

Compact Full-Text Indexing of Versioned Document Collections

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

Hao Yan
CSE Department
Polytechnic Institute of NYU
Brooklyn, NY, 11201
hyan@cis.poly.edu

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

ABSTRACT

We study the problem of creating highly compressed full-text index structures for versioned document collections, that is, collections that contain multiple versions of each document. Important examples of such collections are Wikipedia or the web page archive maintained by the Internet Archive. A straightforward indexing approach would simply treat each document version as a separate document, such that index size scales linearly with the number of versions. However, several authors have recently studied approaches that exploit the significant similarities between different versions of the same document to obtain much smaller index sizes.

In this paper, we propose new techniques for organizing and compressing inverted index structures for such collections. We also perform a detailed experimental comparison of new techniques and the existing techniques in the literature. Our results on an archive of the English version of Wikipedia, and on a subset of the Internet Archive collection, show significant benefits over previous approaches.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Inverted index, search engines, inverted index compression, versioned documents, web archives, wikipedia.

1. INTRODUCTION

Over the last decade, web search engines and other information retrieval tools have become the primary means of locating relevant information for millions of users. The largest search engines, built from thousands of machines, are able to

answer tens of thousands of queries per second on billions of web pages with interactive response times. 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, they continuously retrieve new pages and update their inverted index to reflect the new data. If a page was previously crawled, then the engine typically replaces the old with the new version, and any information contained only in the older version becomes unavailable.

While web searchers are primarily interested in current pages, there are other scenarios where a search over previous versions would be of interest. As one important example, the Internet Archive has collected more than 150 billion web pages since 1996 in an attempt to archive and preserve the information available on the web. Wikipedia maintains a complete history of all versions of every article, and it would be useful to be able to search across the different versions. Other scenarios are search in revision control and document management systems (e.g., searching in source code or across different drafts of documents), and indexing support for versioning file systems.

In this paper, we study the problem of creating highly compressed full-text index structures for versioned document collections, that is, collections that can contain multiple versions of each document. The main challenge is how to exploit the significant similarity that often exists between different versions of the same document to avoid a blowup in index size. That is, a collection containing say 50 versions of each document should not result in an index of 50 times the size of a single-version index. This problem has received some limited attention in the research community [2, 9, 3, 15, 31], but many aspects are still unexplored.

We note that this problem is related to, but different from, the problem of exploiting similarities between distinct documents in a collection for better index compression [7, 24, 6, 25, 29, 31, 1]. For example, different pages from the same web site [10], or from different sites but about the same topic, often share some degree of similarity. When exploiting similarities between distinct documents, one major challenge is to identify which documents are similar to each other, and the observed similarities are often more limited than in the case of versions, thus resulting in more limited gains in compression. In a versioned document collection, on the other hand, we typically already know which versions belong to a common document, and the challenge is to model the change between consecutive versions such that sequences of possibly many versions with often only minor changes can be efficiently indexed. Nonetheless, techniques

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'09, November 2–6, 2009, Hong Kong, China.

Copyright 2009 ACM 978-1-60558-512-3/09/11 ...\$10.00.

for one problem are also often useful for the other, and we discuss both cases in more detail in the next section.

Thus, in this paper we focus on versioned document collections. We are given a set of documents d_0, \dots, d_{n-1} and for each document d_i a sequence of versions v_0, \dots, v_{m_i-1} . 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 also limit ourselves to non-positional indexes, that is, index structures that store document ID and frequency data, but not within-document position information. As discussed later, the case of positional indexes tends to require somewhat different approaches than the non-positional case, since a single insertion or deletion of a word at the beginning of the document affects the position information of every posting in the document.

Our main contributions in this paper are new techniques for organizing and compressing inverted index structures for versioned document collections. We also perform an experimental study that compares known techniques in terms of compressed size and resulting query processing speed, using data from Wikipedia and the Internet Archive. The approach achieving the smallest size is based on a two-level index structure where index compression reduces to modeling and compressing *version bitvectors* that, for a given term t and document d , specify which versions of d contain t . This approach also integrates well with existing IR query processors, and should allow for future improvements as we find better models for document change.

The remainder of this paper is organized as follows. In the next section, we provide some technical background and discuss related work. In Section 3 we describe techniques for docID compression, while Section 4 considers compression of frequency information and Section 5 discusses query processing. Section 6 gives our experimental results. Finally, Section 7 provides concluding remarks.

2. BACKGROUND AND RELATED WORK

In this section, we first give some background on inverted indexes and index compression techniques. We then discuss related work, which falls into three categories: Indexing of versioned document collections, index compression techniques that exploit similarity between different documents, and redundancy elimination in storage systems.

2.1 Indexing and Index Compression

Most current search engines use an *inverted index* structure to support efficient keyword queries [32]. Informally, an *inverted index* for a collection of documents is a structure that stores, for each term (word) occurring somewhere in the collection, information about all locations where it occurs. In particular, 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 about the locations of the occurrences within the document and their contexts. In this paper, we assume that postings have docIDs and frequencies, but we do not consider the case of a *positional index* where within-document positions (i.e., where in the document the term occurs) are stored in the postings.

The postings in each list are usually sorted by docID and

then compressed using any of a number of index compression techniques from the literature [32]. Most of these 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. Of course, these values have to be summed up again during decompression, but this can be done very efficiently. We cannot take gaps of frequencies as they are not in sorted order, but most frequency values are already very small. (In addition, we may deduct 1 from each d-gap and frequency, so that the integers to be compressed are non-negative but do include 0 values.) The d-gaps and frequencies are often stored separately, and thus we compress sequences of d-gaps and sequences of 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 the following, we focus on two techniques that we directly employ in this paper, *Interpolative Coding* (IPC) and *PForDelta* (PFD).

