii) for each such page a check if any version of the page contains all words (this requires retrieval of the doc/version table to understand how fragments map to versions), and (iii) computation of the actual score for a page or version. Note that this possibly also allows us to rank pages and versions based on other factors such as when and how long a page contained the query words, or other temporal factors that we do not explore in this paper.

Query Processing with Global Sharing: In this case, we process a query in a similar manner as in a standard index, by scanning lists from start to end. However, there are some differences: In standard indexes, we can often skip forward over many postings (using *nextGEQ()*), while the existence of global sharing now prevents us from doing so in many cases. For example, consider the case of a query for "cat","dog", and "sheep" where "cat" is the shortest list and "sheep" the longest, and "cat" occurs first in fragID $(5, 1)$, "dog" occurs first in $(2, 0)$, and "sheep" occurs first in $(9, 3)$. Suppose we start by reading the first posting in the "cat" list, with fragID $(5, 1)$. Then we cannot simply follow the standard DAAT approach: If the fragment $(5, 1)$ is also used in a version of page 9, then we have to make sure not to skip any fragments for page 9 in the "dog" list since a match for "dog" there would indicate that each of the three words occurs at least in some version of page 9. If the fragment $(2, 0)$ also occurs in page 5, then we cannot even skip any fragments for page 5 in the "dog" list.

In other words, global sharing restricts the amount of skipping we can perform in the index traversal when computing an intersection. In particular, in all lists, in addition to the stops implied by standard DAAT, we need to stop to check for the existence of any postings for fragIDs that are later reused in another page. If such a posting exists, we also need to stop to check for any postings in pages where those fragIDs are later reused – we can find the docIDs and thus fragIDs of those later pages through a lookup into the fragment reuse table. If reuse is concentrated on a few fragments that are reused by many other pages, then we can still get a reasonable amount of skipping in the index, but in other cases we end up traversing all postings. In general, when we find a posting that is later reused, we create a *ticket* that stores the information in the posting itself and the pages where the fragID is reused later, and we put these tickets into a priority queue organized by the docID of the next reuse. As we later visit postings that match a ticket, we update and eventually erase the ticket.

A few remarks about the costs of this scheme. Forward skips in DAAT processing can save a significant amount of CPU time as we can avoid uncompressing all postings (but the importance of this aspect may be more modest for very recent schemes such as [15] that can decompress postings at a rate of several GB per second). On the other hand, skipping seems to rarely save much on disk transfer times given current disk performance characteristics, as skips are rarely long enough to skip a large enough chunk of data on disk. (In our experiments, the query processor ended up almost always fetching all the list data). There is additional time and space overhead due to the priority queue; however, we find that most sharing is fairly local, i.e., between pages on the same site, and thus the queue stays fairly small most of the time.

Query Processing with Local Sharing: If we only allow local sharing, then the above algorithm simplifies in several ways. First, we do not need a fragment reuse table at all. Second, no tickets are required, and we can essentially run standard DAAT on the index to implement phase (i). On the other hand, we would expect a larger index size. Local sharing alone should already give decent benefits if there are many versions of each page, but of course will give no benefit at all for a single snapshot of pages with a certain amount of redundant content between different pages.

Discussion: In summary, our approach requires changes in query processing that may result in additional computational and memory costs. The details depend on our choice of sharing policy (local, global, or in between), but also on whether we assume AND or OR semantics for our query terms. For AND, we may not be able to perform as many forward skips, while for OR this is less of an issue. On the other hand, fast schemes for OR often use precomputed quantized score that take document lengths into account; it is not clear how to do this in our approach where a fragment may occur in several documents of different lengths. Finally, additional differences and challenges may arise once we add early-termination techniques to the query processor, but this is beyond the scope of this paper.

5.3 Index Updates

We now consider how to use our framework for efficient updating of the index. First, consider the case of a new version (or new page) being added to the index. In this case, we partition the page into fragments, look up the hashes of the fragments in the hash table, and then index only those fragments that do not find a match while discarding the others. In addition, we update or insert the appropriate records in the various tables. As we index the new fragments, we generate postings that are first inserted into a main-memory structure and later periodically merged into disk-based structures. One nice feature of the approach, and advantage over *Landmarks* [23], is that we do not even have to fetch the old version of the page from disk.

