Optimizing Positional Index Structures for Versioned Document Collections

Jinru He*
Facebook Inc.
1601 Willow Road
Menlo Park, CA, 94025
jihe@fb.com

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

ABSTRACT

Versioned document collections are collections that contain multiple versions of each document. Important examples are Web archives, Wikipedia and other wikis, or source code and documents maintained in revision control systems. Versioned document collections can become very large, due to the need to retain past versions, but there is also a lot of redundancy between versions that can be exploited. Thus, versioned document collections are usually stored using special differential (delta) compression techniques, and a number of researchers have recently studied how to exploit this redundancy to obtain more succinct full-text index structures.

In this paper, we study index organization and compression techniques for such versioned full-text index structures. In particular, we focus on the case of positional index structures, while most previous work has focused on the non-positional case. Building on earlier work in [32], we propose a framework for indexing and querying in versioned document collections that integrates non-positional and positional indexes to enable fast top-k query processing. Within this framework, we define and study the problem of minimizing positional index size through optimal substring partitioning. Experiments on Wikipedia and web archive data show that our techniques achieve significant reductions in index size over previous work while supporting very fast query processing.

Categories and Subject Descriptors

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

Keywords

Inverted index, index compression, versioned documents, positional index structures, redundancy elimination.

1. INTRODUCTION

Large search engines now have to process thousands of queries per second over tens of billions of documents, resulting in very significant hardware and energy costs. Query processing algorithms in these engines are based on inverted index structures, and a large amount of research over the last decade has focused on how to better organize, compress, and access such indexes. This has contributed to significant increases in algorithmic efficiency that, together with increases in CPU speeds and counts, have allowed the major engines to keep up with the ever increasing user demands [9].

In this paper, we study how to organize and compress inverted index structures. However, we focus here on the case of versioned document collections, i.e., collections where each document is represented by multiple versions. Such versioned document collections occur in a number of search applications, such as search in the web collection of the Internet Archive, consisting of more than 150 billion snapshots (versions) of web pages collected since 1996, or in Wikipedia, which retains the complete edit history of all articles. Other important scenarios are version control systems, document management systems, and versioned file systems, which retain all past versions of the files they manage. The goal in indexing versioned document collections is to build full-text index structures that allow keyword search across all versions.

The primary challenge is that because all past versions are retained, a versioned document collection is much larger than a collection that only keeps the latest version. For example, the English language part of Wikipedia contained about 2.4 million articles in January 2008, but each article had 35 versions on average. An inverted index that simply indexes each version separately would thus be much larger than an index that only considers the latest versions. However, versioned document collections also have very significant redundancies between the different versions, which could be exploited by suitable compression techniques. In fact, there has been a large amount of research on delta (or differential) compression and other redundancy elimination techniques for storage systems and networks (see, e.g., [23, 16, 27, 19, 29, 28]), and many systems now use such techniques to store their collections.

There has also been some amount of work on how to exploit these redundancies to better compress full-text indexes for versioned document collections [4, 7, 32, 15, 5, 11, 12, 8]. Note that this is a different problem and that compression schemes for content do not directly imply how to compress an inverted index, though some basic ideas are useful in both scenarios. Most previous work on indexing has focused on non-positional index structures storing document IDs and maybe term frequencies or quantized impact scores, while only [32, 8] consider positional index structures that also store the locations of term occurrences inside the document versions.

We focus here on such positional index structures. Positional indexes are used by most search engines and other IR systems to enable better ranking and for features such as phrase searches. However, positional indexes are typically about 3 to 5 times larger than non-positional ones. An additional challenge arises in the versioned case, in that a single insertion or deletion of a term produces a shift in the position information for all subsequent terms in the new doc-

*Work performed while this author was a PhD student at Polytechnic Institute of NYU

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.
SIGIR'12, August 12–16, 2012, Portland, Oregon, USA.
Copyright 2012 ACM 978-1-4503-1472-5/12/08 ...$10.00.

ument version. This problem does not exist in the non-positional case, where we can simply add or remove one posting, and as a result, techniques for the positional case are very different from those for non-positional indexes.

In this paper, we propose improved techniques for positional indexing in versioned document collections, building on the approach described in [32]. In particular, we describe a framework for efficient query processing with non-positional and positional index structures that combines the approaches in [32] and [12]. Within this framework, we study the problem of optimizing positional index size by modeling it as a substring partitioning problem. We then describe heuristic optimization algorithms for this problem that can scale to large document collections. Our experiments on large versioned data sets from Wikipedia and the Internet Archive show significant reductions in index size over [32] and [8] with very fast access speeds.

The remainder of the paper is organized as follows. Next, we provide some technical background and discuss related work. Section 3 summarizes our contributions. Section 4 describes our overall indexing and query processing framework and provides some baseline experimental results. Sections 5 and 6 describe algorithms for document partitioning and our lookup mechanism for accessing position data. Finally, Section 7 provides some concluding remarks.

2. BACKGROUND AND RELATED WORK

We now provide some background and describe related work. We first introduce inverted indexes and index compression methods. Then we discuss previous work on indexing and querying versioned document collections. Finally, we provide background on redundancy elimination using content-dependent partitioning methods.

2.1 Inverted Index Structures

Most search engines and other textual IR systems use inverted index structures to support keyword queries. For a given document collection C, let $w_0, \ldots, w_{m-1}$ be the set of all distinct terms in C. An inverted index I_C for C contains an inverted list I_{w_i} for every i. Each inverted list I_{w_i} is a list of index *postings*, where each posting contains the ID (called docID) of a document containing w_i, and additional information such as the number of occurrences of w_i in d (called frequency), or the locations of each occurrence of w_i in the document (called positions). An index structure is called *positional* if it contains position information, and non-positional otherwise. In this paper, we focus on indexes with docIDs, frequencies, and positions. We refer to [33] for a survey of inverted indexing techniques and applications.

There are several ways inverted index structures can be laid out on disk or in memory. A direct layout would place each posting's docID next to the corresponding frequency and position values, but in practice this is rarely done. Better performance is usually achieved by either keeping docIDs, frequencies, and positions in separate layers with their own access structures, or at least interleaving larger chunks of values (e.g., 128 docIDs followed by their associated frequencies). We will use separate layers for docIDs, frequencies, and positions.

Index compression: The postings in inverted lists are usually sorted by docID and then compressed [33]. To obtain better compression, most approaches store not the actual docIDs, but the differences between consecutive docIDs in the list, called *d-gaps*. The same is also done for different position values within the same document. Of course, this requires summing up these differences again during decompression. To support fast random access, inverted lists (or their individual layers) are often organized into blocks of, say, 128 values, such that each block can be individually accessed and decompressed by searching in an auxiliary array containing the last (or first) docID of each block in uncompressed form.

