Faster Temporal Range Queries over Versioned Text

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

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

ABSTRACT

Versioned textual collections are collections that retain multiple versions of a document as it evolves over time. Important large-scale examples are Wikipedia and the web collection of the Internet Archive. Search queries over such collections often use keywords as well as temporal constraints, most commonly a time range of interest. In this paper, we study how to support such temporal range queries over versioned text. Our goal is to process these queries faster than the corresponding keyword-only queries, by exploiting the additional constraint. A simple approach might partition the index into different time ranges, and then access only the relevant parts. However, specialized inverted index compression techniques are crucial for large versioned collections, and a naive partitioning can negatively affect index compression and query throughput. We show how to achieve high query throughput by using smart index partitioning techniques that take index compression into account. Experiments on over 85 million versions of Wikipedia articles show that queries can be executed in a few milliseconds on memory-based index structures, and only slightly more time on disk-based structures. We also show how to efficiently support the recently proposed stable top-k search primitive on top of our schemes.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance.

Keywords

Inverted index, query processing, temporal search, range queries, versioned documents.

1. INTRODUCTION

The major web search engines now have hundreds of millions of users that issue several billion queries per day. Each query needs to be evaluated against the tens of billions of pages covered by the engine, resulting in significant hardware and energy costs. This situation has led to renewed interest in basic query processing architectures and index organizations for large document collections, and a lot of recent research has focused on topics such as index building, index compression, fast index access and traversal, and caching and early termination techniques for fast query processing.

Here, we focus on the special case of versioned document collections. These are collections where each document consists of multiple versions that represent the evolution of the document over time, and the goal is to enable search across the different versions of the documents. Maybe the best-known example of a versioned document collection on the web is Wikipedia, which retains all past versions of all articles, and a lot of researchers have studied the evolution of Wikipedia articles over time. Another important example is the web collection at the Internet Archive, consisting of more than 150 billion web pages that have been crawled since 1996 – or more precisely, 150 billion distinct versions, as many pages were repeatedly crawled over the years. Other examples of versioned collections are revision control systems for source code, document management systems, or versioned file systems that retain all past versions of the files.

Such versioned document collections differ from standard collections in several important ways. First, keeping all past versions results in much larger collections than keeping only the latest version. Second, different versions of the same document are often very similar, but exploiting this redundancy between versions to decrease inverted index size requires specialized index compression techniques. Third, the types of information requests that users have are also different. In particular, many queries have a temporal component, and may ask for relevant documents during a certain time interval. This is the main focus of this paper.

In particular, we are studying how to efficiently support search requests with a temporal restriction (temporal range queries) in versioned document collections. As discussed later, a trivial way to implement such queries is via post filtering, i.e., by traversing the index structures for the complete time range and discarding results outside the query range. Of course, one should hope to actually do much better, by enabling efficient access to only those subsets of the index structures that correspond to the query range.

However, techniques for enabling efficient access to time ranges, usually by partitioning or suitably reordering the index, can result in significant increases in total index size under state-of-the-art versioned index compression techniques, which in turn can slow down processing in both memory-based and disk-based system architectures. Thus, the main

SIGIR'11, July 24–28, 2011, Beijing, China.

challenge we address here is fast temporal range query processing in versioned document collections under state-of-the-art versioned index compression methods.

The remainder of this paper is organized as follows. We first give some background and discuss related work. Then Section 3 lists our main contributions. Section 4 discusses the data sets we use and provides some baseline experimental results. Sections 5 and 6 provide our main contributions, smart partitioning and index reorganization techniques that achieve improved performance for memory- and disk-based indexes. Section 7 shows how to efficiently implement the recently proposed Stable Top-k operator [22] on top of our index. Finally, Section 8 provides some concluding remarks.

2. BACKGROUND AND RELATED WORK

We now provide some background and discuss related work. We start with basics of inverted indexes, and then discuss versioned document collections, versioned index compression, and previous work on range queries.

2.1 Searching Document Collections

There is a vast literature on how to organize and query index structures for standard (i.e., non-versioned) textual document collections. This includes work on compressing index structures using specialized techniques, executing basic forms of Boolean and ranked queries over such collections, and implementing various early termination and tiering strategies that allow ranked query processing without complete index traversal. We refer to the survey of Zobel and Moffat [27] for an overview.