This leads to a very simple protocol for the case where, say, a crawler needs to update a remote index over a network. In the first phase, we send only the MD5 hashes of the fragments, and the index replies with a bit vector indicating which hashes found a match. In the second phase, we send only those fragments that did not find a match. We note that this protocol is of course very close to *rsync* [37], *LBFS* [25], and similar schemes for distributing redundant data sets. In general, our approach naturally combines with transmission and storage schemes based on block-wise elimination of redundancies, with interesting implications for possible applications in storage and file systems.

Finally, we can also support updates where a new version replaces an older version. In this case, in addition to creating new postings for new fragments, we also need to delete old fragments that are not used by any page anymore. This is done most efficiently by first adding a "delete fragment" command to a main-memory structure, and propagating the deletion to disk during the next periodic merge of the updates onto disk. One assumption in this approach is of course that we can hold a strong enough hash value for each fragment in memory; our experiments indicate that this is realistic for many scenarios. (If no incremental updates are performed, then of course the hash table is not needed at all during query processing.)

5.4 Application Scenarios

We now briefly summarize a few of the scenarios and applications that are covered by our framework.

- **Archival Search:** This is the main focus of our presentation here, where for each page we have a number of different versions.
- **Redundant Web Collections:** Pages from the same site, or even from different sites, often share common blocks of data that can be eliminated with our approach.
- **Email and Personal Files:** As also observed in [7], large amounts of email data, and collections of personal files, also frequently contain significant amounts of common data.
- **Versioning File Systems:** As indicated in the previous subsection, our techniques are uniquely suited for integration with versioning file systems that keep all versions of files, or with any storage systems that use block-wise redundancy elimination when storing data. Another interesting application in this direction might be for use in revision control systems for code or documents.
- **Distributed Indexing:** As also discussed, our approach allows efficient updating over a network. This includes indexing in distributed/P2P systems as well as functionalities such as the "search across machines" feature in Google Desktop Search that synchronizes indexes across machines.

6. EXPERIMENTAL EVALUATION

We now present our results from a preliminary experimental evaluation of our approach. We first describe the data sets, then evaluate the amount of redundancy detected by our schemes, and then provide some limited results for an actual compressed index structure and query processor implementing our framework.

Data Sets: The main data set for our evaluation was extracted from a set of 19 weekly crawls of several hundred large sites during Spring 2006 that was made available by the Stanford WebBase project. Each crawl consisted of about 2 million pages that were obtained by starting from the same set of seed pages in each site. However, due to changes in the site structure over time, this resulted in somewhat different sets of URLs that were crawled in each week. Thus, the set does not contain 19 versions of each page, or even most pages.

We preprocessed the set by removing all pages that we could not effectively parse (mostly pages primarily or completely in Flash or Javascript), and by removing all exact duplicates among the versions. Exact duplicates are easily handled with existing techniques and thus not a good measure of the efficacy of our approach. This left us with a total of 6,356,374 versions of pages from 2,528,362 distinct URLs. Thus, on average there were only 2.5 versions of each page, though some pages have more versions while many others have only one.

We show the cumulative distributions of the number of URLs and number of versions over the 19 weeks in Figure 6.1. As we see, about 40% of the URLs and 15% of the total data (versions) is concentrated in the crawl for the first week, while afterwards new versions and URLs arise at some smaller but fairly constant rate.

Number of Distinct Fragments and Term Positions: We first look at the resulting numbers of fragments and the amount of redundancy that we observed in the collection. In Figures 6.2 and 6.3, we see the reduction in the number of fragments and total number of term positions that occurs when we eliminate duplicate fragments under a local sharing policy. For the first crawl, where there are no different versions of the same URL, the ratio is (of course) essentially 1.0, with very minor savings due to fragments repeated within the same version of a page. When we use data from all 19 weeks, between 40% ($w = 300$) and 50% ($w = 100$) of all fragments (Figure 6.2) and positions (Figure 6.3) are eliminated. The results in Figure 6.3 are about 1% worse than those in Figure 6.2 as small fragments are slightly more likely to occur repeatedly.