Many inverted index compression algorithms have been proposed. In our experiments, we use the OPT-PFD method proposed in [31], which belongs to the *PForDelta* [13, 34] family of compression methods. OPT-PFD achieves very high decompression rates (up to more than a billion integers per second and per core on current CPUs). It also achieves a very small compressed index size, particularly for highly clustered data sets. OPT-PFD was previously used to compress non-positional indexes for versioned collections in [11, 12]. Some moderate additional reductions in index size could be obtained by using the Interpolative Coding techniques (IPC) in [22], at the cost of slower position lookups.

Query processing and positions: Very simple ranking functions such as BM25 or the Cosine measure can be implemented by intersecting (conjunctive queries) or merging (disjunctive queries) the inverted lists of the query terms, and then computing a score based on the frequency data in the postings and other information such as document sizes and term frequencies. However, it is known that position data can be used to obtain significantly better result rankings than those achievable using only frequency data (see, e.g., [21]).

Fetching the position data for all postings in the intersection or union of the inverted lists would be quite expensive. Instead, a widely used approach [24] first applies a simple non-positional ranking function (e.g., BM25), and then determines the top-k results under the positional ranking by fetching position data only for the top k' results under the simple function, for some $k' > k$. This significantly reduces the cost of fetching positions, such that positional ranking can be performed in almost the same time as non-positional ranking. One consequence of this approach is that accesses to positional index data are much sparser than accesses to docIDs and frequencies.

2.2 Indexing Versioned Collections

A versioned document collection C is a set of documents $d_0, \ldots, d_{n-1}$ where each document d_i consists of m_i versions $d_i^1, \ldots, d_i^{m_i}$. For simplicity, we assume a linear history of versions; however, our techniques also apply to collections with branches (forks) in the revision history. For a general overview of challenges in building large-scale archival systems for versioned data, see [3, 26]. There is some amount of recent work on indexing and searching in versioned document collections that can be split into three subsets, as follows.

Non-positional versioned indexing: A number of compression methods for non-positional versioned indexes have been proposed [4, 15, 7, 5, 11, 12, 8]. The first work on this problem appears to be the work of Anick and Flynn in [4], which proposes a scheme based on the idea of indexing the delta between consecutive document versions and then adjusting query processing suitably. The last few years have seen several new techniques that achieve much better compression. While our focus in this paper is on positional index structures, our complete query processor also requires an additional non-positional index. This index is used to first compute a simple ranking function that determines which items need to be fetched from our positional structure, as discussed at the end of the previous subsection.

We employ a non-positional index structure proposed in [12], which appears to achieve a very good trade-off between size and speed. This structure is based on a two-level approach, inspired by an earlier idea in [1] for a related but different problem. In a nutshell, the idea is to use two levels of indexing, a small first-level index that only stores whether a term occurs anywhere in a document (plus quantized upper-bound scores for each document in the case of disjunctive queries), and a larger second-level index that stores which versions contain the term and what their frequencies are. Query processing involves accessing the first level to perform filtering, and then selectively those parts of the second level that are still needed.

However, other non-positional structures could also be used with our approach.

Positional versioned indexing: There has been much less work on positional indexes for versioned collections. We note that there are basic differences between the positional and non-positional case. Non-positional indexes can use a set- or bag-oriented approach to model differences between consecutive versions of a document. In the positional case, however, we need to think in terms of common substrings. This way a change in position information due to an insertion or deletion before the start of the substring can be handled by changing the offset of the substring in the document, following an idea first proposed in [20] for index updates. Thus, techniques for non-positional indexes are of limited help for the positional case.

We are aware of two previous approaches for positional indexing of versioned collections [32, 8]. The basic idea in [32] is to use a content-dependent partition technique such as Karp-Rabin fingerprints [18] or *Winnowing* [25] to split document versions into fragments during indexing. Then each unique fragment is only indexed once. This is done by computing for each fragment a hash value that is compared to those of the already indexed fragments. In addition, extra meta data structures are built, and appropriate changes in the query processor made, in order to translate fragment-based matches back to document versions during query processing.

Thus, the performance of the method in [32] depends on finding a high-quality partition of document versions into fragments, such that the total size of the distinct fragments is minimized. In this paper, we follow the approach in [32], and our main goal is to come up with new fragment selection and optimization mechanisms that improve significantly over the Winnowing technique used in [32].

The approaches in [8], on the other hand, are based on very different techniques such as Lempel-Ziv compression, re-pairing, and self-indexing structures. We will provide an experimental comparison with results in [32] and [8] in later sections.

Range and Aggregation Queries: Several recent papers have studied query operations other than standard ranked queries for versioned collections. In particular, [2, 10] looked at how to support temporal range queries over versioned collections, where queries are restricted to a certain interval of time. Also, [30] proposed a query operator called *durable search* that returns documents that consistently rank highly over some time period, thus aggregating scores over multiple versions.

We note here that these operations are orthogonal to our approach, and could be easily integrated into our framework by suitably modifying the non-positional part of the index and query processor. In general, we try to sidestep the question of what the best ranking function and query operations are, and in our experiments we just use a fairly simple ranking function (BM25) in the non-positional part in order to drive the lookups into our positional structure.

2.3 Content-Dependent String Partitioning

A large amount of research in operating systems and networking has focused on how to detect and eliminate redundancies in large data collections; see, e.g., [23, 16, 27, 19, 29, 28]. While the proposed methods utilize a number of interesting algorithmic techniques, one technique, *content-dependent string partitioning*, is of particular interest to us here.

In a nutshell, the goal is to divide each file into substrings (or fragments) such that files that contain a lot of similarity have a lot of common substrings. Note that if we simply split files into fixed-size fragments, we might not see any common fragments at all even if the files differ by only a single insertion or deletion at the beginning of the file, due to misalignments of the fragment boundaries in the files. Thus, the goal is to come up with partitioning methods that are robust to small changes due to insertions and deletions, and that can be applied independently on the various files, which might in fact be located on different machines.

Several approaches have been proposed in the literature which all use hash values on small local substrings to determine fragment boundaries. This means that any long common substrings in different files (or versions) are partitioned in the same way, resulting in common fragments. The first and best known partitioning technique is based on Karp-Rabin (KR) fingerprints [18], and works as follows: We first slide a window of some size c through the file and compute a hash value $h(s)$ for each substring s of length c. We then declare a fragment boundary before any substring s such that $h(s)$ is $x \, mod \, y$ for some fixed choice of x and y. Two more recent techniques, the Winnowing (WIN) approach in [25] and the 2-way Min (2MIN) approach in [29], follow the same idea of hashing substrings of fixed size, but use different conditions for defining fragment boundaries.

