Improved Index Compression Techniques for Versioned Document Collections

Jinru He
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
jhe@cis.poly.edu

Junyuan Zeng
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
jzeng04@students.poly.edu

Torsten Suel
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
suel@poly.edu

ABSTRACT

Current Information Retrieval systems use inverted index structures for efficient query processing. Due to the extremely large size of many data sets, these index structures are usually kept in compressed form, and many techniques for optimizing compressed size and query processing speed have been proposed. In this paper, we focus on versioned document collections, that is, collections where each document is modified over time, resulting in multiple versions of the document. Consecutive versions of the same document are often similar, and several researchers have explored ideas for exploiting this similarity to decrease index size.

We propose new index compression techniques for versioned document collections that achieve reductions in index size over previous methods. In particular, we first propose several bitwise compression techniques that achieve a compact index structure but that are too slow for most applications. Based on the lessons learned, we then propose additional techniques that come close to the sizes of the bitwise technique while also improving on the speed of the best previous methods.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval.

General Terms

Algorithms, Performance.

Keywords

Inverted index, index compression, versioned documents.

1. INTRODUCTION

Over the last few years, web search engines and other information retrieval tools have become the primary means of finding relevant information for millions of users. The largest search engines now have to answer tens of thousands of queries per second over billions of web pages. To do so, current search engines rely on a data structure called an *inverted index*, which allows efficient retrieval of all documents containing a particular term or set of terms. A lot of research over the last three decades has focused on how to efficiently build, compress, and access inverted index structures, resulting in the highly optimized implementations deployed in the current generation of search engines.

In this paper, we focus on an important special case of the text indexing problem that has received much less attention, the case of a versioned document collection, i.e., a collection where each document is represented by multiple versions. Important examples of such collections are web archives containing many past versions of web pages, the page history in Wikipedia, or revision control systems storing all past versions of program source code. Our goal is to build a concise and efficient inverted index structure for versioned collections, such that queries can be evaluated over all versions of all documents. A trivial way to build such an index would treat each version as a separate document. However, this is wasteful, since for a collection with 50 versions per document we obtain an index with 50 times the size of a single-version index, even if many versions are almost identical.

Thus, the main challenge in indexing versioned document collections is how to exploit the significant similarity that often exists between versions of the same document, in order to avoid a blowup in index size. Ideally, the resulting index should have a size that is proportional to the amount of change between the versions, rather than the total collection size. This problem has received some attention in the research community [2, 8, 4, 13, 31, 11], but we believe there is still room for improvements.

We are motivated by two important search applications, search in the Internet Archive and in Wikipedia, and our experiments use data from these collections. The Internet Archive (`www.archive.org`) is a non-profit organization that has collected more than 150 billion web pages since 1996 in an attempt to archive the World Wide Web. While current commercial search engines provide only access to the most recent snapshot of the evolving web, and simply replace old versions of pages with the most recently crawled version, the Internet Archive aims to also provide access to all previous versions. However, indexing all these versions is very expensive (and not currently done by the Archive), especially for a non-profit without the deep pockets of the major search engine companies. Our second application is Wikipedia, which keeps a complete history of all past versions of every article, and where it would be desirable to be able to search across the different versions. Other applications are search in revision control and document management systems, and indexing support for versioning file systems.

Our contributions in this paper are improved techniques for organizing and compressing inverted index structures for ver-

CIKM'10, October 26–29, 2010, Toronto, Ontario, Canada.

sioned document collections. In particular, we analyze typical properties of versioned document collections that make them highly compressible, and provide simple combinatorial upper and lower bounds for index size. Our main result is a set of index organization and compression schemes, building on previous work in [2, 13, 11, 1], that achieve improvements in both index size and query processing speed over all previous approaches.

The remainder of this paper is organized as follows. Next, we provide some technical background and discuss previous work. Section 3 discusses some typical properties of versioned document collections that help compression, and Section 4 establishes combinatorial upper and lower bounds. Section 5 provides improved practical schemes and evaluates them on data sets from Wikipedia and the Internet Archive. Finally, Section 6 provides some concluding remarks.

2. BACKGROUND AND RELATED WORK

In this section, we first give some background on versioned document collections, inverted indexes, and index compression techniques, and then discuss related work.

2.1 Background on Indexing

Most current search engines use an *inverted index* structure to support efficient keyword queries [32]. An *inverted index* for a collection of documents is a structure that stores, for each term (word) occurring in the collection, information about all locations where it occurs. For each term t, the index contains an *inverted list* I_t consisting of a number of *index postings*. Each posting in I_t contains information about the occurrences of t in one particular document d, usually the ID of the document (the docID), the number of occurrences of t in d (the frequency), and possibly other information such as the locations within the document. In this paper, we assume that postings have docIDs (or version IDs) and frequencies, but we do not consider the case of a *positional index* where within-document positions are stored in the postings.

Index Compression: The postings in each list are usually sorted by docID and then compressed using any of a number of techniques from the literature [32]. Most techniques first replace each docID (except the first in the list) by the difference between it and the preceding docID, called a *d-gap*, and then encode the d-gaps using some suitable integer compression algorithm. Using d-gaps instead of docIDs decreases the average value that needs to be compressed, resulting in better compression. These values have to be summed up again during decompression, but this can be done quite efficiently. The d-gaps and frequencies are often stored separately, and thus we compress sequences of d-gaps and frequencies.

A large number of inverted index compression techniques have been proposed; see [32] for an overview and [30] for a recent experimental evaluation of state-of-the-art methods. In our work here, we utilize two techniques that have been previously applied to versioned document collections [11], *Interpolative Coding* (IPC) and *PForDelta* (PFD).

Interpolative Coding is a technique introduced in [17] that was designed for the case of clustered or bursty term occurrences, such as those encountered in longer linear texts (e.g., books). IPC achieves a very small index size in many scenarios, but it is not very fast in terms of decompression speed. PForDelta is a family of compression schemes first introduced in [12, 33], and further optimized in [30, 29], that allows extremely fast decompression (beyond a billion integers per second per core) while also achieving a fairly small compressed size. Here, we use a version of PFD called OPT-PFD described in [29] that achieves very good compression on clustered data, in some cases coming close to IPC in size.

Inverted lists are often organized into blocks that can be independently accessed, thus enabling forwards skips during query processing. We use such blocked indexes throughout this paper, with a block size of 128 postings.

Versioned Collections and Two-Level Indexes: We define a versioned document collection D as a set of documents $d_0, \ldots d_{n-1}$, where each d_i has m_i versions $d_i^1, d_i^2, \ldots, d_i^{m_i}$. (In some cases, it is convenient to define d_i^0 as the empty document.) We assume a linear history, and do not try to model the case of branches (forks) in the revision history, though we believe our ideas could be adapted to this case.