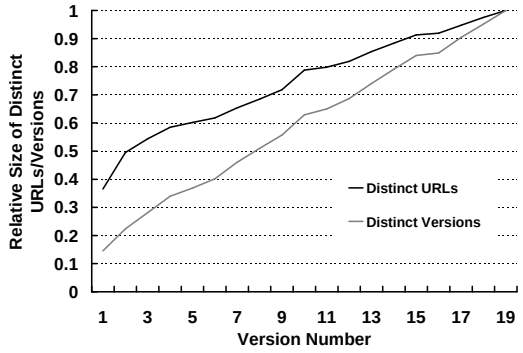

Figure 6.1: Increase in the number of URLS and versions versus crawl week.

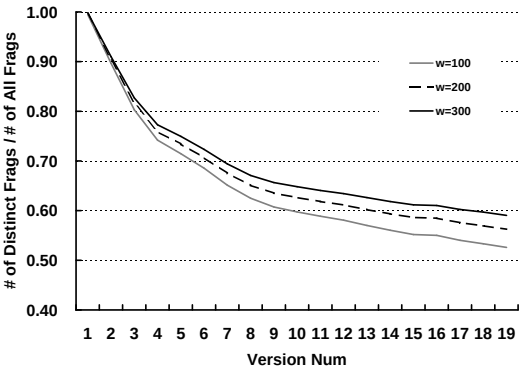

Figure 6.2: Cumulative percentage of unique fragments versus week of crawl.

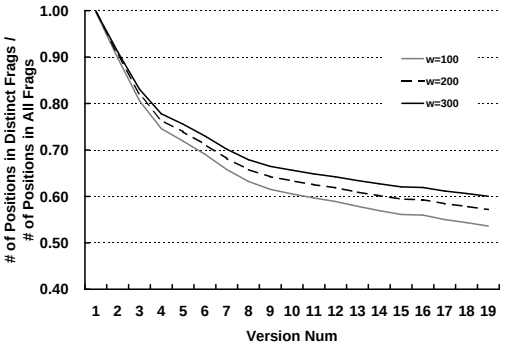

Figure 6.3: Cumulative percentage of positions within unique fragments versus week of crawl.

Next, we compare local versus global fragment sharing policies in Figure 6.4. We see that for our data set there is a significant additional benefit due to sharing between different pages. For a window size of $w = 50$, we observe almost a factor of 4 in reduction in the number of fragments, while even for $w = 300$ we get a factor of 2.5, when compared to an index with no redundancy elimination.

Figure 6.5 shows the results as a fraction of the corresponding numbers from a standard index, for both the number of fragments and the number of positions. We see significant benefits for global sharing over local sharing, and for local sharing over no sharing, that increase as we decrease the window size w. We note in particular that global sharing benefits more than local sharing from smaller

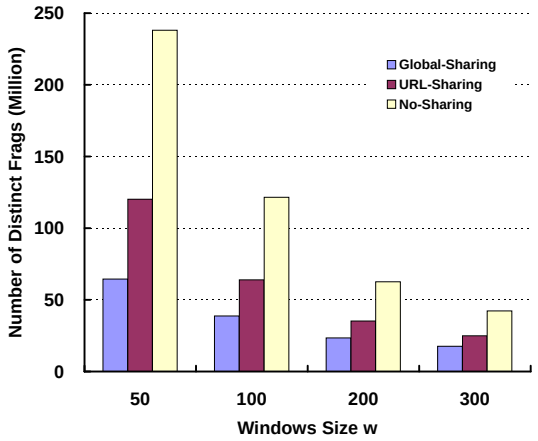

Figure 6.4: Comparison of the number of unique fragments under different sharing policies and window sizes.

window sizes, and thus the difference between the two methods increases for small w. This agrees with our expectation that different versions of the same page are more similar, and have larger blocks of common content, than similar but different pages. (Recall that the difference between local and global sharing is due to matches not available in other versions, but only in other pages.)