While all three approaches are used in the literature, experimental results suggest that 2MIN [29] results in more (and overall longer) common substrings, thus identifying more redundancy. The technique works as follows: As before, we compute hash values for all substrings of some fixed size c, say $c = 10$ or $c = 20$. This gives us an array of integer hash values $h[0 \cdots n-c]$. We then set a fragment boundary before position i in the input file if $h[i]$ is strictly smaller than all other $h[j]$ with $i - w \leq j < i + w$ for some fixed w that we select. In other words, we set a boundary wherever the hash value is the smallest value within a neighborhood of radius w. This results in an average fragment size of about $2w$. The process is illustrated in Figure1:

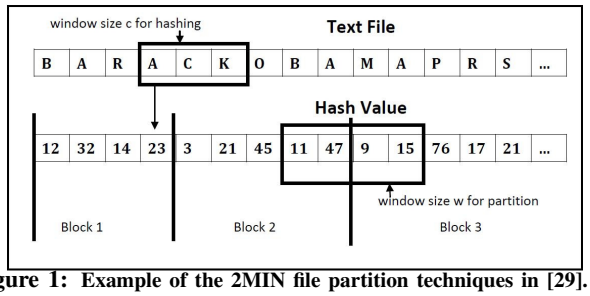

Figure 1: Example of the 2MIN file partition techniques in [29]. A small window of size $c = 3$ moves over the input file (top) to create an array of hash value (bottom). Then we select fragment boundaries using $w = 2$.

3. OUR CONTRIBUTIONS

In this paper, we study the problem of organizing and compressing positional index structures for versioned collections. In particular, our main contributions are as follows:

- We describe a complete framework for full-text indexing and querying in versioned document collections. This framework combines ideas from [32] and [12], and enables fast top-k query processing with succinct index structures.
- We study the problem of finding good partitionings of document versions into fragments. Our methods differ from strictly local methods such as KR, WIN, and 2MIN in that we can exploit knowledge of the edit history of a document.
- We describe a multi-level optimization approach for finding good partitionings that scales to very large data sets.
- Finally, we perform an experimental evaluation of our approach based on large document collections from Wikipedia and the Internet Archive. The results show significant reductions in index size over previous work while maintaining very fast query processing.

4. VERSIONED INDEXING FRAMEWORK

We now describe our general framework for indexing and querying versioned document collections. The framework further develops earlier ideas in [32], and combines them with ideas for the non-positional case in [12]. We first give a high-level description of our positional indexing approach, and then show how to combine positional and non-positional indexes to support query processing. Finally, we provide some experimental results for several baseline methods.

In the following, we use the term *fragment* to refer to any substring generated by some partitioning of the document versions, based on the content-dependent partitioning schemes in Subsection 2.3 or other techniques. All our partitioning schemes and fragments respect word boundaries, such that no word is cut in the middle; this is done by first replacing words with word IDs, and then hashing each sequence of c word IDs. Each occurrence of a fragment f is called an application of f. We assume that each document is identified by a *docID*, each version of a document by a *vID* (which could consist of a docID and a version number), and each fragment by a *fragID*.

4.1 Positional Index Structure

Now we describe our positional index structure for versioned document collections.

Positional indexing process: We first describe the indexing process assuming that we directly apply one of the content-dependent partitioning techniques in section 2.3, as in [32]; we later discuss how to accommodate the new partitioning schemes used in this paper. To build the positional index, we partition each new version of a document into fragments. For each fragment, we then compute an MD5 hash over the entire fragment, and check a table to see if the same fragment was previously encountered in the same document. If the fragment has not occurred before, we assign it a new fragID, and then index the fragment as if it were a separate document, creating postings consisting of a fragID and the position of a term within the fragment. Note that there is no frequency stored in the postings since most such frequencies are 1; it is more efficient to repeat the fragID in several postings if needed, at least after index compression. The resulting posting lists are ordered by fragID, where fragments in the same document are assigned a common range of fragIDs. The fragment-based position index is then compressed using standard techniques. In particular, we use OPT-PFD on blocks of 128 values for both fragIDs and positions within fragments, thus allowing efficient random lookups into the lists.

Meta data structures: To support positional lookups on this index structure, we need additional data structures that allow us to map between positions in fragments and positions in document versions. Thus, we maintain a *version/frag table* that stores, for each version, the number of fragment applications in the version and the list of fragIDs in the order in which the fragments occur in the version. This table is compressed using suitable techniques, as described later. In addition, we store the size of each fragment (in words) in a separate *frag size table*. Note that these two tables, which we call meta data, are kept completely in main memory during query execution.

All the major structures are shown in Figure 2. Our meta data structures are similar to, but different from, those described in [32]. Note that we are not showing the table of fragment hashes mentioned earlier. The reason is that this table is only useful for index updates when directly applying the content-dependent partitioning schemes in Subsection 2.3. Our goal in later sections is to come up with new partitioning schemes that exploit the history of a document for better compression; however, the update scheme based on a hash table will not apply in this case anymore. We discuss this issue later.

Performing position lookups: Accesses to the positional index structure are in the form of lookups containing a term and a vID, asking for the positions of all occurrences of the term in a version. A lookup is performed by first obtaining the range of fragIDs assigned to the document – recall that each document has its own contiguous range of fragIDs, though this is not true for each version. Then we access the inverted lists to get all fragIDs within this range that contain the term. Next, we access the version/frag table and fetch and uncompress the entry for the version. Finally, we use this information to check which fragIDs returned by the index occur in the requested version, and for those that do, we translate the position within the fragment back to a position within the version.

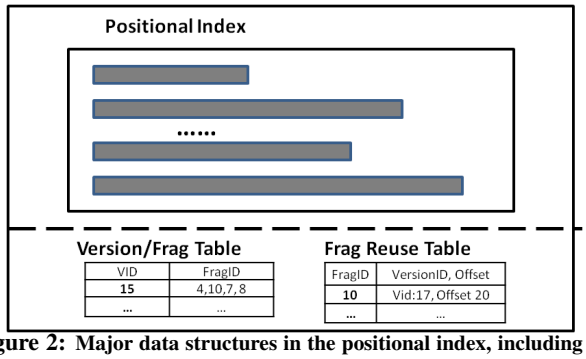

Figure 2: Major data structures in the positional index, including the inverted lists (top) and the meta data (bottom).

Note that if several versions of the same document need to be looked up for the same term, we can do this faster by passing a list of vIDs and performing only one lookup into the inverted list. Finally, when accessing several inverted lists for the same vID (as would happen as part of a multi-term query), we could try to fetch and uncompress the corresponding entry in the version/frag table only once, but the benefits of this are very small.

4.2 Query Processing

Next we describe how to integrate the positional index structure into our overall framework for positional query processing over versioned document collections. For this, we add a non-positional index structure. In particular, we implemented a two-level index structure called 2R-MSA described in [12], which supports fast ranked top-k queries under simple non-positional ranking functions such as BM25. The resulting structure is sketched in Figure 3, showing the two levels of the non-positional index on the left, and the positional index and meta data on the right.