We define the first-level index of a versioned document collection D as an inverted index where the inverted list for a term t contains a posting for document d_i if at least one version d_i^j of d_i contains t. For a term t that occurs in at least one version of d_i, we define the bit vector of t and d_i as an array of m_i bits such that the j-th bit is set to 1 iff version d_i^j contains t. Similarly, we define a frequency vector that contains in its j-th entry the frequency of t in d_i^j. (When the bit vector is already known, it may be convenient to think about the frequency vector as having entries only for those versions that contain t.)

A two-level index structure for a versioned document collection consists of the first-level index, and a second level storing all the bit and frequency vectors. All our constructions in this paper use the two-level approach, which was first proposed in [1] in the context of non-versioned collections, and then adapted to versioned collections in [11]. This approach allows efficient query processing by first running a query on the small first-level index, and then fetching and decompressing any necessary bit and frequency vectors.

The first-level index contains only docIDs, and no frequency values. It is compressed using standard index compression techniques such as IPC or PFD, since there is no obvious structure in the data that distinguishes this case from that of a non-versioned collection. The main challenge is the modeling and compression of the bit and frequency vectors in the second level, and as we will show, these vectors have a wealth of interesting structure that can be exploited.

2.2 Related Work

We now discuss previous related research, including work on indexing and querying versioned document collections, on indexing collections with similar documents, and on storage and transmission of similar and versioned documents.

Positional Indexing of Versioned Collections: Previous work on indexing versioned text collections falls into two classes, work on positional indexes [31] and work on non-positional indexes [2, 8, 4, 13, 11]. For positional indexes, [31] uses content-dependent string partitioning techniques [22, 14, 27] to split each document into fragments, and then removes duplicate fragments from the collection; this results in significant reductions in index size and also supports very efficient updates as new versions are added to the collection.

We note here a basic difference between positional and non-positional indexes. In the former case, similarity is based on common substrings; this way a change in position information due to an insertion or deletion before the start of the substring can be efficiently handled by changing the offset of the substring in the document. In the latter case, we have a set- or bag-oriented model where similarity is based on common subsets of terms; for non-positional indexes this performs much better than a substring-oriented approach,

and thus techniques for positional indexes are of limited relevance. However, improved techniques for positional indexes are an interesting open problem.

Non-Positional Indexing of Versioned Collections: The approaches in [2, 8, 4, 13, 11] consider non-positional indexes, our focus in this paper. The earliest work we are aware of is that in [2], which proposes to index the differences between the versions rather than the versions themselves. In particular, [2] indexes the last version, and then adds *delta postings* to store changes from the last to previous versions. We will refer to this basic approach as DIFF (for difference).

In more recent work in [8], the set of terms of a document is organized into a tree structure, where each node has some private and some shared terms, and each node inherits its ancestors' shared terms. The versions are then expressed in terms of these sets of terms, and the sets of terms are indexed. Followup work in [13] identifies subsets of terms that are contained in a range of consecutive versions. Such subsets are then treated as virtual documents and indexed, and the main challenge is to minimize the number and size of virtual documents. This can be done by reordering the versions in an optimal manner, resulting in an instance of the NP-Complete Multiple Sequence Alignment (MSA) problem. However, in most cases a simple ordering of versions by time appears to achieve very good results, and this is the approach we use later in our constructions. We nonetheless refer to this general approach as MSA.

Another approach in [4] is based on the idea of coalescing index postings corresponding to consecutive versions if they contribute almost the same score to the overall ranking function. This is a lossy compression technique that uses an index with precomputed quantized scores, and thus it is not easily comparable to other approaches.

Recent work in [11] compares the DIFF and MSA approaches to two additional techniques. One technique, called Sorted, simply assigns docIDs to the versions of a document in the natural order, and then relies on standard index compression techniques to exploit the resulting clustered nature of the index data. This can be seen as a non-lossy variant of the approach in [4], and it is shown to do much better than the trivial baseline, but not as well as either DIFF or MSA. Another method, called HUFF, uses the two-level approach in [1], and compresses the bit and frequency vectors in the second level using a simple hierarchical Huffman coding scheme adapted from [10]. This method outperforms all other methods in terms of size, but results in slightly slower query processing due to the use of Huffman coding.

Thus, the work in [11] provides a recent comparison of existing methods that we use as a baseline for our results. We also use the same Wikipedia and Internet Archive data sets as [11], allowing a direct comparison of the numbers. Overall, we will show significant improvement in both index size and access speed over previous methods.

Query Processing in Versioned Collections: Another related problem is how to perform query processing in versioned document collections, and in particular what types of operations should be supported. This is a non-trivial problem that has received only limited study. The simplest type of query treats all versions as separate documents, and thus would return the most relevant versions over the entire history of the collection (probably with additional filtering to remove multiple versions of the same document). It is likely that in many scenarios, users would also like to be able to restrict a query to a limited time interval (e.g., the most relevant pages during 1998); this could be achieved either by post-filtering, or through specialized index structures that support more efficient range queries over time. Recent work in [16] proposes *durable search*, where the goal is to return documents consistently ranked high over a range of time, and describes a number of algorithms for this problem.

In general, in this paper we are trying to sidestep the question of what the right set of query operations is. We evaluate our index structures by running queries over all versions, with no temporal restrictions, but we believe that our techniques are fast enough to allow efficient filtering and aggregation on top in order to implement other operations. Of course, specialized index organizations for range search and durable search should outperform more general index layouts, and how to optimize versioned index structures for these cases is an interesting problem for future work.

Succinct Indexing of Similar Documents: There are several techniques that exploit similarities between distinct documents (i.e., not different versions of the same document) for better index compression. This problem is closely related to that of indexing versioned collections, but also differs from it in important ways: First, similarities between distinct documents are typically smaller, leading to more limited gains, and second, an important part of the problem is to identify which documents are similar, while in versioned collections this information is more or less implied.

The most common approach to indexing similar documents is based on reordering the documents in the inverted lists, by assigning consecutive or close-by docIDs to documents with high similarity [7, 24, 26, 6, 25, 29]. This results in a more skewed d-gap distribution in the inverted lists, with many more small d-gaps and a few larger ones, leading to a reduction in index size under common index compression schemes. The Sorted method in [11] is an application of this idea to versioned collections.

Another approach was studied in [1], where similar documents are clustered into disjoint groups and then indexed by a two-level index structure. While in [1] this does not reduce index size, it leads to an increase in query processing speed. The HUFF approach in [11] and our techniques here adapt the two-level approach to versioned collections, where it results in improvements in both size and speed.