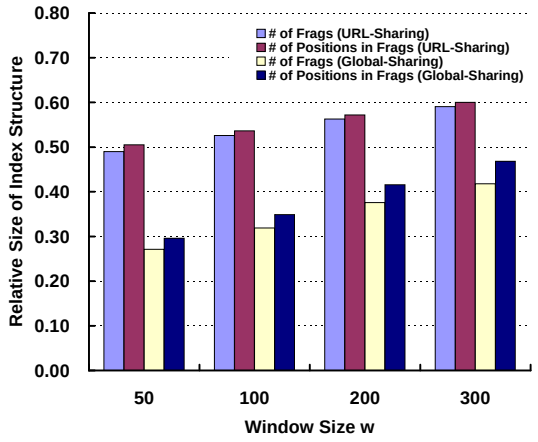

Figure 6.5: Relative reduction in the number of fragments and positions compared to a standard index, for different sharing policies and window sizes.

Compressed Size and Query Performance: We now show some preliminary results on the actual compressed index size and the cost of processing queries. All subsequent results are for a window size of $w = 100$ on the 19 crawls, and were performed on a machine with a 3.2 Ghz Pentium-4. We experimented with two different index compression techniques, the simple byte-aligned var-byte method [31], and the very recent word/bit-aligned Simple-9 technique in [1]. Both schemes were reimplemented and carefully optimized by us, resulting in decompression speeds of between 150 and 350 million integers per second for var-byte, and between 200 and 400 million integers per second for Simple-9. Overall, Simple-9 consistently outperformed var-byte in terms of decompression speed. As observed in [1], decompression speed varies based on the sizes of the numbers involved. For our standard index, we observed decompression speeds close to the upper limits of the ranges for docIDs and frequencies (which tend to be smaller values), and closer to the lower limit for position information (which also dominates the size of the index overall). We note that these numbers slightly shift as we move from docIDs to fragIDs with larger gaps but smaller frequency and position values.

	# of Frags (millions)	# of Posits (billions)	Index Size (GB)		
			var-byte	Simple-9	S9 no Pos
No Sharing	121.48	6.100	13.03	10.51	2.23
Local Sharing	63.89	3.271	7.34	5.23	2.68
Global Sharing	38.74	2.126	5.01	3.59	1.84

Table 6.1: Comparison of resulting index sizes for a standard index and for fragment-based indexes with local and global sharing. Shown from left to right are the number of fragments, total number of positions stored, index size under var-byte compression, index size under Simple-9 compression, and index size under Simple-9 when no position information is stored.

In Table 6.1, we compare the resulting index sizes for a standard index, an index with local sharing, and an index with global sharing. We note that Simple-9 achieves a smaller compressed size than var-byte, and that the advantage becomes more pronounced for local and global sharing. The reason is that var-byte cannot exploit the smaller frequency and position values in indexes with local and global sharing, as it must use at least one byte for every value that is compressed. For Simple-9, we observe a reduction in index size by a factor of 2 for local sharing, and almost 3 for global sharing, over a standard index. We also note that benefits are much smaller, and in many cases nonexistent, for index structures that do not store position information; for local sharing, we see an increase in index size while for global sharing we get a small benefit. This is not surprising since a fragment-based index essentially stores approximate position information (i.e., in which fragment a term occurs) that is not available in a standard index without positions. However, for sets with enough redundancy (many versions or very similar pages) our approach would also give significant benefits.

Next, we look at the performance of a prototype that implements query processing on fragment indexes. We limit ourselves here to local sharing and the var-byte compression scheme. The inverted index resides on disk, and the query processor uses a main-memory cache of size 512 MB for index data. We then issued 1000 queries selected at random from a publicly available trace of the Excite search engine, starting with an empty cache. (Performance should be slightly better when starting with a hot cache.) Table 6.2 shows that disk accesses during query processing decrease by almost 50%, which is slightly better than the reduction in index size for var-byte from Table 6.1. The reason is that the reduced index size also results in a higher hit ratio in the list cache, as a higher percentage of the index now fits in memory. Total wall clock time is also reduced significantly, though not as much as I/O. We note here that we are using a somewhat older version of our query processor, and that optimizations would decrease these numbers somewhat.