Inverted Indexes: Most current search engines use an *inverted index* structure to support efficient keyword queries [27]. 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. For each term t, the index contains an *inverted list* I_t consisting of a number of *index postings*. Each posting in I_t contains information about the occurrences of t in one particular document d, usually the ID of the document (the docID), the number of occurrences of t in d (the frequency), and possibly other information such as the locations of the occurrences within the document. We assume an index with docIDs and frequencies in this paper.

To a first approximation, ranked query processing in search engines involves traversal of the inverted lists for the query terms, and computation of a score for each document satisfying an initial Boolean filter, usually the AND or OR of the query terms. For efficiency, this score should be computable from the information stored in the inverted index, though in practice often additional features are fetched for a second round of scoring after winnowing of the initial set of candidates. Our techniques here do not depend on the precise scoring function.

Index Compression: The postings in each list are usually sorted by docID and then compressed using any of the techniques in the literature [27]. Most techniques first replace each docID (except the first) by the difference between it and the preceding docID, called *d-gap*, and then encode the d-gaps using a suitable integer compression method. The d-gaps and frequencies are often stored separately, and thus we compress sequences of d-gaps and sequences of frequencies. Inverted lists are usually organized into blocks that can be independently accessed, thus enabling forwards skips during query processing. We use such blocked indexes in our implementations, with a block size of 128 integers.

A large number of inverted index compression techniques have been proposed; see [27] for an overview. Here, we use two techniques that have been previously applied to versioned document collections [12, 13]: *Interpolative Coding* (IPC) [17] and the OPT-PFD method [23] from the PForDelta family of compression schemes [14, 28]. Both methods are especially suitable for compressing clustered (or bursty) sequences of integers, where there are runs of small numbers separated by a few larger numbers. IPC achieves the smallest compressed size but is somewhat slow in decompression, while OPT-PFD has a slightly larger compressed size but allows extremely fast decompression (up to a billion integers per second per core).

2.2 Versioned Document Collections

A versioned document collection D is a set of documents $d_0, \ldots d_{n-1}$, where each d_i has m_i versions $d_i^1, d_i^2, \ldots, d_i^{m_i}$. We assume a linear history, and do not try to model the case of branches (forks) in the revision history, though we believe most of our ideas could be adapted to this case. Each version has a lifespan associated with it, i.e., a time period where it is valid, and we assume that different versions of the same document have disjoint lifespans.

For an overview of the issues involved in building large scale web archival systems, see [21, 3]. There has been some amount of recent work on indexing and searching of versioned collections. Overall, this work can be split into three subsets, corresponding to different layers of a search architecture for such collections:

- Index compression techniques for versioned collections that exploit the similarity between different versions of a document [4, 15, 8, 25, 5, 12, 13], leading to significantly smaller index sizes and faster index traversal.
- Support for efficient temporal range (or point) queries over versioned collections through suitable index organization and index traversal [5, 2]. Here, the goal is to process queries with temporal constraints (say, for versions that were valid during a certain time interval) faster than a complete traversal of the index followed by subsequent filtering, by allowing efficient access to those index postings that fall into the right time range.
- Implementation of higher-level temporal operators, such as the Stable Top-k operator recently proposed in [22], on top of the index. The idea is that users may be interested in, e.g., documents that were relevant throughout the entire query time interval, or on average very relevant. In general, there may be some aggregation across different versions within the time interval to determine the most suitable documents and versions to return.

We focus here on the middle layer, supporting fast temporal range search. However, we also consider how this interacts with the other two layers. In particular, we will see that a naive approach for range search based on index partitioning can severely affect compressed index size, resulting in significant performance degradation. Thus, one main concern here is on supporting range search while preserving the effectiveness of the lower-level compression methods. We note that this challenge distinguishes the temporal range search problem on versioned document collections from that on non-versioned collections, such as news archives where each article has a publication time or a period of time when it is considered relevant but there is no versioning. (For completeness we also provide some limited experimental results for a non-versioned collection of articles from the New York Times.)

We also consider how to implement higher-level operators on our methods. Here, recent work in [22] proposed algorithms and specialized index structures for one such operator, Stable Top-k. We show in Section 7 that we can in fact implement this operator on top of our unchanged index organizations, with very little overhead, resulting in running times that are faster than those reported in [22] for a specialized index structure on a smaller collection.