Storing Redundant Document Collections: There has been a significant amount of research on the problem of reducing space or network transmission costs in storage systems by exploiting redundancy in the data. This includes the Low Bandwidth File System [18], various storage and backup systems [9, 5, 15, 21, 27], and remote file synchronization techniques [28, 23, 19]. This work differs from our work in that the goal is to reduce the size of the collection rather than the size of the index. However, some of the techniques are also relevant to our work, and we will adapt ideas from [19] based on Communication Complexity for our analysis in Section 4.

3. DATA SETS AND DATA ANALYSIS

In this section, we describe the data sets that we use, and then perform a preliminary analysis of the data in order to detect patterns that can be exploited for better compression.

Data sets: In our experiments, we use versioned collections obtained from Wikipedia (Wiki) and the Internet Archive (Ireland). The Wikipedia data set consists of 0.24 million distinct documents with 35 versions per document on average, resulting in slightly more than 8 million pages. This is about 10% of the English version of Wikipedia. The Internet Archive data set consists of 1.06 million documents from the Irish web domain, collected between 1996 and 2006, with

15 versions per document on average. The data sets were used in [11], thus allowing a direct comparison of results.

Data analysis: Next, we perform a simple analysis of the data in order to identify four properties that lead to good data compression. For each property, we also point out which previous methods take advantage of it.

Property 1: Most changes are small. Figure 1 shows the distribution of the sizes of the changes between versions. Here, the size of the change is the number of unique terms that are either added or removed between versions. As we see, at least half of all versions in Wiki and Ireland differ by less than 5 terms that are added or removed. (Note that we are looking at a set-based model – a change size of 0 does not mean that two consecutive versions are identical, but only that they are based on the same set of terms. All identical versions were removed from the data.) This is of course not surprising, and the main reason why a versioned index structure should be much smaller than a standard structure that treats each version as a separate document. All existing techniques for versioned collections exploit this property.

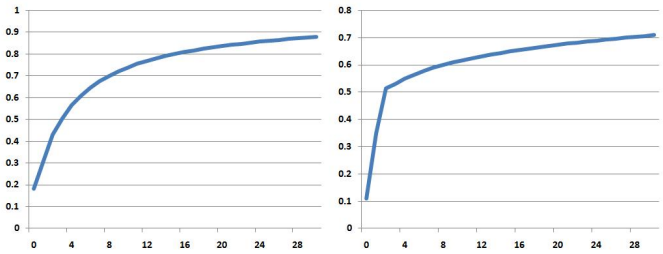

Figure 1: Cumulative distribution of change size for Wiki (left) and Ireland (right). The x-axis is the size of the change (symmetric difference between consecutive versions) while the y-axis shows the fraction of versions with at most this amount of change.

Property 2: Terms changes are bursty. Figure 2 shows the distribution of the change distance, defined as the number of versions between two changes in a term bit vector. From the figure, we see that the longer a term stays unchanged, the more unlikely it is to change in the next step. Conversely, terms that have just been added or removed are more likely to disappear or reappear again in the next few steps. (To limit the impact or the truncated history, we removed terms that stay until the last version.) We note that all existing compression techniques also exploit this property; in the case of DIFF, MSA, and Sorted this happens through use of compression methods such as IPC and PFD that exploit data clustering, while in HUFF the Huffman table catches on to this pattern.

Property 3: Change size is bursty. Figure 3 shows the cumulative size of the amount of change when we sort versions from largest to smallest change. This is the flip side of Property 1: while most changes are small, less than 10% of all versions are responsible for more than 50% (Wiki) and 70% (Ireland) of the total change. So, large changes are rare, but account for much of the total change. (However, the distribution of change does not seem to follow power law.)

We note that none of the existing techniques really exploits this property, but that it is potentially very useful. If we have a term that exists in the current version, then knowing, say, that the next 3 versions have very little change, but that the fourth version has a large amount of change, would lead us to guess that the term is most likely to change in this fourth version than in the others. In fact, in the next section, we show that even a simple combinatorial technique that uses only this property, and that knows the change size for each version, outperforms all previous methods in terms of index size. Most of the improvements in this paper are based at least indirectly on exploiting this property.

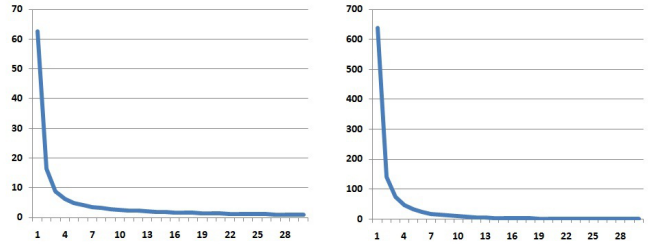

Figure 2: Distribution of change distance for Wiki (left) and Ireland (right). The x-axis is the change distance and the y-axis shows the number of term changes (essentially postings in DIFF) with this distance in millions.

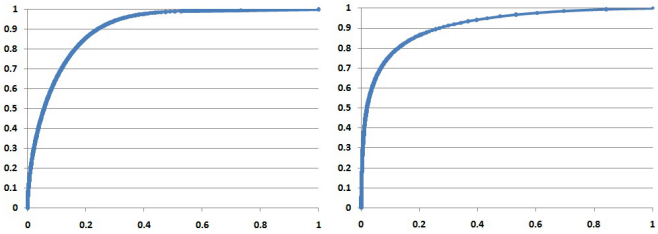

Figure 3: Cumulative distribution of change size for Wiki (left) and Ireland (right). We sort versions in decreasing order by change size, and then plot on the y-axis the cumulative amount of change.

Property 4: Terms are dependent. One more property of the collections is that terms often come and leave together. In particular, we found that a pair of terms in Wiki that exists in a given version has a 48.8% change of being deleted at the same time if it was also added at the same time, while the chance was only 30.5% if the words were added at different times. For Ireland, the numbers were 55% and 34.6%. This is of course not really surprising and can be explained in several ways. Terms that are added in the same version are often next to or close to each other, so that a future change often also impacts both terms. Or, this could be mainly a result of Property 3 since many pairs of terms will be added in a large update and then removed together in the next large update. Or it could be a consequence of Property 2. We note that this property is only exploited in the MSA approach [13], and is an important reason for its good performance compared to DIFF.

4. A COMBINATORIAL APPROACH

In this section, we discuss the index compression problem from a combinatorial perspective, and derive worst-case lower bounds and upper bounds for the problem. The techniques we use are fairly simple, and motivated by previous work in Communication Complexity [19], but we are not aware of any previous formal analysis of versioned index compression. We note that the algorithms in this section are not directly useful in practice, as their bitwise approach to compression is too slow for a state-of-the-art query processor. However, we believe the results are of interest on their own, and they also inform the practical approaches in subsequent sections.