Note that only the meta data needs to be kept in memory during query processing, while all layers of the inverted lists can be kept on disk or partially cached in main memory using any state-of-the-art caching policy. Also, the first-level, second-level, and positional inverted lists for each term are actually laid out next to each other on disk, such that all three structures can be fetched with a single disk seek. Finally, we note that in principle the non-positional index is not even required to perform query processing, as all necessary information is also contained in the positional index and meta data. However, each such query would take hundreds of ms to complete, while the non-positional two-level index (which is much smaller than the positional index) accelerates it to a few ms.

Using this setup, a top-k keyword query for a positional ranking function is executed by first issuing the query to the non-positional index structures to return the top k' results (vIDs) for some $k' > k$. This is done by using DAAT query processing over the first level, and lookups into the second level, as described in [12]. In the second stage, we perform lookups into the positional index for these k' results, and then use the positional data to rerank the results and return the top k results.

Note that there is a natural trade-off between meta data size and positional index size that is present in both our work here and [32].

Basically, using smaller fragments results in more meta data but also more common fragments between versions and thus (up to a point) a smaller index. Our goal in this paper is to obtain the best trade-off between meta data and index size. A smaller meta data size means that less memory is needed to store this data, and that more main memory is available for caching the other index data, an issue that we will explore later.

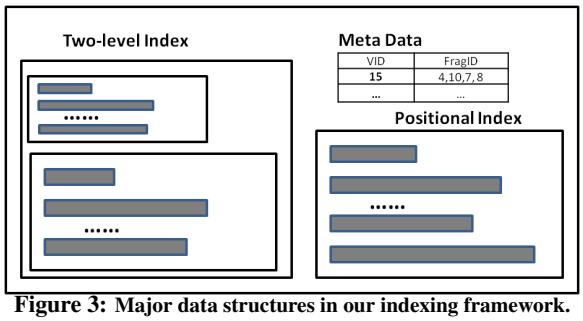

Figure 3: Major data structures in our indexing framework.

One feature of our framework is that it separates positional and non-positional query processing issues. Thus, for example, the decision on whether to perform disjunctive or conjunctive query processing only impacts query processing in the non-positional part. It is also easy to support temporal range queries by applying the techniques in [10] in the non-positional index. Aggregate operations such as the durable search operator in [30] can also be handled in the first level. Finally, when performing searches in versioned document collections, it is often desirable to limit the number of results from the same document that are returned, say, returning only the highest-scoring version of each document. This is also easy to do in our setup.

4.3 Preliminary Experimental Results

In this section, we first describe the data sets used in this paper. Then we present some preliminary experimental results comparing existing approaches, thus setting a baseline over which to improve in subsequent sections.

Data sets: We use a versioned document collection from Wikipedia (WIKI) and a data set from the Internet Archive. The Wikipedia data consists of 240,000 distinct documents with on average 35 versions per document. The documents were selected as a 10% random sample from a complete snapshot of the English language Wikipedia from Jan 2001 to Jan 2008. The Internet Archive data consists of 1.06 million documents from the Irish web domain collected between 1996 to 2006. Only documents with at least 10 distinct versions were selected, and each document has 15 versions on average. Similar data sets were also used in [30, 12, 8].

Experimental results: We now present preliminary experimental results on our data sets. We compare three basic approaches, the one in [32] (ZS), the best approach in [8] based on VByte and LZMA (CFMN), and a trivial baseline where all versions are treated as separate documents (no sharing). For the ZS approach, we explore three different content-dependent partitioning techniques: KR, WIN (used in [32]), and 2MIN.

We set the parameter $w = 20$ for the 2MIN version of ZS, resulting in a total of $348,548,098$ fragment applications. To enable a fair comparison with ZS-KR and ZS-WIN, we selected parameters for those methods such that they result in approximately the same number of applications and the same meta data size. For CFMN, we report the best result presented in [8]. CFMN and the trivial baseline do not have a distinction between meta data and the actual index.

Table 1 shows the index size, meta data size (if applicable), total index size, and number of indexed positions for each method, for the WIKI data set. All methods except CFMN use OPT-PFD to compress the positional index. As we see, all methods achieve significant improvements over the trivial baseline. Both ZS-WIN (used in [32] and ZS-2MIN) achieve a smaller size than CFMN, with ZS-2MIN outperforming ZS-WIN by about 10%. Surprisingly, ZS-KR is much worse than the other methods, showing that the choice of content-dependent partitioning technique can make a big difference in practice.

	index	meta	total	millions of positions
No sharing	19053	-	-	14404
ZS-KR	2762	96	2858	1880
ZS-WIN [32]	1479	96	1575	971
ZS-2MIN	1324	96	1420	867
CFMN [8]	-	-	1954	-

Table 1: Comparison of resulting index sizes for different baseline methods on the Wikipedia data. Shown from left to right are index size, meta data size, and total index size in MB, and the number of positions indexed in millions.

	index	meta	total	millions of positions
No sharing	13026	-	13026	9885
ZS-KR	4820	84	4904	3391
ZS-WIN [32]	3708	84	3792	2422
ZS-2MIN	3618	84	3702	2386

Table 2: Comparison of resulting index sizes for different baseline methods on the Internet Archive data set. Shown from left to right are index size, meta data size, and total index size in MB, and the number of positions indexed in millions.

Table 2 shows results for the Internet Archive data set. For this data set, no results for CFMN were available. Here, the reduction in size over the trivial baseline is smaller but still substantial (up to a factor of 3.6). This is probably because the average number of versions per document is smaller than for Wikipedia. Again, ZS-2MIN achieves the smallest index size of all. Thus in the remainder of the paper, we use ZS-2MIN as the baseline over which we need to improve.

5. IMPROVED PARTITIONING METHODS

In this section, we describe new algorithms for selecting good fragments, i.e., for partitioning versions into substrings in a way that obtains a good trade-off between index and meta data size.

We start by defining the problem more formally. To model index size, we assume (for now) that it is proportional in the number of positions that are indexed, that is, the sum of the sizes of all distinct fragments that are created in the partitioning. On the other hand, the size of the meta data is proportional to the total number of fragment applications. This is because the meta data size is dominated by the version/frag table, which needs to store at least one fragID for each application. Note that fragments in our setup are only reused within versions of a single document, and thus we perform fragment selection separately in each document. For a document d with versions $d^1, \ldots, d^m$, the problem of minimizing index size given an upper bound on meta data size can be formulated as follows:

Problem: Given strings $d^1, \ldots, d^m$ and an integer t, find a set of strings F minimizing $\sum_{f \in F} len(f)$ such that every d^i can be partitioned into a multiset of substrings C^i with $\sum_i |C^i| \leq t$ and $set(C^i) \subset F$.[1]

The problem discussed here is similar to the *Minimum Substring Cover Problem* studied by Hermelin et al. [14], which was motivated by applications in Computational Biology and Formal Language Theory. The main difference is that in [14] the bound t is on the size of each C^i, rather than the sum of the sizes. This changes the structure of the problem, and thus the algorithms in [14] and their proof of NP Completeness do not extend to our case. (We con-

[1] $set(C^i)$ is the set corresponding to multiset C^i.

jecture that our problem is also NP Complete, but are still working on a proof.)