Interpolative Coding is a coding technique introduced in [18] that was designed for the case of clustered or bursty term occurrences, such that those encountered in longer linear texts (e.g., books). IPC has also recently been shown to perform very well in scenarios where docIDs are assigned to documents based on textual similarity [7, 24, 25, 5, 26, 29], as discussed in the next subsection.

The idea in IPC is best described using docIDs rather than d-gaps. Given a set of docIDs $d_i < d_{i+1} < \dots < d_j$ where $l < d_i$ and $d_j < r$ for some bounding values l and r known to the decoder, IPC first encodes d_m where $m = (i + j)/2$, and then recursively compresses the docIDs d_i, \dots, d_{m-1} using l and d_m as bounding values, and then the docIDs d_{m+1}, \dots, d_j using d_m and r as bounding values. Thus, we compress the docID in the center, and then recursively the left and right half of the sequence. To encode d_m , observe that $d_m > l + m - i$ (since there are $m - i$ values d_i, \dots, d_{m-1} between it and l) and $d_m < r - (j - m)$ (since there are $j - m$ values d_{m+1}, \dots, d_j between it and r). Thus, it suffices to encode an integer in the range $[0, x]$ where $x = r - l - j + i - 2$ that is then added to $l + m - i + 1$ during decoding; this can be done trivially in (at most) $\lceil \log_2(x + 1) \rceil$ bits, since the decoder knows the value of x .

While IPC achieves very good compression, it is somewhat slow, with decompression rates of less than 100 million integers per second on current CPU cores. Note that we can also use IPC to compress frequency values, by first performing a prefix sum on the frequencies and then compressing the resulting values [18].

PForDelta is a family of compression schemes first introduced in [14, 33], and further optimized in [30, 29], that allows extremely fast decompression while also achieving a small compressed size. PFD first determines a value b such that most of the values to be encoded (say, 90%) are less than 2^b and thus fit into a fixed bit field of b bits each. The remaining values, called *exceptions*, are coded separately. If we apply PFD to blocks containing some multiple of 32 values, then decompression involves extracting groups of 32 b -bit values, and finally patching the result by decoding a smaller number of exceptions. This process can be imple-

mented extremely efficiently by providing, for each value of b , an optimized method for extracting 32 b -bit values from b memory words. Versions of PFD have been shown to achieve decompression rates beyond a billion integers per second on a single core of a CPU.

In this paper, we use a version of PFD called OPT-PFD described in [29] that differs from those in [14, 33, 30] in two ways: (1) Instead of organizing exceptions in a linked list, the lowest b bits of each exception are stored inside the bit field, and then the higher-order bits and the offsets of the exceptions are separately encoded using another compression method called S16. (2) Instead of choosing a b that guarantees less than 10% exceptions in the block, we select the b that results in the smallest compressed size for the block. As shown in [29], OPT-PFD achieves very good compression on clustered data, in some cases coming close to IPC in terms of compressed size, while still maintaining decompression rates close to a billion integers per second.

Bitvector Compression: Inverted lists can also be modeled and compressed by treating them as bitvectors. In [11], a hierarchical scheme is proposed for compressing such sparse bitvectors. We will later adapt this scheme to our scenario of a versioned document collection.

2.2 Indexing Versioned Document Collections

While the general problem of compressing inverted index structures has been extensively studied over the last 20 years, very few of these results have addressed the case of versioned document collections. Obviously, a trivial solution could treat every version as a separate document, but this would result in a very large and highly redundant index structure. We now discuss previous work that improves on this solution, in particular [2, 9, 3, 15, 31]. In the following, we assume a versioned document collection that consists of a set of documents d_0, \dots, d_{n-1} , and for each document d_i a sequence of versions v_0, \dots, v_{m_i-1} .

Positional Indexes: The basic idea underlying the approaches in [9, 31] is to avoid repeated indexing of substrings that are common to several versions of a document. To achieve this, [9, 31] both propose to partition each version into a number of fragments and to then index these fragments. During query processing, any matches from these fragments then have to be joined again to decide which versions contain the query words. However, the way documents are partitioned is different in the two approaches. In [9], content is organized in a tree structure, where each node has some private and some shared content, and each node inherits its ancestors' shared content. In contrast, [31] uses content-dependent string partitioning techniques [22, 16, 27] to split each document into fragments; one advantage of this approach is that it allows for highly efficient updates as new versions are added to the system. Both approaches give significant reductions in index size, and are particularly suited for positional index structures that also store within-document position data.

Non-Positional Indexes: The approaches in [2, 3, 15] consider the case of a non-positional index, which is also our focus in this paper. We note here a basic difference between the approaches for 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 starting position 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 approach performs much better than the substring-oriented models. Thus, we focus on set- and bag-oriented approaches in this paper.

The work in [2] proposes to deal with document versions by indexing the last version as usual, and then adding *delta postings* to store changes from the last to previous versions. We note that the DIFF algorithm we describe and evaluate later is a variant of this earlier approach.

The approach taken in [15] attempts to find subsets of terms that are contained in many versions of a document. Such subsets are then treated and indexed as virtual documents, and the main challenge is to minimize the overall number and size of the virtual documents. Assume that W is the set of all terms appearing anywhere in the document versions. Then [15] first builds a matrix M , where each column represents a term in W , while each row represents a version. In most cases, the versions are ordered by time, though in the general case better results could be achieved with another ordering. The binary value (1 or 0) of entry $M_{i,j}$ indicates whether the i -th version contains the j -th term. Then each virtual document $V_{i,j}$ is composed of all terms whose corresponding columns have a maximal run of 1s that starts at row i and ends at row j . In [15], the matrix M is called an exclusive alignment of these versions, and the problem of how to optimally order the versions in the matrix is the Multiple Sequence Alignment (MSA) problem. We thus refer to the method in [15] as the MSA algorithm.