In the following, we assume the two-level approach to indexing as described earlier. Thus, we have a small first-level index that specifies for each term which documents contain the term in at least one version, and a second-level index containing compressed bit vectors specifying which versions

contain the term. (We focus on the case of docIDs only, but the approach can be extended to indexes with frequency values.) The first-level index is compressed using standard techniques, as the problem of compressing this level is similar to the standard non-versioned index compression problem. Our focus here is on the second level, and the modeling and compression of the bit vectors, which encapsulates the essence of the versioned index compression problem.

We start out with an analysis of a simple model based on set difference in Subsection 4.1, where we establish upper and lower bounds in this model. Subsection 4.2 extends the results to models that use term appearances, term deletions, and term reappearance, and provides some experimental results on real data. In Subsection 4.3, we provide another upper bound that uses additional features to obtain better results, and finally we discuss conclusions from this section.

4.1 A Simple Model Using Set Difference

Let $D = \{d_0, d_1, \ldots, d_{n-1}\}$ be a collection of versioned documents. Consider one document $d_i \in D$ consisting of m_i versions $d_i^1, d_i^2, \ldots, d_i^{m_i}$. For convenience, we define d_i^0 as the empty document. The size of d_i, $s(d_i)$, is the number of terms that appear in at least one version of d. For each d_i^j, we define $ch(d_i^j)$, the change in d_i^j, as the size of the symmetric difference between d_i^j and d_i^{j-1}, that is, the number of terms added or removed between versions d_i^{j-1} and d_i^j.

As discussed before, we expect that the index size of a versioned document collection should depend on the amount of change between versions. In the following, we show that this can be formalized, by deriving suitable lower and upper bounds for index size based on change. In particular, we can show the following simple information-theoretic lower bound:

Theorem: For any numbers $c_1, c_2, \ldots, c_m$ and s, and any index compression scheme, there exists a document d with $s(d) = s$ and m versions such that $ch(d^j) = c_j$ for all j, that requires an index size of at least

$$\sum_{j=1}^{m} \log_2 \left(\binom{s}{c_j} \right) \geq \sum_{j=1}^{m} c_j \cdot \log_2 \left(\frac{s}{c_j} \right) \text{ bits}$$

.

Proof Sketch: Given a set of s terms, there are at least

$$X = \prod_{j=1}^{m} \binom{s}{c_j}$$

distinct documents d that satisfy the stated constraints on change between versions. An index needs to be able to distinguish between all such documents, and thus there must exist a document with index size at least $\log_2(X)$ bits. **End.**

We note that this is not just a bound on index size, but also a bound on the amount of bits needed to store the versions themselves (modeled as sets rather than sequences of terms). More precisely, it is a lower bound on the second level of the index, that it, it holds even if we already know the information in the first level of the index. The theorem is stated for one document, but generalizes to collections by summing up over all documents, with each document having its own constraints on the change between versions. The result and its proof are inspired by previous work on the communication complexity of exchanging similar documents in [19], which we apply to a set-based model with multiple versions.

Next, we will derive an upper bound on second-level index size based on the amount of change between versions, and relate it to the lower bound. The second level consists of bit vectors that we need to represent in a concise manner. Consider a bit vector $\vec{x}$ of length m for a term t that occurs in at least one version of an m-version document d. It is well known that compression is related to prediction, and one way to compress such a bit vector is to derive a probability for the next bit to be 0 or 1, given the previous bits and the bounds c_j on change between versions. These probabilities can then be used to drive a binary coder matching the entropy bound (e.g., binary arithmetic coding) to code the bit vector.

The main insight here is that the values c_j are very useful in predicting the next bit of the bit vector. Assume that version d^{j-1} of d contained a term t, and we would like to guess how likely it is that t will also be present in version d^j. This should depend on $ch(d^j)$, the amount of change between the two versions, and if we assume that terms affected by change (terms that disappear or appear in the next version) are selected at random from all eligible terms, than t has a chance of $ch(d^j)/s(d)$ of disappearing in version d^j. Thus, we will code the j-th bit of the bit vector by using the probability $ch(d^j)/s(d)$ in the binary coder. Overall, we use the following probabilities to encode a bit vector $\vec{x} = x_1 x_2 \ldots x_m$:

$$pr[x_1 = 1] = \frac{ch(d^1)}{s(d)}$$

$$pr[x_j | x_1 x_2 \ldots x_{j-1}] = \begin{cases} \frac{ch(d^j)}{s(d)} & x_j \neq x_{j-1} \\ 1 - \frac{ch(d^j)}{s(d)} & x_j = x_{j-1} \end{cases}$$

We now analyze this approach. For simplicity, assume that $ch(d^j) \leq s(d)/2$ for all j. (Otherwise, we can simply use the inverse of d^j.) We also assume that bit vectors are compressed optimally and in a continuous manner – a practical arithmetic coder would have some additional waste in bits, and we would also need to periodically align the data with bit or byte boundaries in order to allow retrieval of particular bit vectors, but this is harder to model. Then we can show:

Theorem: The number of bits used by the above algorithm for a document d is

$$\sum_{j=1}^{m} \left(ch(d^j) \cdot \log_2 \left(\frac{s(d)}{ch(d^j)} \right) + (s(d) - ch(d^j)) \cdot \log_2 \left(\frac{s(d)}{s(d) - ch(d^j)} \right) \right)$$

$$\leq 2 \cdot \sum_{j=1}^{m} ch(d^j) \cdot \log_2 \left(\frac{s(d)}{ch(d^j)} \right)$$

The bound follows from the fact that when we look at the j-th bit of all the $s(d)$ bit vectors for d, there are $ch(d^j)$ cases where we have to encode an $x_j \neq x_{j-1}$, and $s(d) - ch(d^j)$ cases where we encode an $x_j = x_{j-1}$. The well-known formula for entropy then gives the expression in the first line, which can be easily bounded by the second line.

Thus, at least under the idealized assumptions about coding and data alignment, the algorithm achieves a size within a factor of 2 of our lower bound. We note that the factor of 2 can be improved for values of $ch(d^j)$ bounded away from $s(d)/2$. (However, we were unable to come up with a useful form for such an improved bound.) As shown in our later experiments, the actual gap on real data is much smaller.

In general, there are two challenges to designing even tighter upper bounds. First, our lower bound is based on the size of a non-trivial combinatorial space (the set of documents satisfying the constraints) that is not easy to map to a dense

set of bit representations. Second, an upper bound for index structures needs to be able to separately decode the bit vectors for a particular term, and simply labeling the space of possible documents in an optimized way would not assure this. Our approach based on probabilities clearly allows decompression on a per-term basis, but at some inefficiency in representing the document space.