Note that the numbers of postings and fragment applications do not provide a precise model for the sizes of the index and meta data structures, respectively. The meta data contains an additional frag size table that is not accounted for in this model, and both structures are compressed using a technique (OPT-PFD) that defies easy analysis. However, we found that this simplistic model is usually good enough to guide our optimization methods towards good solutions (though we introduce a slightly refined model later).

In the following subsections, we describe several methods for finding good partitionings via careful selection of fragments, given a limitation on the number of fragment applications. Most of our methods follow the same overall structure, consisting of several levels of optimization. On the top level, we need to decide how much of the allowed meta data size (or number of applications) should be allocated to each document, and on the lower level we optimize fragment selection in each document given a bound on the number of applications. This fragment selection method greedily selects fragments from a pool of candidate fragments, and the main overall challenge is how to generate a limited-size pool of promising candidate fragments, given that putting all possible substrings into the pool is not a feasible option. Due to space constraints we can only sketch some of the methods.

The main idea motivating our work here is that standard content-dependent partitioning techniques such as KR, WIN, and 2MIN choose boundaries using only local information, and are thus blind to version history. In fact, this is a major advantage in many applications, where boundaries have to be chosen independently, e.g., for similar files located on different machines. But in our case, we have all the different versions available during indexing, and thus we should be able to select better fragments by looking beyond a single version and peeking into earlier and later versions of the same document. While this seems intuitive, it turns out to be quite challenging to really outperform the results obtained by just applying 2MIN to the approach in [32].

5.1 A Simple Improvement

We start out with a simple improvement over ZS-2MIN that demonstrates some of the issues we have to address. Recall that the 2MIN partitioning uses a parameter w that determines the average fragment size. However, it is not obvious what the best choice of w is, as this depends on the amount of redundancy between different versions. More significantly, we should not expect the best overall trade-off from using the same parameter w for all documents – we may get much better results by using a smaller w for some documents and a larger for others.

ZS-MF: This leads to the first improved heuristic, ZS-MF. We start out by partitioning every document several times using 2MIN with different parameters w, and storing the number of postings generated and the number of fragment applications for each w. This results in a small table such as Table 3 showing a trade-off between postings and fragment applications. We generate such a table for each document in the collection.

w	# of frag. applications	# of dist. frags	# of positions
no sharing	-	-	68906
50	4832	483	51580
42	5518	509	46112
34	6641	552	41056
26	9003	608	34167
18	12538	713	27762
10	21844	956	21420

Table 3: An example of an actual position/meta data trade-off table for a document, where w is varied from 10 to 50.

Note that there is no obvious best choice for w based on the above table, as the best choice depends on choices available in other documents. To deal with this, we optimize across all different documents as follows. We first select in all documents the largest value of w. If this already results in more applications than the global budget, then we would fail, so we make sure to choose a large enough maximum value for w. Now we repeatedly select one document and decrease its w value, resulting in an increase in the number of applications but a decrease in index size, until we reach the limit for the number of applications. In fact, we greedily select the document, and the new smaller value of w, that gives us the largest benefit, defined as the ratio of index size decrease and meta data increase.

To do this, we initialize a heap structure containing an entry for every possible choice of document and new value of w. Elements in the heap are organized by their benefit. Now we repeatedly extract the element with the largest benefit, update our overall counts of fragment applications and positions, and then update any other elements in the heap whose benefit is changed due to our choice. In the example table above, if we decrease w from 50 to 34, we would delete the entry for $w = 42$ from the heap, and update the benefits of $w = 26$, $w = 18$, and $w = 10$ to reflect the new starting point $w = 34$. (Only entries for the same document need to be deleted or updated.) We continue this process until we have exhausted our fragment application budget.

The simple heuristic used in ZS-MF already gives measurable improvements over ZS-2MIN, as we will see in the later experimental results. Note that the heap-based optimization across different documents using trade-off tables is also used in all our subsequent algorithms. Finally, we note that partitioning files using many different values of w does not have to be expensive at all, due to an interesting property of the 2MIN partitioning scheme. In particular, any boundary selected using w will also be a boundary for any $w' < w$, fragments obtained using different values of w form a hierarchical partitioning, and we can compute the boundaries for all possible values of w in a single computation using some simple data structures.

5.2 Hierarchical Fragment Selection

We now present the next set of heuristics for further improvements. While the approach in the previous subsection tried different values of w, in the end we selected one single value of w in each document. However, one would expect that updates between versions are not uniformly distributed over files, and that it might be beneficial to use different fragment sizes in different parts of the file (e.g., smaller or larger fragments at the start or end of the file, or in earlier or later versions). In fact, a similar idea was used in [17] in the context of removing redundancy in network transmissions.

Implementing this requires a new mechanism that allows us to select a mix of fragments of different sizes. Our high-level approach for this is as follows: We start out by generating a large pool of candidate fragments for each document. Then we greedily select good fragments from the pool until all versions are completely covered, as discussed further below. The main problem is how to create a good candidate pool. We propose two simple approaches for this:

ZS-MFS: We use 2MIN partitioning in order to define a hierarchical partitioning, by varying w from a small value w_0 to some maximum value, and defining the candidate pool as containing all fragments on all the levels. We note that this number is linear in the number of bottom-level fragments.

ZS-MOF: This approach attempts to create a larger and richer candidate pool by using a hierarchy that contains additional partially overlapping fragments on some levels. In particular, we first select all fragments for the smallest value of w_0. Then we add to the pool new artificial fragments consisting of any consecutive k fragments from the bottom level, for $k = 2, 3, 4$ up to some k_{max}. Above this level, we simply continue as in ZS-MFS. Thus, in ZS-MOF we put all candidate fragments from ZS-MFS into the pool, and then add additional overlapping (shifted) fragments at the lower levels. As we

will see, in practice $k_{max} = 3$ or 4 is enough, and thus the candidate pool does not get overly large. (We also tried other schemes for creating larger candidate pools of overlapping fragments, but this simple approach did almost as well as the best.)