Setup	Million Frags	Block Reads	Time (s)
No Sharing	121.48	71,608	374
Local Sharing	63.89	38,881	295

Table 6.2: Number of fragments, number of 64KB blocks retrieved from disk, and wall clock time for processing 1000 queries, for a standard index and an index with local sharing of fragments.

Finally, in Table 6.3 we show the cost savings during index updates when compared to the baseline method of inserting each posting. We see that each new version of a page results in about 220 new positions, compared to about 950 for the baseline method. Overall, we believe that our results indicate the potential for significant performance improvements with our framework.

Crawl	New Versions	New Posits per Version	Same for Standard Index
17	279 K	219.66	924.56
18	227 K	228.71	960.93
19	248 K	254.01	995.59

Table 6.3: Cost of index updates per new version of a page, for versions arriving during the 17th to 19th crawl. We compare the average number of new position values per version that need to be indexed with local sharing, and the number of postings that would be generated using a standard index.

7. CONCLUDING REMARKS

In this paper, we have presented a new framework for indexing and query processing on textual collections with significant amounts of redundancy. This includes as important cases archival collections containing many versions of documents, and general collections of web pages, emails, or personal files that have some amount of redundancy. Our preliminary evaluation showed the potential for significant benefits, but there are several ways to further optimize our methods. For example, a new content-dependent file partitioning approach proposed in [35] might give slight improvements in the trade-off between the number of fragments and the amount of redundancy detected.

There are a number of intriguing possibilities for future research. It would be nice to combine our framework with the approaches in [7, 16] to further improve compression. In general, it seems that redundancies between versions or pages provide a new avenue for further improvements in index compression, similar to the gains in document and file compression that have been obtained from global redundancy elimination techniques (see, e.g., [22]).

We are particularly interested in exploring applications of the approach in file and storage systems (including versioning file systems and revision control systems). We observe that storage systems typically perform redundancy elimination in a manner that is completely transparent to the higher levels, and our indexing approach would thus have to be implemented at the lower levels for best performance. Extensions to regular expression search would also be of interest.

Finally, it might be interesting to reexamine the query processing issue in the case of significant global sharing. It could be that in this case, a pure DAAT approach is not the best due to the extra complexity, or that the ideas from [7] are more appropriate than unrestricted sharing.

Acknowledgments: This work was supported by NSF ITR Award CNS-0325777. We also thank the Stanford WebBase project for providing access to the crawl data used in our experiments.

8. REFERENCES

[1] V. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. of the 15th Int. Australasian Database Conference*, pages 61–67, January 2004.

[2] A. Arasu, J. Cho, H. Garcia-Molina, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technologies*, 1(1), June 2001.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[4] D. Blandford and G. Blelloch. Index compression through document reordering. In *IEEE Data Compression Conference*, April 2002.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the Seventh World Wide Web Conference*, 1998.

[6] A. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of the 12th Int. Conf. on Information and Knowledge Mana gement*, pages 426–434, November 2003.

[7] A. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. Shekita. Indexing shared content in information retrieval systems. In *Proc. of the 10th Int. Conf. on Extending Database Technology*, pages 313–330, October 2006.

[8] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Sixth Int. World Wide Web Conference*, 1997.

[9] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *Proc. of the 20th Int. Conf. on Very Large Databases*, pages 192–202, September 1994.

[10] R. Burns and D. Long. Efficient distributed backup with delta compression. In *Proc. of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, 1997.

[11] T. Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. Technical Report TR-66,, Experimental Computer Systems Lab, Stony Brook University, March 1999.

[12] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated web collections. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 355–366, May 2000.

[13] L. Cox, C. Murray, and B. Noble. Pastiche: Making backup cheap and easy. In *Proc. of the 5th Symp. on Operating System Design and Implementation*, December 2002.

[14] D. Hawking. Web search engines: Part 1 & 2. *IEEE Computer*, 39, June and August 2006.