4.2 Extensions and Experimental Results

The results in the previous section are based on only the information about the total amount of change between versions. We now improve these bounds using additional constraints, and state upper and lower bounds for each case.

In the first case, assume that rather than just knowing the change $ch(d^j)$ between d^j and d^{j-1}, we know the number of newly added terms $ins(d^j)$ (insertions) and the number of removed terms $del(d^j)$ (deletions). Given this, we can further restrict the combinatorial space, resulting in the following worst-case lower bound on the number of bits:

$$\sum_{j=1}^{m} \log_2 \left(\binom{s(d^{j-1})}{del(d^j)} \cdot \binom{s(d) - s(d^{j-1})}{ins(d^j)} \right),$$

where $s(d^j)$ is the size (number of distinct terms) of version d^j. (Note also that $del(d^1) = 0$ by definition.) Furthermore, if we divide insertions into first occurrences $fo(d^j)$ (where a term occurs for the first time) and reappearances $re(d^j)$ (where a term reappears after having previously been deleted), we obtain the following worst-case lower bound on the number of bits:

$$\sum_{j=1}^{m} \log_2 \left(\binom{s(d^{j-1})}{del(d^j)} \cdot \binom{\sum_{i<j} fo(d^i) - s(d^{j-1})}{re(d^j)} \cdot \binom{s(d) - \sum_{i<j} fo(d^i)}{fo(d^j)} \right)$$

We can also derive upper bounds for these two cases, by assuming that in each step, terms affected by a change (insertion, deletion, first occurrence, or reoccurrence) are selected at random from all eligible terms. This gives us the following probabilities to drive a binary coder:

$$Pr[x_1 = 1] = \frac{s(d^1)}{s(d)}$$

$$Pr[x_j = 1 | x_1 x_2 \ldots x_{j-1}] = \begin{cases} \frac{ins(d^j)}{s(d) - s(d^j - 1)} & x_{j-1} = 0 \\ \frac{s(d^j) - ins(d^j)}{s(d^{j-1})} & x_{j-1} = 1 \end{cases},$$

and for the second case:

$$p[x_1 = 1] = \frac{s(d^1)}{s(d)}$$

$$p[x_j = 1 | x_1 x_2 \ldots x_{j-1}] =$$

$$\begin{cases} \frac{fo(d^j)}{s(d) - \sum_{i<j} fo(d^i)} & x_i = 0, 1 \le i < j \\ \frac{s(d^j) - fo(d^j) - re(d^j)}{s(d^{j-1})} & x_j = 1 \\ \frac{re(d^j)}{\sum_{i<j} fo(d^i) - s(d^{j-1})} & else \end{cases}$$

Next, we apply the three different types of upper and lower bounds to the Wiki and Ireland data sets described in Section 3. The results are shown in Table 1.

	Wiki		Ireland	
	upper	lower	upper	lower
Case 1	117	114	295	288
Case 2	92	89	267	261
Case 3	92	88	254	249

Table 1: Compressed size in MB for Wiki and Ireland, for our three upper and lower bounds. The lower bound here is a worst-case bound for a class of inputs, and does not imply a lower bound for the particular data sets used here. To compare numbers in the section 5, we need to add the size of the first level index as shown in Table 2

To implement the upper bound for Case 1, we need for each version d^j the value of $ch(d^j)$; this can be stored at a cost of about 1 byte per version. For Case 2, we need $ins(d^j)$, $del(d^j)$, and $s(d^j)$. Note that $s(d^j)$ is probably stored anyway as it is needed, e.g., for basic ranking under BM25 or cosine, and that $del(d^j)$ and $ins(d^j)$ can be computed from $ch(d^j)$ and $s(d^j)$; in this case the extra space is the same as for Case 1. Finally, for Case 3, we need $fo(d^j)$ and $re(d^j)$ instead of just $ins(d^j)$, resulting in an additional about 1 byte per version in the collection.

Looking at the results, we see that the upper and lower bounds are quite close, and that adding additional constraints results in visible improvements, particularly when moving from Case 1 to Case 2. The most surprising result for us, however, was that even the simple upper bound in Case 1 is much better than the best bound from previous work, the Huffman-based bit vector encoding in [11], which achieves $140MB$ on Wiki and $304MB$ on Ireland.

4.3 Feature-Based Prediction

Results in the previous subsection showed that a simple coding method that focuses on exploiting the amount of change in each version beats all previous published methods in terms of index size. The previous methods exploit various other patterns in the data set, but none of them uses the amount of change in each version in a principled way. This raises the question of whether we can improve the upper bound by using the amount of change as well as other patterns.

We again take a bitwise approach where we model the probability of the next bit in the bit vector being 0 or 1. However, in addition to the amount of change, we want to use various other *features* that might lead to a better prediction. In particular, from Section 3 we know that the likelihood of a term disappearing in the next version depends on how long the term has been in the document; recently added terms are more likely to disappear again. It is difficult to derive a good explicit formula for the probabilities without a better model for generating versioned documents (i.e., better than our earlier model where affected terms are selected at random), and thus we take a simple statistical approach where we analyze the complete data to derive a table of precomputed probabilities. We use the following features to predict the next bit:

- $F1$: the value of the current bit.
- $F2$ to $F4$: the lengths of the previous 3 ups and downs in the bit vector. Thus, if F2 is 5, the value of the current bit has been the same for 5 versions.
- $F5$: The number of 1's seen in the bit vector so far, divided by the total number of bits so far.
- $F6$: if F1=1, the number of term deletions between the

current and the next version; otherwise, the number of term insertions.

- $F7$: the inverse document frequency of the term.
- $F8$: the relative position in the bit vector, defined as the current bit position i divided by the total length m of the bit vector.

We implement this model using a k-D tree [3], which is a data structure for organizing k-dimensional points. In particular, our goal is to partition our 8-dimensional feature space into regions such that in each region the probability of the next bit being 1 is roughly uniform. We store this probability with the region, so we use it to arithmetic-code the next bit.

To construct the k-D tree, we go over all bit vectors, and obtain the set of corresponding 8-dimensional points, with each point having a label of 0 or 1 corresponding to the next bit. We recursively partition the space by looking at possible cuts along each axis, for the cut that achieves the largest decrease in overall entropy. (This is similar to the MHIST algorithm for constructing multi-dimensional histograms in [20].) We recursively apply cuts until the entropy is below a selected threshold. A lower threshold will result in a better prediction, but also in a larger tree stored as meta data.

We see from Figure 4 that approaches achieve moderate additional improvements over the formula-based approach from the previous subsection. The best results we get are 77.5 MB for Wiki and 209 MB for Ireland for the second level of total index. There is a trade-off between tree size and prediction quality. Note that using a k-D tree may appear cumbersome and impractical, but our goal here was to see how far we can reduce the index size by using many available features, and to use this as guidance for more practical methods in subsequent sections.