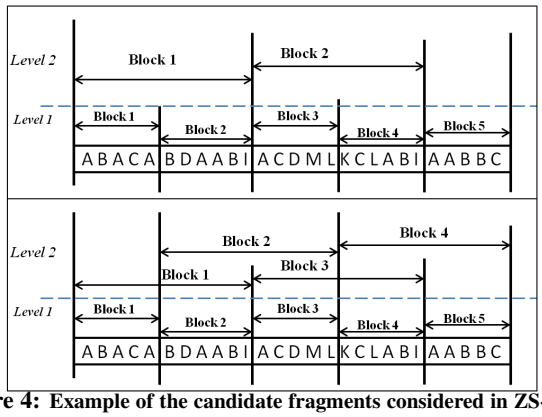

Figure 4: Example of the candidate fragments considered in ZS-MFS (top) and ZS-MOF (bottom), showing only the bottom two layers. As shown, ZS-MOF adds additional overlapping fragments in Level 2. Note that we use characters instead of word IDs for the underlying terms, to simplify the figure.

The ZS-MFS and ZS-MOF schemes are illustrated in Figure 4. We now describe how to select fragments from the candidate pools. We first insert all fragments in the pool into a heap, organized by a fragment score s, and then greedily select the fragments with the highest scores from this heap. To define the score s, we first need to introduce three other values for each fragment:

- the coverage $c = l \times m$ where m is the number of times the fragment occurs in the document AFTER removing all occurrences that intersect with previously chosen fragments.
- the index cost, which we define as $i_c = 1.0$ (one unit per position value).
- the meta data cost, which we define as $m_c = 1.0 + z \times m$ for some constant z (we choose $z = 1.0$).

Then the score of a fragment is defined as $s = c/(i_c + x \times m_c)$. Here, the value x determines how much we value meta data size versus index size, and running the optimization with a larger x results in a solution with a smaller meta data size. Thus, by trying several different values of x we can again get a trade-off table as in ZS-MF, and then run a global optimization across documents as before.

Note that after each extraction of a fragment from the heap, we need to update the s-scores of all other fragments that overlap the chosen fragment somewhere in some version of the document. This is done by keeping for each fragment a list of conflicting (overlapping) other fragments and then updating only these after an extraction. The overall problem of selecting the fragments given a candidate pool and a value x is sketched in Algorithm 1, with some details omitted due to space constraints:

Algorithm 1 optimize(frags, x)

```
Output: results
  results ← ∅;
  h ← ∅;
  for i = 0 to frags.size() do
    frags[i].score = getScore(frags[i], x);
    h.push(frags[i]);
  end for
  while not (h.empty()) do
    f ← h.pop();
    results.insert(f);
    flist ← findConflictFragments(f);
    for i = 0 to frags.size() do
      flists[i].count ← flists[i].count−1;
      if flists[i].count = 0 then
        h.remove(flists[i]);
      else
        /* update count and score for flists[i] in the h */
        flists[i].score = updateScore(flists[i], x);
        h.update(flists[i]);
      end if
    end for
  end while
  return results
```

5.3 Frequency-Based Partitioning

In the previous subsection, we explained how to obtain better partitionings by creating a suitable pool of candidate fragments and then greedily selecting from this pool. For efficiency reasons, the size of the pool had to be limited; thus it is not feasible to place all substrings of all sizes occurring anywhere in the document into the pool, as some documents are quite large. Instead, we attempted to create sparse but rich enough sets of fragments for the pool.

However, all of the candidate fragments were derived from a 2MIN-based partitioning, and all the fragment boundaries occur as boundaries in 2MIN for some w. But there is really no reason to believe that a random hash function (as used in 2MIN) is very good as picking exactly the right fragment boundaries. Thus, given that we can look at the entire history of the document, can we come up with a better and different way to determine fragment boundaries?

In this subsection, we describe such a way. We note that we still take the general approach from the previous subsections. Thus, we have a candidate pool from which fragments are greedily selected based on their s-score, and we also have the global optimization across documents using the trade-off tables.

Consider the first step in 2MIN partitioning, as described in Subsection 2.3, where we slide a window of some size c over a file and compute a hash value for each window. Now however, instead of choosing cuts based on this hash value, we choose cuts based on the frequency (multiplicity) of the hash value which, ignoring collisions, is equal to the number of times this substring of size c occurs in entire document. In particular, we place a cut whenever we see a large change in this frequency from one window position to the next. In particular, when we see a sharp increase, we use the left boundary of the current window to make a cut, and when we see a sharp decrease we use the right boundary of the previous window for the cut.

The intuition for this is quite simple: A high frequency indicates that the content is shared by many versions and thus good material for fragment candidates, while a low frequency indicates content that had a much shorter lifetime. Placing boundaries this way should thus allow us to choose fragments of maximum size that are undisturbed by edits, or more informally, we identify and cut out the bad parts that limit the number of applications that a fragment sees. Also, we have a good chance of getting consistent cuts in different versions, avoiding misalignment problems as discussed in Subsection 2.3 in the context of fixed-size blocks.

To implement this, we need to add two details. First, we need to define when a difference in frequency is significant enough to merit placing a cut; this is determined by a value x. Second, we need a safety valve for cases when the frequency-based approach creates some very large fragments. We do this by using 2MIN to cut any fragment of size greater than some value y, say $y = 100$ or more.

This gives us a partitioning of the versions into basic fragments. We now create additional candidate fragments as in ZS-MFS and

ZS-MOF, by combining several consecutive base fragments into larger candidate fragments. This gives us our candidate pool, and then the rest of the method is as before. We refer to the resulting method as ZS-FREQ.

5.4 Experimental Results

We now evaluate the different partitioning schemes by looking at the resulting index sizes and index construction costs. We compare the four new methods, ZS-MF, ZS-MFS, ZS-MOF, and ZS-FREQ, and the ZS-2MIN method from Section 4. As before, we use OPT-PFD to compress the resulting positional indexes.

Overview of results: We start with the positional index size, total number of positions, and total number of distinct fragments for the various methods, shown in Table 4. To allow a fair comparison, we set the target budget for fragment applications as the number of fragment applications in ZS-2MIN for $w = 20$. The numbers given are for the best choices of various parameters; we investigate some of these choices later. We can make the following observations: First, all methods achieve decreases in total index size and in the number of positions compared to ZS-2MIN. The best result is obtained by ZS-FREQ, which reduces index size by about 22% over ZS-2MIN, followed by ZS-MOF, ZS-MFS and ZS-MF. Second, the total number of distinct fragments is also reduced in these methods. Since the number of fragment applications is fixed, this means that the produced fragments are used more frequently on average (which is of course the whole point).

	index	meta	total size	positions	distinct frags
ZS-2MIN	1324	96	1420	867	20.3
ZS-MF	1238	96	1334	771	18.9
ZS-MFS	1121	97	1218	702	16.7
ZS-MOF	1089	97	1186	661	15.8
ZS-FREQ	1035	97	1133	623	15.4

Table 4: Comparison of the different partitioning methods on the Wikipedia data. Shown from left to right are the index size, meta data size, and total index size in MB, and the total number of positions and distinct fragments in millions.