2.3 Versioned Index Compression

An inverted index for a versioned collection should allow us to determine which versions of which documents contain a term, and how often. Versioned document collections often use index organizations and compression methods that are somewhat different from those used for standard collections. We now describe three inverted index compression schemes that we use, *Sorted*, *2-DIFF*, and *2R-MSA*, which were shown to achieve very fast query processing speeds in [12, 13].

The *Sorted* method is easy to explain: It simply indexes the versions as if they were separate documents, but assigns docIDs (or really version IDs) such that consecutive versions of the same document receive consecutive IDs. Then either IPC or OPT-PFD is used to compress the resulting inverted lists. This idea of better index compression through proper assignment of docIDs has recently been studied by a number of researchers for the case of standard collections [7, 18, 20, 6, 19, 23], and is also related to the lossy compression approach for versioned indexes proposed in [5].

To describe *2-DIFF* and *2R-MSA*, we need to introduce two-level indexes, first proposed for standard collections in [1], and then applied to versioned collections in [13, 12]. We define the first-level index of a versioned document collection D as an inverted index where the inverted list for a term t contains a posting for document d_i if at least one version d_i^j of d_i contains t. For a term t that occurs in at least one version of d_i, we define the bit vector of t and d_i as an array of m_i bits such that the j-th bit is set to 1 iff version d_i^j contains t. We also define a corresponding frequency vector that has one integer entry for each 1-bit in the bit vector (that is, each version d_i^j containing the term); this entry contains the frequency of the term in the corresponding version.

A two-level index structure for a versioned document collection consists of the first-level index plus a second level storing all the bit and frequency vectors in suitably compressed form. This allows efficient query processing by first running a query on the smaller first-level index, and then fetching and decompressing any necessary bit and frequency vectors. Note that the first level contains only docIDs, and no frequency values. It is compressed using standard index compression techniques such as IPC or PFD, since there is no obvious structure in the data that distinguishes this case from that of a non-versioned collection.

The 2-DIFF and 2R-MSA methods have the same first level structure, as described above, but use different techniques for compressing the vectors in the second level. This is achieved by a transformation that creates virtual document versions on which bit and frequency vectors are defined. For convenience, we assume each document has an initial empty version d_i^0.

Then in 2-DIFF, for each version d_i^j we define a virtual document version d'^j_i consisting of the symmetric difference between d_i^j and the previous version d_i^{j-1}. (Note that in the virtual version, some terms may have a negative frequency.) In 2R-MSA, we create a virtual version $d'^{j,k}_i$ for every $0 < j \leq k \leq m_i$, containing any terms that occur in all versions d_i^j to d_i^k but not in d_i^{j-1} or d_i^{k+1}. (Note that versions are bags, not sets, of terms.) We then keep all non-empty versions. In the case of 2R-MSA, we also reorder (renumber) the virtual versions in decreasing order of size.

Given these virtual versions, we create bit and frequency vectors as defined earlier, and finally we compress these vectors using either IPC or OPT-PFD. We note that 2-DIFF and 2R-MSA are two-level versions of techniques originally proposed in [4] and [15], respectively, with the additional reordering of virtual versions by size in 2R-MSA leading to better compression through clustering. The techniques we present in this paper also apply to other versioned index compression techniques (though the results may vary), and the details of the virtual document definitions above are not really crucial for understanding later sections (as long as the reader understands the concepts of levels, bit vectors, and frequency vectors). Finally, note that we limit ourselves to indexes storing docIDs and frequencies, and the problem of compressing versioned indexes with positions is quite different from the case without positions [25], as it requires working with substrings rather than bags of terms.

2.4 Temporal and Range Search

There is of course a lot of previous work on range search. This includes extensive work by the database community that is focused on numeric or string attributes but that does not look at very large sets of documents and inverted index structures. Work in IR includes support for numerical range constraints [11] (e.g., keyword queries on product catalogs with price or weight constraints), and techniques for geographic or local search [16, 26, 9, 10] (e.g., keyword queries on business listings with location constraints). The main difference is the presence of versioning in our problem, making the previous methods unsuitable.