4.4 Discussion

We now briefly discuss the results from this section. Our main contribution was a set of upper and lower bounds for versioned indexing inspired by results in communication complexity. The most useful take-away from this was that a combinatorial approach that just exploits information about the amount of change in each version can beat all previous methods, which fail to exploit this information. Moreover, adding additional features can result in modest further gains.

However, the upper bounds in this section are not directly applicable in practice because of their bitwise nature. That is, for every bit in the bit vector, we need to call an adaptive arithmetic coder using the derived probabilities. While such coders can decode millions of bits per second on current CPUs, they are still by one to two orders of magnitude slower than index compression methods such as PFD or even the slower IPC. Thus, a query processor based on these methods would be too slow for most practical applications. Our challenge in the next sections is to design methods that exploit information about the amount of change without a bitwise approach that looks at one version at a time. We will show that this can be achieved through appropriate reorderings of the bit vectors, resulting in methods with both smaller index size and faster index access than previous results.

5. BETTER PRACTICAL METHODS

The approaches from the previous section achieve very good compression but are not practical due to their use of a bitwise arithmetic coder that would make decompression and thus query processing very slow. In this section, we propose alternative methods based on modifying the DIFF and MSA

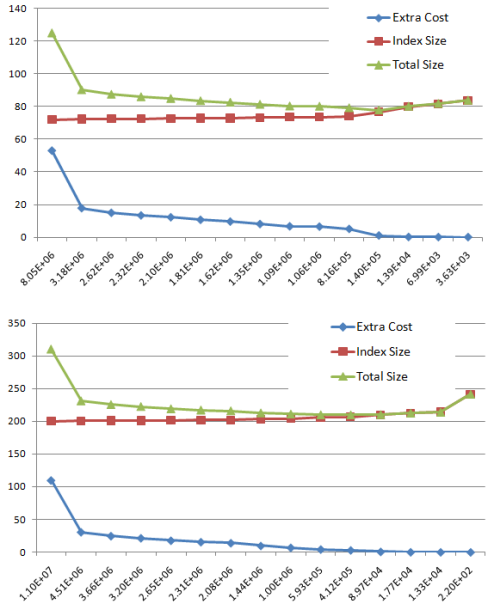

Figure 4: Compressed second-level index size for Wiki (up) and Ireland (bottom), for various entropy thresholds. The x-axis shows the resulting number of leaves in the k-D tree, while the y-axis shows the index size in MB. Again, the first-level index size needs to be added to compare the numbers in section 5

approaches that are also able to exploit knowledge about the amount of change between different versions. The resulting index sizes are close to the best from the previous section, with query processing speeds that are faster than that of all previous methods in the literature.

The new algorithms are based on three modifications to DIFF and MSA described in the following. In particular, we convert DIFF and MSA into two-level methods [1], apply a reordering operation on the versions (i.e. the bits of the bit vectors), and propose a hybrid between DIFF and MSA.

5.1 Two-Level DIFF and MSA

In a nutshell, both DIFF and MSA work by first creating artificial *virtual documents* from versions, and then indexing these virtual documents instead. In DIFF, we create one virtual document for each version, consisting of the symmetric difference between this and the previous version (i.e., the terms that were either added or removed). In MSA, each virtual document represents a range of versions; thus, one virtual document may correspond to versions 3 to 8 and contain all terms inserted in version 3 and deleted in version 9. This means that DIFF results in more postings than MSA, since it needs one posting to signal the arrival of a term, and one posting to signal departure. MSA uses only one posting for this, but creates more virtual documents, thus resulting in a larger docID space and larger d-gaps.

DIFF and MSA can be adapted to two levels as follows. The first-level index is always the same for all methods, and stores which documents have at least one version containing a particular term. For the second level, DIFF now looks at the relevant bit vector corresponding to the virtual documents rather than versions, and compresses this bit vector by storing the gaps between 1 values in the bit vector. This is in fact very similar to standard DIFF, except that gaps are now only created within a single bit vector and not across documents. We then compress all these gap values using either IPC or PFD with a block size of 128 values. In addition,

we also need to store an indicator bit for every gap to signal when a gap is the last gap in the current bit vector.

For MSA, the construction is similar. In this case, we have a bit vector with one bit for each virtual document, and store the gaps between 1 values in this bit vector. As with DIFF, we compress these gaps using IPC or PFD, and also add an extra bit for each gap to indicate when a gap is the last one. As we show in our experiments further below, this change of DIFF and MSA into two-level methods results in reductions in index size and query processing costs. In the following, we refer to these methods as 2-DIFF and 2-MSA.

5.2 Bit Vector Reordering

The two-level variants of DIFF and MSA still do not exploit the information about the amount of change between versions that turned out to be so valuable in the previous section. While it is conceptually simple to use this information in bitwise methods, where we only need to take the amount of change of the next version into account, this is much harder for methods such as DIFF and MSA that store longer gaps. Intuitively, to better model these longer gaps we would have to look at the change information not just of the next version, but of many future versions.

We now describe a simple reordering trick that achieves a similar result in an indirect way. Consider again the virtual documents created by DIFF, and suppose that *after* creating these virtual documents we sort them by size in decreasing order. This means that the largest virtual document now corresponds to the first bit of each bit vector, and that, when we look at all bit vectors for a document, the first bit has the largest number of 1 values, the second bit the second largest, and so on. In other words, we have transformed the bit vectors into "front heavy" bit vectors where the initial bits are more likely to be set to 1. This induces a clustering effect in the resulting gaps that is then automatically exploited by IPC, PFD, or possibly other methods. We refer to this method as 2R-DIFF (where R stands for reordering).

We note here that it is not clear that sorting by virtual document size is the best rule for reordering. In fact, it can be argued that it is desirable to place as close to each other those virtual documents that have a lot of terms in common. This can be modeled by building a graph on the virtual documents, with edges weighted by the size of the intersection, and then running a TSP-like computation on this graph. We experimented with this approach, but were unable to obtain any benefits over simple sorting, and thus we use sorting in all subsequent results.

The same reordering trick can also be applied to two-level MSA, by sorting its virtual documents by size; we call the resulting method 2R-MSA. We note that this reordering approach for MSA is very different from the reordering considered in the original MSA paper [13], in that we reorder virtual documents while [13] reorders the original versions.

Another algorithm can be obtained by again reordering the bit vectors for DIFF, and then using the hierarchical Huffman-based coding scheme from the HUFF method in [11] directly on the bit vectors, rather than coding gaps using IPC or PFD. We call this method 2R-HUFF. Finally, these reorderings have to be inverted during query processing.