In Table 5, we show results for selected methods on the Internet Archive data. As before, the reductions in size are more limited, due to the smaller number of versions per document. We observe that ZS-FREQ achieves a roughly 18% smaller index size than ZS-2MIN.

	index	meta	total size	positions	distinct frags
ZS-2MIN	3618	85	3703	2386	55.6
ZS-MOF	3125	86	3211	2060	40.8
ZS-FREQ	2971	86	3057	1893	38.2

Table 5: Comparison of selected partitioning methods on the Internet Archive data.

In Table 6, we show the time needed to build the index structures for the trivial baseline, ZS-2MIN, and ZS-FREQ. All runs were performed on a single core of an Intel Xeon E5520 processor running at 2.26 Ghz, using the Wikipedia data with a total of about 8 million versions and about 19 billion term occurrences. We divide the index construction into three phases: First, the time for selecting good fragments; this applies only to ZS-FREQ (and the other new techniques). Second, the time for parsing and creating postings, which in the case of ZS-2MIN also includes the cost of running the 2MIN partitioning (for ZS-FREQ, this cost is mostly included in the time for the first phase). Third, the cost of merging the postings into a final index.

We see that ZS-FREQ spends most of its time in the selection phase, which is not surprising. Most of this time is spent inside the **optimize()** algorithm, and we expect this could be brought down somewhat through additional optimizations. We note that the time for ZS-MOF and ZF-MFS is similar to that of ZS-FREQ, while that for ZS-MF is closer to ZS-2MIN. ZS-2MIN spends a lot of time in the 2MIN partitioning code, but then is much faster than the trivial baseline during the merge, as much less position data needs to be processed in this phase.

	No sharing	ZS-2MIN	ZS-FREQ
Selection	-	-	18hours
Parsing	93 mins	2hours 40mins	10mins
Merging	65 mins	5 mins	5mins
Total	2hours 38 mins	2hours 45mins	18hours 15 mins

Table 6: Index construction time for different algorithms on the Wikipedia data.

Choosing parameters: Next, we present results on how to tune the various parameters. In Figure 5, we show the trade-off between positional index size and meta data size for the various methods on Wikipedia. To obtain the results, we first use ZS-2MIN with window sizes ranging from 20 to 40. Then we run the other algorithms using the number of fragment applications from ZS-2MIN as an upper bound. We see from Figure 5 that as the meta data size increases, positional index size decreases for all methods, as expected. Our new partitioning methods all achieve a better trade-off than ZS-2MIN, with ZS-FREQ consistently achieving the best results. We note that even the largest meta data size chosen, about 97 MB for our 10% data set, would result in a meta data size of slightly less than 1 GB on the complete Wikipedia data of 2.4 million articles and over 80 million versions. Keeping this amount of data in main memory is clearly feasible on many current systems.

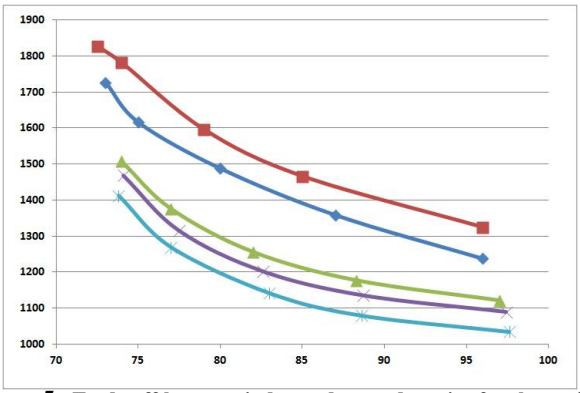

Figure 5: Trade-off between index and meta data size for the various algorithms on the Wikipedia data. The x-axis shows the compressed meta data size in MB, while the the y-axis shows the positional index size in MB.

Next, we investigate some more parameter choices. For the ZS-MOF method, we had to choose a parameter k_{max} that limits how many levels of overlapping fragments are put into the candidate pool, and we suggested that a value of 3 or 4 is sufficient. This issue is explored in Figure 6, where we show the index size and index construction time of ZS-MOF for different choices of k_{max}. For this data, we used a subset of only 1% of Wikipedia (i.e., 10% of our set), as these runs were quite time consuming. We see that index construction time increases with k_{max}, but index size is already close to optimal for $k_{max} = 3$.

Next, Figure 7 shows the impact of choosing different values for the x (left) and y (right) parameter in ZS-FREQ. These experiments were again run on a 1% subset due to time constraints. In both charts, we have the meta data size in MB on the x-axis, and the index size in MB on the y-axis. We see that choices of $x = 15$ and $y = 100$ appear to get good results over the entire range.

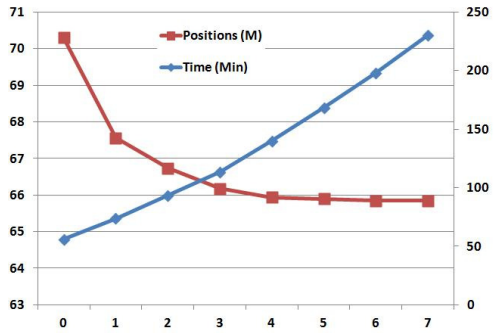

Figure 6: Index size and index construction time for ZS-MOF with different values of k_{max}, on a sample of the Wikipedia data. The y-axis on the left shows the index size in MB, and the y-axis on the right shows the index construction time in minutes.

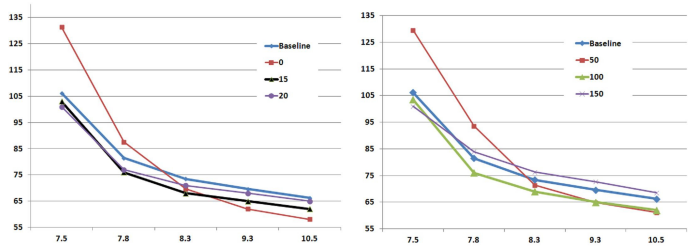

Figure 7: Selecting parameters x and y for the ZS-FREQ method.

6. QUERY PROCESSING

In this section, we describe how to support fast positional lookups during query processing, and then provide some experimental results.

Position lookups: As discussed in Section 4, we assume that a query first executes a simple ranking operation on a non-positional index structure. Afterwards, position lookups are performed for a limited number (a few hundred or a thousand) of the most promising documents that were returned. This means that position lookups are fairly sparse and essentially random accesses into the positional lists. (However, given the characteristics of hard disks, we still have to fetch the entire position list into main memory if it is not cached, as skips in the accesses are rarely large enough to justify extra disk seeks.)