We define temporal range search as the problem of returning the docIDs, version numbers, and scores of all versions satisfying the keyword as well as temporal range constraints. A simple baseline solution to this problem would first perform a Boolean filter to identify documents satisfying the keyword constraint, then check for the range constraint, and finally fetch the frequencies and compute the scores. In fact, we will implement this method in Section 4, and show that with state-of-the-art index compression and traversal techniques, such an approach is actually reasonably efficient for the collection sizes typically studied in the literature. Note that we do not consider early termination techniques; i.e., we limit ourselves to techniques that must score any version that satisfies the Boolean keyword and range constraints. This means that our results are largely independent of the choice of scoring function, as long as it can be computed from the index data. (We use BM25 in our implementation. Clearly, our methods can be extended to return top-k results, with essentially the same running times.)

However, we should expect to do better than the baseline by exploiting the temporal constraint, if we can organize the index in a way that allows fast access to postings that satisfy this constraint. This idea was evaluated in [5], leading to improvements over the base case through index partitioning along the time axis. However, their work used compression methods that lead to much larger index structures, and thus a slower baseline, than our work. We focus in particular on the trade-offs between index organization and index size for state-of-the-art index compression techniques. Another recent study [2] attempts to accelerate range search by trading off speed and recall; the goal is to retrieve most of the

relevant results by selectively searching in only a subset of the index partitions overlapping the query range. However, the results suggest that significant gains in speed can only be obtained with severe decreases in recall, and arguably the same or better trade-offs could also be achieved with existing standard early-termination techniques.

3. OUR CONTRIBUTIONS

In this paper, we describe and evaluate techniques for optimizing the performance of keyword queries with temporal range constraints in versioned collections. In particular, we make the following contributions:

- We evaluate a state-of-the-art implementation of simple baseline methods for searching versioned document collections, establishing the results that need to be beaten by more optimized methods.
- We describe and evaluate document and index partitioning techniques for main memory-based indexes that achieve improved performance over the baseline with only moderate increases in total index size.
- We also study how to organize index structures when the index is partially or completely on disk.
- We show that our methods are fast enough to efficiently support higher level operations, in particular the Stable Top-k operator [22], without the need for specialized index structures.

4. DATA SETS AND BASELINE METHODS

In this section, we perform a careful study of some baseline methods on several data sets. We first introduce the three document sets and the query traces that we used, then describe the baseline methods and their performance, and finish with a discussion of our observations.

4.1 Data Sets and Query Traces

We use versioned collections from Wikipedia and the Internet Archive, and a non-versioned collection with time stamps from the New York Times.

The Wikipedia data (Wiki) consists of 2,401,789 documents with a total of 85,352,299 distinct versions, i.e., about 35 versions per document on average. (We removed versions marked as minor edits.) This is a complete archive of English language Wikipedia articles from January 2001 to January 2008 (more than 1.5 TB in uncompressed form), more than 2 times larger than the set used in [22, 5] and about 10 times larger than that in [13, 12]. The Internet Archive data (IRE) consists of 1,056,981 web pages with 16,680,002 distinct versions from the Irish web domain collected from 1996 to 2006, i.e., about 15 versions per document. In our experiments, we assume that each version here is valid from the time it is fetched to the time a new, distinct version is crawled. The New York Times data (NYT) consists of 1,831,109 news articles from January 1987 to January 2007. Each article has a time stamp indicating the publication date. We assumed that every article had a lifetime of 90 days, as suggested in [2].

We point out that particularly for the Wiki and IRE data sets, versions are not equally distributed over time. In Figure 1 we show the total number of documents and terms (i.e., total postings) in the versions that were valid at certain points in time. We see that both collections experienced significant growth over the total time period, with more content towards the end of the period.

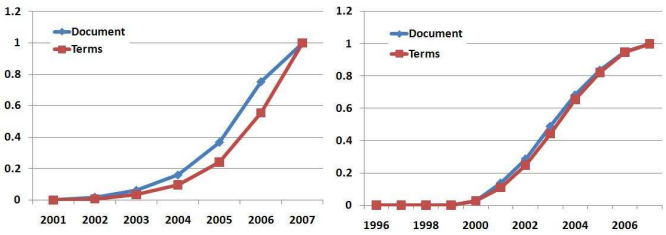

Figure 1: The number of documents and number of terms (i.e., total postings), based on all versions that were valid on a fixed day each year. We show results for Wiki (left) and Ireland (right).