Another approach [3] for the non-positional case 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. Note that this is a lossy compression technique that uses an index with precomputed quantized scores, and thus it is not directly comparable to our approaches in terms of compressed size. We will however describe an ordering-based approach (also briefly mentioned in the next subsection) that can be seen as an efficient non-lossy counterpart to the work in [3].

2.3 Succinct Indexing of Similar Documents

There have been a number of studies that attempt to 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 two important ways: First, similarities between distinct documents are typically more limited, leading to more limited gains, and second, an important part of the problem is to identify which documents in the collection are similar, while in versioned collections this information is more or less implied.

The most common approach to indexing similar documents is based on the idea of reordering the documents in the inverted lists, by assigning consecutive or close-by docIDs to documents with high similarity [7, 24, 26, 5, 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 proposed techniques for reordering documents can be grouped into three classes, (i) top-down techniques that partition the collection into clusters of similar pages [7, 26, 5], (ii) bottom-up techniques that construct the ordering from the ground up [24, 26, 5], and

(iii) heuristics that assign docIDs via sorted URL order [25, 29]. We note in particular that one of our baseline methods in Section 3 is an adaptation of the reordering approach where we assign consecutive docIDs to consecutive versions of the same document.

Another related approach was studied in [1], where similar documents are clustered into disjoint groups and then indexed using a two-level index structure. The first level of the index essentially indexes the union of all documents in a group, while the second level of the index uses Rice coding to specify which of the documents in the group contain the term. We adapt this approach to the case of versioned document collections, but with various additional optimizations that significantly improve compression. Finally, several of the techniques for versioned collections, for example [31], can also be used to exploit similarities across document boundaries.

2.4 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. In fact, many of these efforts preceded the recent work on eliminating redundancy in index structures discussed by us here. This includes the Low Bandwidth File System [19], various storage and backup systems [12, 4, 17, 21, 27], and remote file synchronization techniques [28, 23, 20]. Also relevant are techniques for efficiently identifying duplicated content across large collections using hashing [8, 13, 22, 16, 27], which are in fact used by several of the indexing techniques we have discussed.

3. VERSIONED DOCID COMPRESSION

In this section, we describe algorithms for compressing inverted lists consisting of docIDs only, while the next section will describe how to add frequencies to the index. We first describe two trivial baseline methods that treat versions as separate documents but compress them under different orderings. We then briefly discuss the MSA method in [15], and an algorithm called DIFF that indexes the difference between consecutive versions and that is a variant of the approach described in [2]. Our final, new, algorithm combines a two-level index structure with Huffman-based hierarchical bitvector compression.

3.1 Baseline Methods

As discussed, a straightforward method to compress versioned documents is to treat each version as a separate document and then index these documents using standard techniques. A slightly smarter approach makes sure to sort the versions by document name and version number, thus assigning consecutive docIDs to consecutive versions of the same document. This is of course also the idea underlying the methods discussed in Subsection 2.3, and in particular the sorting-based approach in [25, 29], but it can also be seen as a simple non-lossy counterpart of the idea of merging similar postings in [3]. We are not aware of a previous evaluation of this approach, which seems very natural.

Note that after sorting, the resulting lists of docIDs are likely to be extremely clustered, with many long runs of consecutive docIDs. To compress such lists efficiently, it is important to use compression techniques that work well on clustered data. Interpolative Coding (IPC) is known to

do well in this case, and recent work in [29] showed that a version of PForDelta called OPT-PFD also does very well. Thus, we use these two compression methods in combination with the sorted and unsorted baseline case. We will use the name *Random* to refer to the unsorted case where we assign docIDs to document versions at random, and *Sorted* to refer to the method with sorted assignment. We attach the postfix IPC or PFD to each method to specify which compression method was applied to the resulting index.

3.2 MSA

Another approach we will compare to is the solution based on the Multiple Sequence Alignment problem in [15], which we already described in Subsection 2.2. While the general version of this problem is NP-hard, [15] also provides a greedy polynomial-time algorithm that is shown to be optimal under assumptions that are very likely to be true in our scenario. Under this simplified approach, given a document d with a number of versions, MSA basically defines the virtual document $V_{i,j}$ for d as the set of all terms that occur in versions i through j , but not in versions $i - 1$ or $j + 1$. Note that in principle, the number of virtual documents is quadratic in the number of versions of a document. In practice, most of the virtual documents are usually empty, and can be discarded.

Thus, the MSA approach generates a number of virtual documents, which we then index using either IPC or PFD. We note that some changes in query processing are needed for this approach, as described in [15], and some auxiliary information about the virtual documents also needs to be stored in memory, such as which of the virtual documents are non-empty.

3.3 Indexing Differences

This main idea of this method, which is a variant of the approach in [2], is to directly index the difference between consecutive versions. In particular, for every pair of consecutive versions v_i and v_{i+1} we create a new document that is the symmetric difference between v_i and v_{i+1} . Thus, this new document contains a term t if and only if the term is either added or removed from the document between versions v_i and v_{i+1} . We also keep for every document the first version, which is of course the same as the symmetric difference between the empty document and the first version. (We could also keep the last version instead, as in [2].)

We then index the resulting new virtual documents using either IPC or PFD. This approach also requires some changes during the query processing phase, but we will show later that this can in fact be done in an elegant manner and with little loss in efficiency by adapting a standard DAAT query processor. We refer to this algorithm as DIFF.