Recall the version/frag table, which is assumed to be held in memory. Then a lookup for the positions of a term within a version consists of three steps: (1) determining the range of the fragIDs assigned to a document, (2) fetching all position data for this range from the position list for this term, and (3) fetching the entry for the version from the version/frag table and using that information plus the data from the frag size table to translate positions within fragments into positions within the version. When performing lookups for several versions of the same document, all vIDs are passed to the lookup mechanism in one call, enabling faster batch processing of these lookups.

The range of fragIDs assigned to a document can be stored in a simple table with one entry per document containing a base fragID for this document. The frag size table is also easily stored as an array of unsigned chars (if fragments are of size at most 256) or shorts otherwise. Some more work is needed to store and compress the version/frag table, which is the largest of the structures. Here, we store the first fragID of each version as an offset from the base fragID of the document, and each subsequent fragID as a gap from the previous fragID (with an additional bit for the sign of the gap). These values are then organized in blocks of 128 and compressed using OPT-PFD, with an appropriate lookup structure that allows fetching of the list of fragIDs for a particular version. A further improvement exploits the fact that the list of fragIDs in consecutive versions is often very similar, by using the list of the preceding version as a reference for encoding the next list (similar to what is commonly done to compress adjacency lists in web graphs [6]). Overall, this results in significant size reductions that are crucial for a good trade-off between meta data and index size.

Lookup performance: We present some performance results for position lookups in Table 7. The results are based on the ZS-FREQ method, but similar numbers hold for all partitioning methods since lookup costs are fairly insensitive to the quality of the partitioning. In the results, we exclude the cost of the initial traversal of the non-positional index, and only count the costs of the positional lookups.

From top to bottom, Table 7 shows the number of positions per query that are returned, the time for finding the first fragID in the relevant fragID range in the index, the time for decoding index data, the time for translating fragment positions back into version positions, and the total time, all in ms per query. Finally, at the bottom we see the cost per position fetched, in microseconds. This is for the case where we allow each document to be represented by multiple versions in the top-k results. As we see, lookups are quite efficient, and costs are around 2.5 ms when we fetch positions for the top-1000 results returned by the non-positional index.

Table 8 shows how lookup times increase as we limit the number of versions per document that are allowed in the top-k results. The motivation for this is that the user may not be interested in receiving 10 results that are all different versions of the same document, and thus a query processor may decide to limit the number of versions from the same document (similar to the way in which search engines may limit the number of results from the same web site). We would expect this to result in slower lookups into our structure as there is less access locality, and less opportunity for batching accesses as discussed. However, even when fetching position information for 1000 versions from 1000 distinct documents, costs are moderate at about 6.44 ms per query.

	Top-10	Top-100	Top-1000
# of decoded positions	130	306	2700
Seek time	0.1	0.24	1.87
Decode time	0.03	0.08	0.29
Translation time	0.02	0.03	0.38
Total time	0.15	0.35	2.54
Cost per position in us	1.2	1.1	0.9

Table 7: Performance of our positional lookup mechanisms. All times are in ms per query, except for the cost per position which is in microseconds.

	Top-10	Top-100	Top-1000
1	0.22	0.92	6.44
5	0.21	0.80	6.25
10	0.15	0.65	5.56
20	0.15	0.51	4.83
40	0.15	0.41	3.79
60	0.15	0.38	2.78
no limit	0.15	0.35	2.54

Table 8: Performance of our positional lookup mechanisms when we limit the number of versions per document allowed in the top-k, in ms per query.

Impact of caching: In the above experiments, we assumed that the positional index is in main memory. We also performed experiments in a scenario where the index resides on disk but is partially cached in main memory, as is the case in many real systems. As expected, all methods benefited significantly from caching, with ZS-FREQ again outperforming the other methods.

7. CONCLUSIONS

In this paper, we have studied how to build succinct positional full-text index structures for versioned document collections. We

described a general framework for indexing and querying based on document partitioning, and then proposed new techniques for selecting good partitionings. Our experimental results showed that the proposed methods achieve significant reductions in index size while also supporting very fast query processing.

There are a number of open issues that we are trying to resolve. First, we believe that additional refinements in the ZS-FREQ approach could result in additional index size decreases and much faster index building methods. The current approach with its reliance on the x and y parameters seems somewhat ad-hoc. Second, we are looking at how to best update our structures. One of the benefits of the approach in [32] was that updates could be handled easily through use of a hash table mechanism, but this breaks down with the new partitioning techniques. Part of this is fundamental: Since our goal is to exploit the entire history of a document, any incremental updates are bound to eventually degrade the performance of the scheme. However, we believe that some of these drawbacks can be ameliorated.

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.

8. 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] A. Anand, S. Bedathur, K. Berberich, and R. Schenkel. Efficient temporal keyword search over versioned text. In *Procòf the 19th ACM Int. Conf. on Information and Knowledge Management*, 2010.

[3] A. Anand, S. Bedathur, K. Berberich, R. Schenkel, and C. Tryfonopoulos. Everlast: a distributed architecture for preserving the web. In *Proc. of the 9th ACM/IEEE Joint Conference on Digital Libraries*, 2009.

[4] P. G. Anick and R. A. 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*, 1992.

[5] 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.

[6] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *Proc. of the 13th Int. World Wide Web Conference*, 2004.

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

[8] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM)*, 2011.

[9] J. Dean. Challenges in building large-scale information retrieval systems. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, 2009.

[10] J. He and T. Suel. Faster temporal range queries over versioned text. In *Proc. of the 34th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2011.

[11] J. He, H. Yan, and T. Suel. Compact full-text indexing of versioned document collections. In *Proc. of the 18th ACM Int. Conf. on Information and Knowledge Management*, 2009.

[12] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *Proc. of 18th ACM Int. Conf. on Information and Knowledge Management*, 2010.

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

[14] D. Hermelin, D. Rawitz, R. Rizzi, and S. Vialette. The minimum substring cover problem. *Information and Computation*, 206(11), 2008.

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

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

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

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

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

[21] D. Metzler and W. Bruce. A markov random field model for term dependencies. In *Proc. of the 28th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2005.

[22] A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3:25–47, July 2000.

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

[24] Y. Rasolofo and J. Savoy. Term proximity scoring for keyword-based retrieval systems. In *Proc. of the 25th European Conference on Information Retrieval*, 2003.

[25] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 76–85, 2003.

[26] S. Song and J. Jaja. Archiving temporal web information: Organization of web contents for fast access and compact storage. In *Technical Report UMIACS-TR-2008-08*, 2008.

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

[28] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In K. Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002.

[29] 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.

[30] L. U, N. Mamoulis, K. Berberich, and S. Bedathur. Durable top-k search in document archives. In *Proc. of ACM SIGMOD Conf. on Management Of Data*, 2010.

[31] 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.

[32] J. Zhang and T. Suel. Efficient search in large textual collection with redundancy. In *Proc. of the 16th Int. World Wide Web Conference*, 2007.

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

[34] 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.