Queries: For each of the three data sets, we compiled query traces using the AOL search query log. For Wiki, we extracted 10,000 queries where the user clicks on a result from the Wikipedia.org domain (following [13]). We also extracted 5,000 and 10,000 queries, respectively, where users clicked on results in the `ie` and `nytimes.com` web domains. Then we assign temporal ranges to these queries by choosing start and end times. The start time is chosen using two different random distributions, (a) uniformly from the total time interval, and (b) biased according to the document distribution shown in Figure 1 so that more queries focus on those times where most of the content resides. We then randomly select an end point such that the expected size of the range is equal to 7 days, 30 days, or one year, to model different granularities for the temporal constraint.

System Setup: All experiments were run on a single core of an Intel Xeon E5520 processor running at 2.26 Ghz. Disk costs were modeled assuming 8 ms access time for each separate access and 50 MB/s sequential transfer rate, which is typical of current low-cost SATA disks.

4.2 Non-Temporal Baseline

We built compressed versioned index structures containing docID and frequency data for our three collections, using existing components described in Section 2. For the non-versioned NYT data set, only the *Sorted* method is applicable; we assigned docIDs according to the time stamps of the articles. We also built a fast query processor using existing components in our group. This processor uses various state-of-the-art techniques such as block-compressed indexes and fast document-at-a-time index traversal. It can execute queries with and without temporal range constraints, and we will later extend it with various improvements. In the following, we provide some benchmark results for our processor that serve as a baseline for later sections.

Index Size: We start by reporting some results for index size and query processing without additional temporal constraints. We first look at the first level of the indexes for Wiki and IRE. In Table 1 we show the total size (in MB) and query processing speed (in millisecs) for the first-level index only, for Wiki and IRE and using IPC and OPT-PFD compression. Queries are chosen from the relevant traces but without range constraints, and we traverse the docID-only postings in the first level to report all documents that contain all query terms (intersection).

	Wiki	Ireland
IPC size (MB)	864	273
IPC speed (ms)	1.0	0.3
PFD size (MB)	1062	394
PFD speed (ms)	0.6	0.1

Table 1: Compressed sizes and query processing speeds for the first-level index structure.

We see from Table 1 that both index sizes and query pro-

cessing speeds for the first level are quite small. We also see that using IPC rather than OPT-PFD results in decreases in first-level index size, but at the cost of disproportional increases in time. Thus, in the remainder, we choose OPT-PFD for both levels of our index structures. (Results in [13] already suggest using OPT-PFD for the second level.)

Next, we look at total index size (in MB) for the three collections under the different compression schemes, shown in Table 2. We see that, as should be expected from [13], 2-DIFF and 2R-MSA achieve a significantly smaller compressed index size than Sorted, with 2R-MSA doing slightly better than 2-DIFF. In particular, for Wiki with 85 million versions and more than 1.5 TB of uncompressed (though highly redundant) data, we get a complete index size of only about 4 GB, thus allowing the index to be kept completely in main memory on many current machines.

	docID	freq	total
Wiki Sorted	5508	8364	13872
Wiki 2-DIFF	3802	1109	4911
Wiki 2R-MSA	2963	1104	4067
IRE Sorted	1137	856	1993
IRE 2-DIFF	894	282	1176
IRE 2R-MSA	780	170	950
NYT Sorted	453	135	588

Table 2: Compressed index size in MB for docIDs and frequencies using the three compression methods on our data sets.

Non-Temporal Queries: Next, we report results for query processing without temporal range constraints on the full indexes, shown in Table 3. We report the CPU time of query processing (on a single core) assuming the index is in main memory, as well as the disk access and transfer times involved if all inverted lists are located on disk and need to be fetched first (we look at caching of index data further below). For the disk numbers, we use disk access times of 8 ms and a transfer rate of 50 MB/s, and we assume that the first and second levels of each inverted list are next to each other so they can be read with a single seek. (Thus, the access time divided by 8 ms gives the average number of terms per query for each trace, as every list requires a separate access.)

	cpu	disk access	disk transfer	total
Wiki Sorted	18.2	17.4	117	151.1
Wiki 2-DIFF	26.3	17.4	37.2	80.9
Wiki 2R-MSA	27.1	17.4	36.9	81.4
IRE Sorted	0.84	22.9	12.2	35.9
IRE 2-DIFF	1.31	22.9	6.2	30.4
IRE 2R-MSA	1.43	22.9	5.4	29.7
NYT Sorted	0.4	28.2	12.0	40.6

Table 3: CPU and disk cost per query (in ms) under different compression schemes for our three data set.