3.4 Two-Level Index Compression

The above methods either treat versions as separate documents, or construct artificial new documents (virtual documents) from the versions. Our next method instead models a versioned collection on two levels, a document level where each distinct document is identified by a global docID, and a version level where we specify which of the versions actually contain a given term. Then special techniques are applied to do a good compression at the version level, while standard techniques can be used for the document level.

This idea is motivated by the recent work in [1], which

proposes to cluster a collection into groups of similar documents and then build a two-level index structure. Each group receives a group ID and can be represented by the union of the terms contained in the group documents. The method then creates the first level of the index by indexing these groups using standard techniques. Thus, a group ID will appear in an inverted list in the first level if and only if at least one of its documents contains the term. For the second level, [1] proposes to use Rice coding with a local choice of the base of the code to decrease the size. The main emphasis in [1] is not on size reduction, which is only moderate, but on improved query processing speed due to being able to skip entire groups of documents. This is also closely related to the recent observation in [29] that proper ordering of documents can increase query processing speed by allowing more forward skips in the inverted lists.

We adapt and extend this idea to the case of a versioned collection. Thus, we have a first-level index where each document is identified by a document ID, and where a posting exists in an inverted list if the term exists in at least one version of the document. This first level is compressed using any standard techniques, in our case either IPC or PFD. At the second level, the version level, instead of applying Rice coding (which does not give a lot of benefits), we model each document by a bitvector. More precisely, for any posting on the first level of an inverted list for a term t , we have a bitvector on the second level. The number of bits in the bitvector is equal to the number of versions of the document, and bit i is set to 1 iff version v_i of the document contains t . The main problem is now how to best represent and compress such bitvectors.

First, we note that keeping the bitvectors in raw uncompressed form is not a good idea in most cases. In one of the data sets we use, Wikipedia, the average document (article) has about 35 versions, and in fact many documents have several hundred versions. Another approach is to treat these vectors like frequencies and apply IPC after an MLN transformation [29], or to simply Huffman encode entire bitvectors, but experiments showed that these approaches do not work well. In particular, direct Huffman coding of entire bitvectors suffers because while most vectors have a fairly simple structure with only a few changes between 0 and 1, many of the longer bitvectors are still unique as those few changes may occur in different places.

After some study, we adapted a hierarchical approach based on Huffman coding that was proposed in [11]. We first choose a bitvector block size b , say $b = 10$. We then partition the bitvector into m/b blocks of b bits each. At the next higher level, we represent each block of b bits by a single bit that is the OR of all the bits in the block. We then again group these bits into blocks of b bits, and continue until there are fewer than b bits at the highest level. Thus, for $b = 10$ a bitvector consisting of 400 bits will result in 40 blocks of 10 bits at the lowest level, 4 blocks of 10 bits at the next higher level, and a single block with 4 bits (padded to 10 bits) at the highest level of the second-level index. (In addition, there is also a posting in the first-level index for this bitvector.) This can be improved by observing that for any bit that is 0, we do not have to store the corresponding block at the next lower level, since all bits in that block must be 0. Finally, we compress each bit block using Huffman coding. As we will show, this performs very well compared to the other methods. Some additional space

is also needed to store Huffman code tables, but this is a fairly small amount if b is chosen appropriately. We refer to this algorithm as HUFF.

We note here that there are other ways to model and compress the bitvectors at the second level, which we are currently studying. In general, the structure of these bitvectors depends heavily on the properties of typical edit behavior in versioned collections, and a better understanding of such behavior through mining of Wikipedia or the Internet Archive should result in improved compression.

4. COMPRESSION OF FREQUENCIES

We now discuss how to add frequency values to the index. In general, frequency values tend to be smaller than d -gap values, and should thus be more compressible. However, they can create some problems in the context of clustered data if not properly handled. Consider for example the case of a document where all versions contain a term either 5 or 6 times. The sorted baseline algorithm from the previous section would do very well compressing the docIDs using either IPC or PFD, but standard techniques for frequency compression would not do so well. To handle such cases, recent work in [29] proposed a transformation called MLN (Most Likely Next) that is applied to the frequency values and that decreases the average value that has to be encoded. In a nutshell, the idea in MLN is to precompute for each inverted list a small table that stores for any value (below a threshold) which value is most likely to follow immediately afterwards. Then if a value x is the j -th most likely value to follow another value y , we replace x by j (or $j - 1$ if our coder allows 0 values); see [29] for more details. We will also apply this approach here in several cases.

In particular, for the baseline algorithms *Random* and *Sorted* that treat each version as a separate document, we compress frequencies as in normal indexes but with MLN applied before using IPC or PFD. (For *Random* the benefits of MLN are very minor.) Note that for MSA and DIFF, we can also mix different methods, by applying MSA or DIFF to docIDs but compressing frequencies using the *Sorted* approach, and we experiment with this later. However, better compression can be achieved by extending the methods to directly support frequency values.

4.1 MSA with Frequencies

Recall that in MSA [15] we construct virtual documents by identifying groups of terms that exist in many consecutive versions. The work in [15] does not discuss how to deal with frequencies, so we give a short description here. The key idea to extending this approach to frequencies is to imagine treating multiple occurrences of the same term that appear and disappear in different versions of the document as essentially different terms. Thus, another occurrences of a term such as “panda” appears in a later version, we model it as a separate term “panda2”. When one of the occurrences is removed, we assume that “panda2” is always removed first (since we are in a set-based model, the actual order does not impact correctness).