5.3 Hybrid Methods

Our third optimization involves a hybrid between DIFF and MSA. Recall that MSA decreases the number of postings compared to DIFF, at a blow-up in the docID space. MSA performs best when a few virtual documents contain most of the postings, and does less well when there are a large number of small but non-empty virtual documents. This suggests a hybrid approach where we pick all the large virtual documents in MSA, remove the corresponding postings from the versions, and use DIFF to finish up the rest of the postings.

We refer to this method as 2-Hybrid for the basic two-level method, and 2R-Hybrid when used with reordering. One question is to decide how many of the large virtual documents we should pick using MSA, and when we should stop and mop up the remaining postings using DIFF. We found that selecting all virtual documents above a fixed size works reasonably well.

5.4 Integrating Frequencies

All the described methods can be extended to store frequency values, using ideas similar to those in [11] for the basic versions of DIFF and MSA. During the reordering step, we now sort by a slightly different weight function that takes into account that the virtual documents are bags (multi sets) rather than sets. In the case of 2R-HUFF, as in the HUFF method in [11], we have a choice between keeping docIDs and frequencies separate and integrating them into one vector for better compression. Details are omitted due to space constraints, but we provide experimental results for the frequency case further below.

5.5 Query Processing

We now outline the changes in query processing that are required when using our new techniques. We assume here a standard DAAT type query processor that implements ranked queries on top of a Boolean filter such as an AND or OR of the query terms. Here, we focus on the case of AND. Of course, real queries on archival collections may involve additional features such as restrictions to certain time ranges.

For the two baseline methods, 2-DIFF and 2-MSA, we first traverse the first-level index, and any data from the second-level is only fetched once the Boolean filter has found a docID in the intersection of the first-level inverted lists of the query terms. Recall that the second-level postings are blocked, such that each block of 128 integers can be independently accessed and decompressed. To retrieve the second-level data for a particular document and term, we may need to decompress more than one block if the data goes across block boundaries.

Bit vectors are recreated from compressed data by initializing a vector to zero, and then setting corresponding bits as we decompress the gap values. Afterwards, the reordering needs to be inverted in the cases of 2R-DIFF, or 2R-Hybrid. Finally, we process the information in the recovered bit vector as required in DIFF, or the hybrid. In the case of DIFF we need to loop over the bit vector and apply any changes (recall that DIFF indexes these changes); we found that this step can be accelerated significantly by using a lookup table of size 512 that converts one byte of the bit vector at a time.

5.6 Experimental Results

We now present our experimental results on the Wiki and Ireland data sets.

One versus two levels: In Table 2 we show the sizes of the first-level indexes for Wiki and Ireland under IPC and PFD. As we see, IPC performs better than PFD on this part of the index. Next, in Table 3 we compare the one-level and basic two-level methods for DIFF and MSA in terms of index size. For the first level in the two-level methods, we use IPC, while for the second level, we show results for IPC and PFD. We see that the two-level methods consistently

outperform their one-level counterparts in terms of index size, with particularly large improvements when using PFD.

Thus, even without the reordering technique, we see decent improvements over the one-level versions of DIFF and MSA evaluated in [11]. To directly compare the results to those for the combinatorial approach in Table 1, we need to deduct the size of the first-level structure (73 MB for Wiki and 273 MB for Ireland) from the numbers in Table 3; we see that there is still a gap between the methods.

	Wiki	Ireland
IPC	73	273
PFD	114	394

Table 2: Compressed size of the first-level index.

.

	One Level		Two Level	
	Wiki	Ireland	Wiki	Ireland
DIFF-IPC	269	751	253	652
DIFF-PFD	323	927	269	757
MSA-IPC	237	682	233	663
MSA-PFD	287	799	243	672

Table 3: Compressed index sizes for the one-level and two-level DIFF and MSA approaches with IPC and PFD. In all cases, the first-level index is compressed using IPC.

Results for docID indexes: In Table 4, we look at the compressed size for index structures with docIDs but no frequency values. First, we see improvements in index size over the results in Table 3 from use of the reordering technique. After accounting for the size of the first-level index, many of the numbers are now in the range of numbers that we saw from the bitwise approaches in the previous section. The best result on Wiki is obtained by 2R-HUFF, while for Ireland 2R-Hybrid-IPC performs best. Note that, for 2R-DIFF, 2R-HUFF, and 2R-Hybrid, the size of a global table to reverse the bitvector is included in the compressed size.

In general, improvements are more limited for Ireland, and this is probably due to the smaller number of versions per document: Basically, shorter bit vectors offer less structure that can be exploited for better compression.

	Wiki	Ireland
HUFF	213	577
2R-DIFF-IPC	176	548
2R-DIFF-PFD	193	596
2R-MSA-IPC	179	563
2R-MSA-PFD	195	659
2R-Hybrid-IPC	166	520
2R-Hybrid-PFD	182	572
2R-HUFF	161	542

Table 4: Compressed index size in MB for methods with reordering on the Wiki and Ireland data set. The first-level index is compressed using IPC and is included in the total size. The HUFF method in the first line is the best previous method from [11].

Results for docIDs and frequencies: In Table 5 we show results for index structures that contain docIDs as well as frequencies. This means that for MSA and DIFF we also change the way virtual documents are defined, resulting in increases in the size of the docID data itself. For 2R-HUFF, we have a choice between keeping docIDs and frequencies separate, or integrating them into one vector for better compression. We again see very significant improvements over the best previous approach, the HUFF method from [11].

Tuning the hybrid: In Figure 5 we show the compressed size of a docID-only index as we vary the cut-off between using MSA and DIFF. We see that for Wiki, it is best to use all virtual documents in MSA with size at least 20, while for Ireland, virtual documents with sizes as small as 10 should be selected. We note that slight additional improvements might be possible by further tuning of this policy.

	Wiki			Ireland		
	docID	freq	total	docID	freq	total
HUFF	N/A	N/A	336	N/A	N/A	645
2R-DIFF-IPC	214	86	300	564	215	779
2R-DIFF-PFD	242	98	340	623	240	863
2R-MSA-IPC	197	41	238	597	154	751
2R-MSA-PFD	212	51	263	658	170	828
2R-HUFF	210	64	274	568	236	804
2R-HUFF combined	N/A	N/A	235	N/A	N/A	619

Table 5: Compressed sizes of indexes with docID and frequencies on Wiki and Ireland. The HUFF method in the first line is the best previous method from [11].