As we see from Table 3, query processing in main memory is quite fast, with Sorted achieving a time of 18.2 ms on the large Wiki data set. However, Sorted results in much larger index sizes, and this shows itself in the significant costs for retrieving data from disk that make this the slowest method in total cost. For the smaller IRE data set and for NYT, the disk access cost is dominated by access times. But even for Wiki under 2-DIFF and 2R-MSA, access costs are significant compared to transfer costs, implying that we have to avoid optimizations that decrease data transfers at the cost of additional random accesses.

4.3 Basic Temporal Query Processing

We now look at what happens once we add temporal constraints to the queries (while not changing the index organization). We create an additional (relatively small) memory-based table that contains the time ranges of the different versions, allowing efficient lookup given an index posting. We then have two ways to use this table to check the range constraint: We can either first perform the intersection between query terms and then do a lookup for each qualifying version (intersect-first), or we can do a lookup for every posting in the shortest list and then do the intersection with the other terms (check-first), which decreases the cost of the intersection at the cost of more lookups.

In Table 4, we show CPU costs on the three data sets and different compression schemes, assuming a uniform query distribution. In addition to check-first and intersect-first, we also show a method called best-of-both that assumes that each query uses the best approach, giving us an upper bound on what can be achieved by choosing the best for each query. We note that in some cases we see slight improvements over the non-range case, since we do not have to fetch frequency values and compute scores for versions that do not satisfy the range constraint. On the other hand, the cost of doing the simple lookup into the global table is also significant, in some cases more than wiping out the savings. Overall, numbers are similar to the non-range case.

	check-first	intersect-first	best-of-both
Wiki Sorted	16.7	11.2	10.1
Wiki 2-DIFF	29.7	28.1	27.5
Wiki 2R-MSA	30.4	29.2	28.4
IRE Sorted	0.80	0.72	0.7
IRE 2-DIFF	1.32	1.13	1.08
IRE 2R-MSA	1.45	1.33	1.31
NYT Sorted	0.38	0.31	0.29

Table 4: CPU cost of query processing (in ms) under uniform query distribution with range size 30 days.

	bias		uniform	
	2-DIFF	2R-MSA	2-DIFF	2R-MSA
One Week	19.5	20.1	14.8	15.1
30 days	19.9	20.7	15.1	15.4
One Year	22.9	25.8	17.7	21.3

Table 5: CPU costs of query processing with first-level range check, under biased and uniform query distribution for different range sizes.

There is one additional optimization that we can add for two-level methods, without any changes in index organization. As shown in Figure 1, many documents do not exist yet at the beginning of the timeline, as they are only added or crawled later. Thus, after traversing the first level of the index, we could check the documents retrieved in this phase to see if their total lifetime overlaps with the query interval, thus eliminating some documents created only later without fetching the second-level bit vector. However, this means that running times depend more heavily on the query distribution, and in particular on the choice of uniformly chosen versus biased starting point. The uniform model would create a lot of queries that are early in time and thus many documents can be ruled out after the first phase, resulting in faster query processing.

In Table 5 we show CPU costs with this optimization, for queries with expected query range sizes of 7 days, 30 days, and one year, under the uniform and biased query model. We see that the optimization leads to significantly decreased costs for the uniform model, compared to the previous tables. Costs for the biased model are higher, as in every phases of the computation more candidates survive the various range tests. However, we feel that given the distribution shown in Figure 1, the biased model is more appropriate, and thus we choose this model in the following sections.

4.4 Impact of List Caching

So far, we have assumed that the index structure is either completely in main memory, or only on disk. However, this second case is not really realistic for current search systems, which usually cache at least some of the inverted lists in main memory even when the complete index does not fit. Caching is known to provide significant performance boosts [24]. To determine the impact of caching, we implemented a simple caching scheme that selects a static subset of lists to be kept in memory. Lists are selected based on their frequency in a large query trace (disjoint from the actual queries used to measure CPU costs), and based on their size, as small lists are slightly more preferable due to the high cost of random accesses on disk.