[15] S. Heman. Super-scalar database compression between ram and cpu-cache. MS Thesis, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, July 2005.

[16] M. Herscovici, R. Lempel, and S. Yogev. Efficient indexing of versioned document sequences. In *Proc. of the 29th European Conf. on Information Retrieval*, April 2007.

[17] J. Hunt, K.-P. Vo, and W. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, 7, 1998.

[18] U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In *Proc. of Infocom*, 2005.

[19] U. Irmak and T. Suel. Hierarchical substring caching for efficient content distribution to low-bandwidth clients. In *Proc. of the 14th Int. World Wide Web Conference*, pages 43–53, 2005.

[20] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[21] M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems (TOIS)*, 17(4):406–439, Oct. 1999.

[22] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination wthin large collections of files. In *Proc. of the 2004 USENIX Annual Technical Conference*, June 2004.

[23] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks. In *Proc. of the 12th Int. World Wide Web Conference*, pages 102–111, May 2003.

[24] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. on Information Systems*, 14(4):349–379, 1996.

[25] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 174–187, October 2001.

[26] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. of the 1st USENIX Conf. on File and Storage Technologies*, 2002.

[27] S. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *Proc. of the 12th Int. World Wide Web Conference*, May 2003.

[28] K. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks*, 39:289–302, 2002.

[29] S. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *IEEE Symp. on Foundations of Computer Science*, 1996.

[30] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local

algorithms for document fingerprinting. In *Proc. of the 2003 ACM SIGMOD Int. Conf. on Management of Data*, pages 76–85, 2003.

[31] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 222–229, Aug. 2002.

[32] T. Schwarz, R. Bowdidge, and W. Burkhard. Low cost comparison of file copies. In *Proc. of the 10th Int. Conf. on Distributed Computing Systems*, pages 196–202, 1990.

[33] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. of the 27th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2004.

[34] N. Spring and D. Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proc. of the ACM SIGCOMM Conference*, 2000.

[35] D. Teodosiu, N. Bjorner, Y. Gurevich, M. Manasse, and J. Porkka. Optimizing file replication over limited bandwidth networks using remote differential compression. Technical Report TR2006-157-1, Microsoft Corporation, 2006.

[36] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1994.

[37] A. Tridgell and P. MacKerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, June 1996.

[38] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.

[39] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.

APPENDIX: DAAT Query Processing

To implement *document-at-a-time (DAAT) query processing*, it is useful to consider each inverted list as an input stream that can be accessed using the following operations:

- *openList(t)* opens the inverted list for term t and returns a pointer/cursor lp for this stream. *closeList(lp)* closes the list.
- *nextGEQ(lp, k)* advances cursor lp forward to the next posting with docID $\geq k$, and returns its docID. Returns MAXDOCID if none exists.
- *getPost(lp)* returns the complete posting pointed at by lp.

The algorithm for DAAT query processing is illustrated in the following snippet of code. Here, it is assumed that the terms in the query are sorted from shortest to longest. For each query, we first open all the lists, and then first access the shortest list, and then try to find matching elements in the longer lists. If an element is found to occur in all lists, then its score is computed. We note that operation *nextGEQ()* completely hides any details of the internal index organization, such as layout, caching, and compression methods.

```
for (i = 0; i < numterms; i++)  lp[i] = openList(qterm[i]);

for (docid = 0; docid < MAXDOCID; docid++)
{
  /* get next element from first (shortest) list */
  docid = nextGEQ(lp[0], docid);

  /* see if you find entries with same docID in other lists */
  for (i = 1, d = docid; (i < numterms) && (d == docid); i++)
    d = nextGEQ(lp[i], docid);

  if (d > docid)   /* docid not in intersection; continue */
    docid = d-1;
  else             /* docid in intersection; compute score */
  {
    for (i = 0; i < numterm; i++)  p[i] = getPost(lp[i], did);
    computeScore(p, numterm);
  }
}
for (i = 0; i < num; i++)  closeList(lp[i]);
```

Figure 8.1: Code from an simple implementation of document-at-a-time query processing.