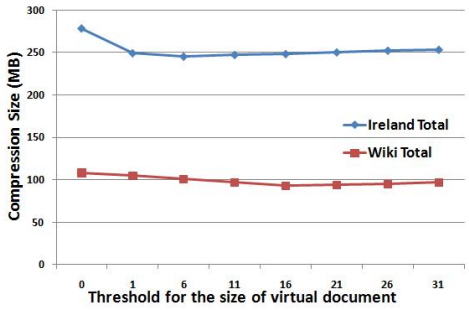

Figure 5: Selecting the best cut-off point between MSA and DIFF in 2R-HYBRID.

Query processing performance: We have observed great improvements in index size, but these improvements would be futile if query processing performance is degraded due to the various complicated ordering and hybrid strategies. We now show that this is not a problem, and that in fact our methods are even faster than the fastest previous method, the Sorted algorithm in [11].

In Table 6, we show query processing cost in milliseconds per query on the Wiki data for a few of our new two-level techniques and for the Sorted algorithm from [11] that was previously the fastest method. For each technique we show the query processing speed, index size, and number of decoded postings per query, using IPC, PFD, and a mixed index with IPC on the first level and PFD on the second level.

Our query processor assumes that the entire index is kept in main memory, and thus there are no disk access costs. Following the approach in [11], we randomly selected 10000 queries from a large trace of AOL queries, considering all queries that resulted in visits to Wikipedia. As we see from Table 6, our new two-level methods are in fact all faster than the fastest previous method! The fastest method is 2-DIFF, which runs in 0.66 millisecond per query using PFD. Adding the reordering only increases the time to 0.83 milliseconds. Note that for 2R-MSA, the query processing time is the same as 2-MSA. Because there is no reversed operation during query processing. For 2R-Hybrid, we were unable to get results in time for the final version, but we expect running times very close to the methods without reordering. 2R-HUFF, on the other hand (not shown), runs slower than the other methods, due to the use of Huffman coding, and takes more than 2ms.

We also note that using IPC instead of PFD in the first level of the index, but keeping PFD in the second level, only marginally increases query processing costs, while giving a decent reduction in index size. The new two-level methods obtain much of their speed gains by decoding many fewer postings than Sorted. In fact, this is not unexpected and was the main motivation for the introduction of two-level

methods in [1]. In summary, our results show decent and simultaneous improvements in index size and query processing speed over all previous approaches.

	speed (ms/query)			index size (MB)			decoded
	IPC	PFD	Mix	IPC	PFD	Mix	postings
Sorted	4.28	0.97	-	570	583	-	245
2-DIFF	1.34	0.66	0.68	253	294	269	34
2R-DIFF	1.51	0.83	0.85	176	217	193	34
2-MSA	1.68	0.99	1.01	233	274	243	28
2R-MSA	1.68	0.99	1.01	179	220	195	28
2-Hybrid	1.59	0.88	0.9	166	207	182	30

Table 6: Query processing cost, index size, and number of decoded docIDs on Wiki using various algorithms and compression methods. Mix is the case where IPC is used on the first level and PFD on the second level.

6. CONCLUSIONS

In this paper, we have studied index organization and compression techniques for versioned document collections. In particular, we analyzed typical properties of versioned document collections that lead to succinct index structures, and then derived combinatorial upper and lower bounds for index size. Our main contribution are new index organization and compression schemes based on the DIFF [2] and MSA [13] approaches that achieve significant improvements in both index size and query processing speed.

Our improvements are based on exploiting the properties of the bit vectors in the second level of the index. The problem of better modeling these bit vectors is closely related to text evolution and user edit behavior in versioned collections. It would be very interesting to come up with simple mathematical models for these bit vectors. In particular, it would be nice to have generative models for user behavior that explain the patterns we observe in these bit vectors. We would expect such models to be different in multi-author environments such as Wikipedia where multiple authors edit the same document, and collections such as the Internet Archive where most pages are maintained by a single person or organization. While there is a lot of recent work on the structure of Wikipedia, we have not seen a good model to explain the overall evolution of text and thus the coming and going of terms in documents.

Other interesting research problems are improved positional index structures for versioned collections, and optimizing query processing performance in versioned collections for different classes of queries.

Acknowledgments

This research was supported by NSF Grant IIS-0803605, "Efficient and Effective Search Services over Archival Webs", and by a grant from Google. We also thank the Internet Archive for providing access to the Ireland data set.

7. REFERENCES

[1] I. Altingovde, E. Demir, F. Can, and O. Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Trans. on Information Systems*, 26(3), June 2008.

[2] P. G. Anick and R. A. Flynn. Versioning a full-text information retrieval system. In *Proc. of the ACM SIGIR Conf.*, 1992.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9), 1975.

[4] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A time machine for text search. In *Proc. of the ACM SIGIR Conf.*, 2007.

[5] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result-caching. In *Proc. of the Workshop on P2P Systems*, 2003.

[6] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. of the ECIR Conf.*, 2005.

[7] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. of the DCC Conf.*, 2002.

[8] 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 EDBT Conf.*, 2006.

[9] 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.

[10] A. S. Fraenkel, S. T. Klein, Y. Choueka, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. of the ACM SIGIR Conf.*, 1986.

[11] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *Proc. of the ACM CIKM Conf.*, 2009.

[12] S. Heman. Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica, 2005.

[13] M. Herscovici, R. Lempel, and S. Yogev. Efficient indexing of versioned document sequences. In *Proc. of the ECIR Conf.*, 2007.

[14] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31(2), 1987.

[15] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. In *Proc. of the USENIX Annual Technical Conf.*, 2004.

[16] H. Leong, N. Mamoulis, K. Berberich, and S. Bedathur. Durable top-k search in document archives. In *Proc. of the ACM SIGMOD Conf.*, 2010.

[17] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3, 2000.

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

[19] A. Orlitsky. Interactive communication of balanced distributions and of correlated files. *SIAM J. of Discrete Math*, 6(4), 1993.

[20] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. *Proc. of the Int. Conf. on Very Large Data Bases*, 1997.

[21] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. of the First USENIX FAST Conf.*, 2002.

[22] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the ACM SIGMOD Conf.*, 2003.

[23] T. Schwarz, R. Bowdidge, and W. Burkhard. Low cost comparison of file copies. In *Proc. of the Int. Conf. on Distributed Computing Systems*, 1990.

[24] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Inf. Processing and Management*, 39(1), 2003.

[25] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of the ECIR Conf.*, 2007.

[26] F. Silvestri, S. Orlando, and R. Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proc. of the ACM SIGIR Conf.*, 2004.

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

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

[29] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. of 18th WWW Conf.*, 2009.

[30] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of 17th WWW Conf.*, 2008.

[31] J. Zhang and T. Suel. Efficient search in large textual collection with redundancy. In *Proc. of 16th WWW Conf.*, 2007.

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

[33] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. of the ICDE Conf.*, 2006.