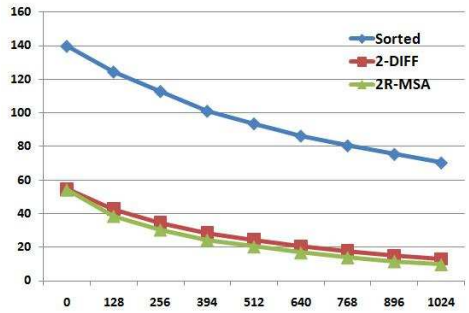

Figure 2: Disk costs in milliseconds per query for different compression methods on the Wiki data set, as cache size is varied between 0 and 1024 MB.

The result are in Figure 2, which shows that disk costs decrease significantly with cache size. With cache size 1024 MB (about 20 to 25% of total index size), the disk cost for 2-DIFF and 2R-MSA has decreased to about 20% of the cost without caching. Also, notice that the relative performance gap of 2-DIFF and 2R-MSA versus Sorted increases as cache size increases – better compression means a higher cache hit rate since a higher percentage of the index fits in the cache, and also less data to fetch when there is a miss. One more observation, not shown due to space constraints, is that the relative cost of disk access times (for seeking the start of an inverted list on disk) increases compared to the transfer cost as cache size increases.

4.5 Discussion

We now briefly discuss the main observations and lessons from this section. First, we observed that using state-of-the-art index compression and traversal techniques, even a baseline query processor is quite efficient, achieving less than 30 ms per query for a memory-based index, and about 80 ms per query for a disk-based index, on the 85 million versions of the Wiki data set. Second, proper compression using two-level techniques makes a significant difference in disk access costs and total index size (the amount of memory we need to buy for the in-memory case). Third, even a small cache in main memory significantly decreases disk access costs. Fourth, even for fairly large data sets, the costs for initiating disk accesses are significant in comparison to the transfer time for sequential reads, particularly with caching. This implies that when we reorganize the index in the next sections to get better performance, it is usually not a good idea to split inverted lists into several pieces that require separate disk reads. Finally, given the results here, we should not expect further improvements by an order of magnitude or more – in the case of a disk-based index, one would expect running times to be lower bounded by $h \cdot t \cdot a$, where h is the cache miss rate (1.0 if there is no caching), t is the average number of terms in a query, and a is the access time (8 ms in our case), assuming a separate seek is needed for any term resulting in a cache miss.

In the next two sections, we will design improved index organizations that decrease both CPU and disk costs. We use the biased query model, and we limit ourselves to the 2-DIFF and 2R-MSA methods as they are much more promising than Sorted.

5. CUTTING DOCUMENTS AND COSTS

We now study how to further decrease the CPU costs of range queries, by suitably splitting documents. Recall that in the previous section, we introduced a first level range check that rules out documents whose lifetime does not intersect the query range. However, the benefits are limited by the properties of the data set. Now suppose we split a document with v versions into two subdocuments, one consisting of the first $v/2$ versions, and the other consisting of the rest. This will increase the size of the first-level index, and the time spent in the first level, as we now have more documents and more postings. It will also increase the size of the second-level index, as we get more, but shorter, bit and frequency vectors that are overall less compressible than the longer vectors we had before (since, in a nutshell, we lose the redundancy between consecutive versions that are separated by subdocument boundaries). However, on the plus side, the first level range checks in our query processor will filter out many more subdocuments that do not overlap the query range.

Thus, we can trade off index size for increased speed, and the question is how to best perform these cuts. One approach is to pick a number of versions s, and greedily partition each document into subdocuments with (at most) b versions each; we call this approach *fix-bits*. However, this means that documents with many versions get split into a large number of subdocuments that each cover only a short period of time, while documents with only a few versions are not cut at all. Decreasing b will exacerbate the first problem, and increasing the latter. Or we can partition by time, and greedily add versions to a subdocument until its lifetime exceeds some fixed amount of time t; we call this *fix-time*. However, this can result in very small subdocuments with only one or two versions for documents with few versions overall, significantly increasing overall index size unless t is chosen quite large. We note that *fix-time* is similar to the time-based index partitioning scheme in [5]; however, we partition each document independently without ever duplicating a version in two partitions. We discuss their method in more detail later.