We can then run the greedy algorithm for creating virtual documents in [15] as before, treating multiple occurrences as distinct terms, and arriving at a somewhat different set of virtual documents. Note that this not only adds frequency values to the postings, but also increases the total number of postings in the virtual documents, thus increasing the cost

of the docID part of the index as well. (Of course, the final index does not distinguish between “panda” and “panda2”; these are only used to create the virtual documents.)

4.2 DIFF with Frequencies

To add frequency values to the DIFF algorithm, we need to define the difference between consecutive versions based on bags rather than sets. Thus, we define the *bag difference* between consecutive versions where every term in the difference has a frequency value and a sign that indicates whether the frequency of the term is increased or decreased by the given value. (Alternatively, we can also store the raw frequency value whenever the frequency changes, and then use MLN followed by IPC or PFD on these, but this performed worse in experiments.)

Thus, in the docID part of the inverted list, we store any docID where the frequency value of the term either increases or decreases between versions. In the frequency part of the list, we store the amount of change plus one extra bit for the sign of the change (except in cases where the sign is obvious because the frequency would otherwise turn negative). As in MSA, this also increases the total number of postings in the virtual documents, and thus the size of the docID part of the index.

4.3 HUFF with Frequencies

Recall that in the HUFF method, we use a bitvector for the second level of the index to indicate which versions contain the indexed term. To add frequencies, we have two options: We can either add an additional array of frequency values, with one value for each 1 in the version bitvector. Each such array could then be compressed as before by splitting it recursively into blocks and applying a separate Huffman compression table to these blocks.

Or alternatively we can replace the docID bitvector by an array of frequency values, where a value of 0 means that the corresponding version does not contain the term, and then encode this array. (We may still refer to this array as a bitvector, though strictly speaking only the higher levels of the structure still contain bitvectors.) It turns out that this second approach achieves a smaller compressed size than encoding docIDs and frequencies separately, so this is our preferred approach. In either case, we apply the MLN transformation of [29] to the full value array before splitting the array into blocks and encoding.

5. QUERY PROCESSING

We now outline the changes in query processing that are needed when using the various versioned compression 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. In our presentation, we focus on the case of an AND. Of course, queries on archival collections may be different from other queries, employing for example additional restrictions in time (e.g., to search for document versions created between 1995 and 1999), but there is little published information or available query sets that could be used as guidance.

For the two baseline methods, *Random* and *Sorted*, no changes at all are needed. Both MSA and DIFF can be easily and elegantly implemented within a standard DAAT query processor by building a very thin extra layer that basically translates between the virtual docIDs used in the

inverted lists and the unique IDs assigned to each document version that are used by the upper layers of the query processor. In particular, DAAT query processors use cursors that traverse the inverted lists; by routing each call to move a cursor forward through our own intermediate layer (which itself calls the original code for cursor movement), we were able to adapt our in-house query processor with only a few lines of code.

This approach can be extended to the case of frequency values, by also appropriately rerouting requests for frequency values by the upper layers of the query processor. In the case where we combine frequency values compressed using *Sorted* with docIDs compressed using MSA or DIFF (as mentioned in Section 4), no changes in the frequency access mechanism are needed as frequencies are stored using the same version IDs used by the upper layer.

Finally, the two-level HUFF technique can be easily integrated into existing query processors, by treating the bitvectors like frequencies (or positions or context information) in standard indexing. Thus, a bitvector is only retrieved and uncompressed once the Boolean filter has found a docID in the intersection or union based on the first-level index. In addition, one could avoid retrieving the lowest level of blocks in some cases by first retrieving and intersecting the higher blocks of the second-level index, but we feel this is unlikely to give much benefit while complicating the code. One advantage of the two-level approach is that it allows us to hide details of the bitvector compression, thus allowing easy addition of further bitvector modeling and compression tricks that we hope to find in the future.

6. EXPERIMENTAL RESULTS

We first describe our experimental setup. We used two large data sets, a data set from Wikipedia, and a data set with pages from the Irish web domain provided by the Internet Archive. The Wikipedia data set consists of a full archive of English language articles from 2001 until mid 2008. It originally contained 2,401,799 documents (each of which had a number of versions) and we randomly selected 10% of the documents for our experiments. The Ireland data set is based on 1,059,670 distinct URLs from the Irish web domain that were each crawled at least 10 times during the period from 1996 to 2006, resulting in non-identical versions; these URLs were selected from a much larger set also containing many other URLs that were only crawled once or twice.

For the Wikipedia data set, we have 0.24 million documents with, on average, 35 versions each, resulting in a total of 5.82 billion postings over 4.06 million distinct terms. For the Irish data set we have 1.06 million documents with, on average, 15 versions each, resulting in a total of 3.42 billion postings over 5.95 million distinct terms. We removed 20 very common stopwords, and no stemming was performed. Note that if we simply treat each version as a separate document, we get over 8.4 million documents in the Wikipedia data and over 15 million in the Irish data. We then implemented the described compression methods and integrated them into an existing IR query processor. All experiments were performed on a single core of a 2.66GHz Intel(R) Core(TM)2 Duo CPU with 8GB of main memory.

6.1 Data Distribution

Figure 1 shows the distribution of the number of versions

per document in Wikipedia on a double-logarithmic scale. Some documents have thousands of versions, and large numbers of documents have hundreds of versions, but most documents have less than 20. On the other hand, most versions, and most postings, belong to documents with a fair number of different versions. (These are usually longer articles about more important topics that have undergone many revisions over time.)

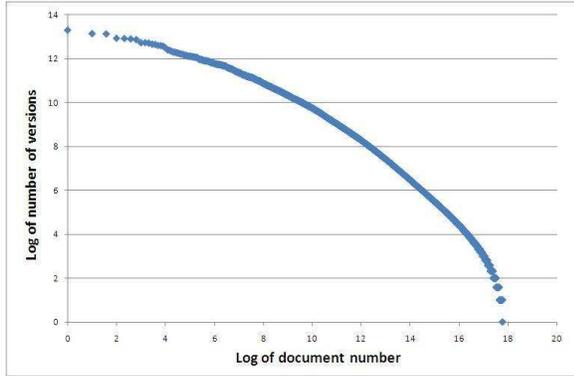


Figure 1: Distribution of the number of versions per document for Wikipedia. We sort documents from left to right by the number of versions they have, and plot both axes on a log scale with base 2.

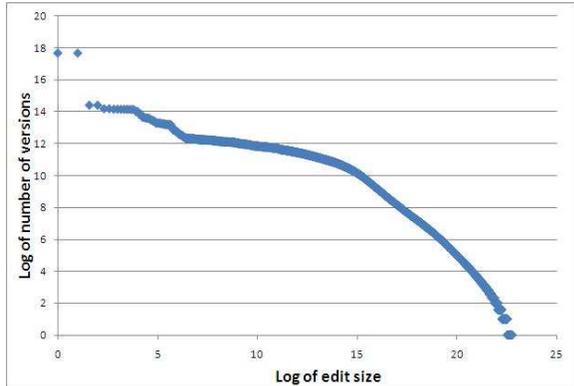


Figure 2: Distribution of edit sizes for the Wikipedia data set. We sort versions from left to right by the size of the difference from the previous version in the DIFF method, and plot both axes on a log scale with base 2.

Figure 2 shows the distribution of edit sizes for the Wikipedia data set, again on a double-logarithmic scale. Here, the size of an edit between two versions i and $i + 1$ is defined as the symmetric difference of the two sets of terms. As we see, not surprisingly most edits are small, involving only a few words that are changed. However, large edits overall make up a significant fraction of the total amount of change (total textual turnover) in the collection.

6.2 Basic Experimental Results

We first compare in Table 1 the compressed sizes of docIDs and frequencies using the two baseline methods in Subsection 3.1, *Random* and *Sorted*, which treat each version as

a separate document but use different docID assignments. We show results for the Wikipedia and Ireland data sets, using both IPC and PFD for compression. From Table 1, we see that the compression performance of *Sorted* is much better than that of *Random* in all cases. Improvements are particularly good for docIDs, while the gains for frequencies are more limited. Also, PFD comes quite close to IPC, and in some cases even does slightly better. Overall, even if no specialized compression techniques for versioned collections are used, at the least consecutive docIDs should be given to consecutive versions of a document. In the following, we compare the other techniques to the sorted case only.

	Wikipedia			Ireland		
	docID	freq	total	docID	freq	total
Random-IPC	4957	2015	6972	2908	1226	4134
Random-PFD	5499	2208	7707	3289	1167	4456
Sorted-IPC	570	986	1556	1086	952	2038
Sorted-PFD	583	896	1479	1137	856	1993

Table 1: Compressed index size in MB for the Wikipedia and Ireland data sets, using *Random* and *Sorted* with IPC and PFD compression.

	Wikipedia	Ireland
Sorted-IPC	570	1086
Sorted-PFD	583	1137
DIFF-IPC	269	751
DIFF-PFD	323	927
MSA-IPC	237	682
MSA-PFD	287	799
HUFF	213	577

Table 2: Compressed index size in MB for the two data sets when compressing docIDs only. For HUFF, IPC was used in the first-level index, and the cost of that index is of course included.

Next, in Table 2 we look at the compressed size for docIDs only for all our methods. We see that both MSA and DIFF are significantly better than *Sorted*, and that HUFF outperforms all other methods on both data sets. As before, improvements are more limited for Ireland because there are fewer versions of each document on average, resulting in less redundancy that we can exploit.

	Wikipedia			Ireland		
	docID	freq	total	docID	freq	total
Sorted-IPC	570	986	1556	1086	952	2038
Sorted-PFD	583	896	1479	1137	856	1993
DIFF-IPC	337	108	445	810	390	1200
DIFF-PFD	430	122	542	1018	397	1415
MSA-IPC	333	79	411	759	266	1025
MSA-PFD	332	92	424	887	280	1167
HUFF	213	191	404	577	240	817
HUFF combined	N/A	N/A	336	N/A	N/A	645

Table 3: Compressed index sizes when both docIDs and frequencies are compressed using the described methods. For HUFF, we use IPC for the first-level index, but we have two cases: One where frequencies are compressed as a separate value array, and one where bitvector and frequency values are combined into one array.

Table 3 shows the compressed index sizes for the two data

sets when docIDs and frequencies are integrated as explained in Section 4. This means that for MSA and DIFF we change the way virtual documents are defined, resulting in increases in the size of the docID data, and in extra cost due to frequency values. For HUFF, we consider two cases, one where we compress frequency values separately, and one where we use one array, in which case it is impossible to report separate values for docIDs and frequencies. (We chose the best block size for the HUFF method.) There is no change for the sorted method.

From Table 3, we see that MSA, DIFF, and HUFF do much better than *Sorted* when compressing frequency values, and thus it is preferable to directly implement frequency compression within those methods. On the other hand, the relative order between the methods is largely unaffected, with HUFF performing best, followed by MSA and DIFF, and with *Sorted* by another factor of 2 to 3 away in terms of total index size.

6.3 Trade-Offs

Figure 3 compares the compressed size for the two data sets using HUFF with different bitvector block sizes. (Recall that this block size is the number of versions (bits) that are stored in a single bitvector, so that a document with more versions results in more than one bitvector for each term.) We can see that for a block size around 45, HUFF achieves the best compression performance on the two data sets. Note that there is a tradeoff: If the block size is too large, then the number of distinct vectors will be greatly increased and thus we get larger Huffman tables. But if the block size is too small, then the total number of vectors increases and the cost of the Huffman codes goes up. We also note that the dependence on the block size is less pronounced for the Ireland data set since many documents in that set have fewer than 20 or 30 versions anyway, and are thus never partitioned into smaller blocks.

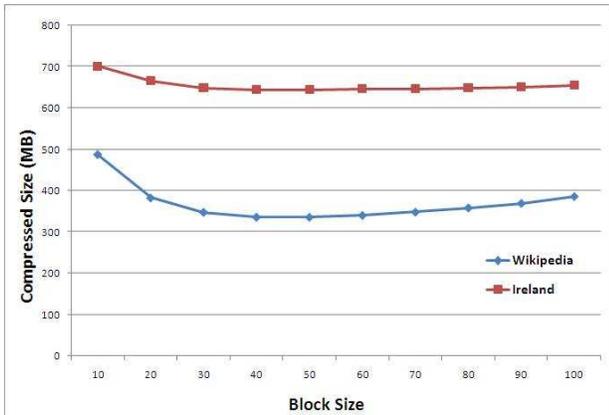


Figure 3: Compressed size for the two data sets using HUFF with different bitvector block sizes.

Next, we show in Table 4 the contributions of the different levels in the HUFF approach to the total index size. We see that the intermediate levels of the second-level only contribute a small amount to the total index size. In the case of Ireland, the first-level index is more significant because there are fewer versions per document on average.

	first-level	mid-level	lowest-level
Wikipedia	73	10	253
Ireland	273	13	359

Table 4: Contributions to the compressed sizes in MB for HUFF (combined docIDs and frequencies) with block size 45, for the first-level index, the higher levels of the second-level index (mid-level), and the lowest level of the second-level index.

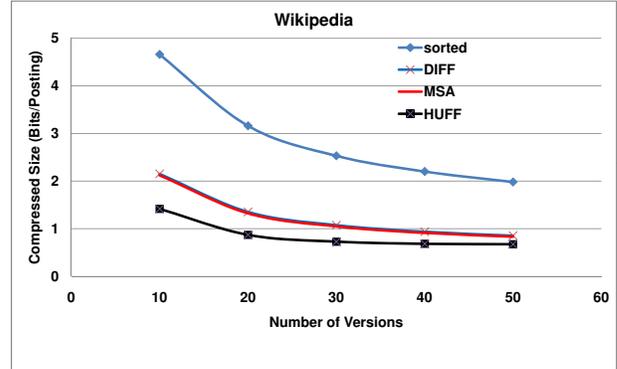


Figure 4: Compressed size in bits per posting for different numbers of versions on the Wikipedia data set. The curves for DIFF and MSA are basically on top of each other.

Next, we look at the impact of the number of versions on compressed size. Intuitively, we would expect more benefit, relative to the baseline unsorted case, when we have more versions. This is shown in Figure 4, where we first select all documents from the Wikipedia data set that have at least 50 versions, and then index the last 10 versions, the last 20 versions, etc., of each document. As we see, not surprisingly, adding additional versions to the index becomes cheaper as we have more versions. Note that for the cost in bits per posting, we assume that the number of postings is given according to the baseline methods, since it is not clear how to define the number of postings for HUFF itself.

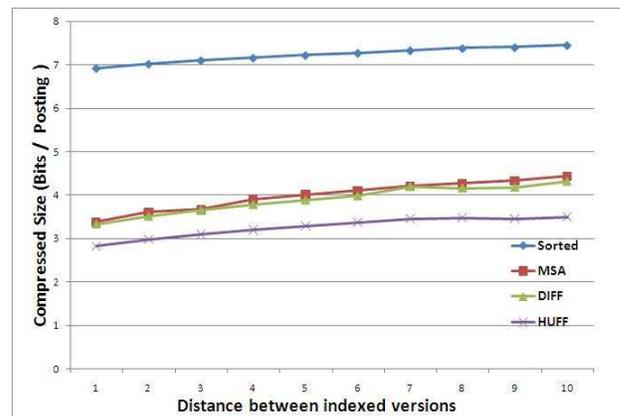


Figure 5: Compressed size for versions selected at different intervals, for the Wikipedia data set. The curves for DIFF and MSA are basically on top of each other.

In Figure 5, we look at what happens when we index not every version, but every k -th version, for k between 1 and 10. In this case, cost slowly increases with k , since versions

a number of edits apart tend to have larger differences than consecutive versions. Note that the absolute numbers in the chart are larger than those in the previous one as we only index 5 versions of each document.

6.4 Query Processing Performance

Next, in Table 5, we show the query processing speeds in milli seconds per query for the Wikipedia data set using *Random*, *Sorted*, MSA, DIFF, and HUFF, with OPT-PFD compression for all methods except HUFF. We assume that the complete index is in main memory, thus there are no disk accesses. A set of 10000 queries was selected at random from a large trace of AOL queries, but limited to those where users clicked on a result from Wikipedia. (In addition, we removed keywords such as “wikipedia” that would not be present if users were directly searching on Wikipedia.) As we see, all the methods are quite fast. *Sorted* does substantially better than *Random*, and DIFF and MSA are almost as fast as sorted but have a much smaller index size. HUFF is slowed down by the cost of decompression using Huffman codes but still outperforms the original baseline.

	speed (ms/query)
Random	3.42
Sorted	0.97
MSA	1.16
DIFF	1.21
HUFF	2.38

Table 5: Speed of Boolean AND query processing on the Wikipedia data with 240000 documents and 8.4 million versions, with complete index in main memory.

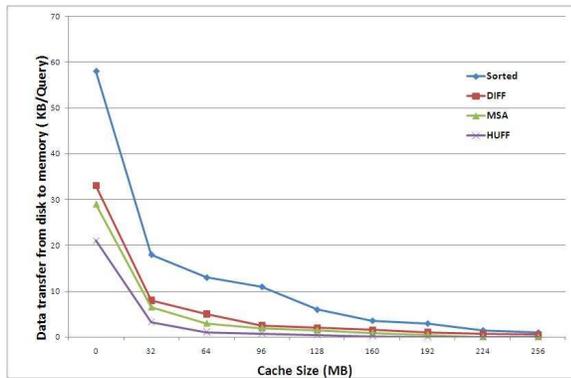


Figure 6: Disk transfers in KB per query for the different methods, as cache size is varied between 0 and 256 MB.

The above results assume that all data is in main memory, but in many practical scenarios the index is on disk and only partially cached in main memory. In this case, better compression results in reduced disk access costs, both because each structure fetched from disk is smaller, and because each structure has a better chance of already being cached in main memory given a fixed cache size. In Figure 6 we show the amount of data per query that is fetched from disk for the different methods as we vary cache size between 0 and 256 MB. Of course, as cache size increases, disk transfers decrease, and the methods with best compression also perform best on disk transfers. As we see, the moderate

improvement in compression offered by HUFF can in some cases result in significant decreases in disk transfers. Thus, while MSA and DIFF are much faster than HUFF when the index is in main memory, when the index is on disk HUFF may sometimes be preferable.

Of course, current machines can afford cache sizes larger than 256 MB. However, our results here are for only 10% of Wikipedia. The costs for caching and query processing are expected to scale roughly linearly with collection size (assuming linear scaling of the cache size), and thus disk accesses and query processing times for the entire collection can be estimated as ten times the above numbers.

7. CONCLUSIONS

In this paper, we have studied techniques for indexing versioned document collections. We proposed several new techniques and experimentally compared them to existing techniques in terms of index size and compression speed. Our results show that specialized techniques for versioned collections outperform naive approaches by a wide margin, and that a two-level approach based on hierarchical bitvectors achieves the best index size among the described methods.

There are a number of interesting open problems related to this work. First, we are currently working to obtain further improvements in index size based on the two-level approach. We believe that additional improvements require a better modeling of text evolution and edit behavior in versioned collections, and we are currently mining data from Wikipedia to better predict changes in text. For example, it is not surprising to observe that terms that first occur in version i of a document are relatively likely to be removed again in one of the next edits, while terms that have survived in the document for a number of edits or a certain amount of time have a good chance of staying in the document. We have already obtained some promising initial results that give further improvements in compression by using such additional knowledge about the collection.

Other related problems of interest are improved techniques for positional indexes for versioned collections, and the proper integration of indexing techniques into revision control systems and versioning file systems.

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, and the anonymous reviewers for very helpful comments and references.

8. REFERENCES

- [1] I. Altıngöve, E. Demir, F. Can, and O. Ulusoy. Incremental cluster-based retrieval using compressed cluster-skipping inverted files. *ACM Transactions on Information Systems*, 26(3), June 2008.
- [2] P. Anick and R. Flynn. Versioning a full-text information retrieval system. In *Proc. of the 15th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 98–111, 1992.

- [3] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A time machine for text search. In *Proc. of the 30th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 519–526, 2007.
- [4] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result-caching. In *Proc. of the 2nd Int. Workshop on Peer-to-Peer Systems*, 2003.
- [5] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. of the 27th European Conf. on Information Retrieval*, pages 375–387, 2005.
- [6] R. Blanco and A. Barreiro. Tsp and cluster-based solutions to the reassignment of document identifiers. *Information Retrieval*, 9(4):499–517, 2006.
- [7] D. Blandford and G. Blelloch. Index compression through document reordering. In *Proc. of the Data Compression Conference*, pages 342–351, 2002.
- [8] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29. IEEE Computer Society, 1997.
- [9] 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, 2006.
- [10] 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, 2000.
- [11] Y. Choueka, A. Fraenkel, S. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. of the 9th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 88–96, 1986.
- [12] 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*, 2002.
- [13] T. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *Proc. of the ACM WebDB Workshop*, 2000.
- [14] S. Heman. Super-scalar database compression between RAM and CPU-cache. MS Thesis, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands, July 2005.
- [15] M. Herscovici, R. Lempel, and S. Yogev. Efficient indexing of versioned document sequences. In *Proc. of the 29th European Conf. on Information Retrieval*, 2007.
- [16] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31(2):249–260, 1987.
- [17] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference*, 2004.
- [18] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.
- [19] 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, 2001.
- [20] A. Orlitsky. Interactive communication of balanced distributions and of correlated files. *SIAM J. on Discrete Math*, 6(4):548–564, 1993.
- [21] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX Conf. on File and Storage Technologies*, 2002.
- [22] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 76–85, 2003.
- [23] 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.
- [24] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Inf. Processing and Management*, 39(1):117–131, 2003.
- [25] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of 29th European Conf. on Information Retrieval*, pages 101–112, 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 27th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 305–312, 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 the 18th Int. World Wide Web Conference*, 2009.
- [30] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. of the 17th Int. World Wide Web Conference*, April 2008.
- [31] J. Zhang and T. Suel. Efficient search in large textual collection with redundancy. In *Proc. of the 16th Int. World Wide Web Conference*, 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 Int. Conf. on Data Engineering*, 2006.