Finally, we consider a third method that combines these two in an elegant way. If we make the simplifying assumption that query ranges are uniformly chosen (not biased, as we actually use), then the likelihood of a subdocument passing the first level check is roughly proportional to its lifetime.[1] If we assume that the cost of accessing a bit vector in the second level is proportional to its length (number of bits), then the expected cost for the bit vectors of a subdocument, given a query, is proportional to the product of its length in bits and its lifetime. To minimize this, we should chop whenever this product exceeds some threshold p, where the choice of p will determine the blowup in index size. This method avoids the problems of the other schemes discussed above, in that it chops documents with many versions into

[1]Strictly speaking, this depends on both lifetime and query range size.

bigger pieces (but not too big) and documents with fewer versions into smaller pieces (but not too small). We call this *smart* partitioning.

Integrating this into our query processor is easy. We just need to determine the best settings for the parameters b, t, and p in the above methods. In Figure 3 we show the resulting trade-off between total compressed index size and the CPU cost for 30-day range queries, for 2-DIFF and 2R-MSA compression. Figure 4 shows the same for the IRE data. We note that the *smart* partitioning obtains the best trade-off in all cases, with the advantage over the other two methods more pronounced (but absolute improvements more limited) on the IRE data set. We also see that when we make too many partitions, both CPU cost and size increase. For 2-DIFF on Wiki, we see that increasing the index size from 4.9 GB (no chopping) to 5.3 GB already brings down CPU cost from more than 20 to less than 7 ms per query, a very significant improvement.

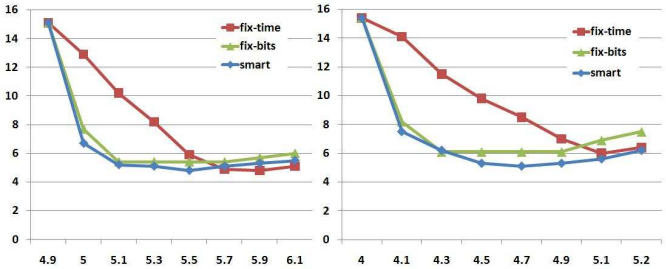

Figure 3: CPU cost per 30-day query under three different document partitioning schemes (in ms) on the y-axis versus total index size (in GB) on the x-axis, on the Wiki data set, for 2-DIFF (left) and 2R-MSA (right).

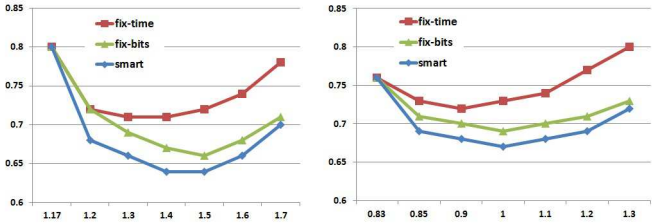

Figure 4: Same as previous figure, on the IRE data set.

We select the parameter p for the smart partitioning that minimizes query processing time. In Table 6, we show the resulting total index sizes and disk costs, plus CPU costs for query ranges of 7 days, 30 days, and one year. We note here that the disk cost is without any caching, that it is independent of query range size as we always fetch the entire inverted list from disk, and that there is a slight increase in disk cost. Overall, we get significant improvements in total cost over Table 5 when the index is in memory, and moderate improvements when it is on disk.

	Wikipedia		Ireland	
	2-DIFF	2R-MSA	2-DIFF	2R-MSA
One Week	6.5	7.3	0.61	0.64
30 days	6.6	7.4	0.62	0.66
One Year	13.8	14.9	0.82	0.85
index size	5532	4727	1418	1024
disk cost	59.1	55.9	29.2	28.9

Table 6: Total index size, disk cost, and CPU cost for various query sizes, under the smart partitioning that minimizes CPU cost.

6. OPTIMIZING DISK ACCESS

In the previous section, we studied methods that decrease the CPU cost of queries through document partitioning. However, the methods do not decrease disk costs, as we still have to fetch the entire inverted list for a query term; in fact, disk cost increased slightly with total index size. In this section, we present methods that optimize disk costs in the case where the index does not fit into main memory. To do so, we need to find ways to partition documents and organize the resulting postings, such that all index postings that intersect with a query range are located in one part of the inverted list; this way, when a list is not in memory, we only need to fetch that part (reducing transfer but not seek time).

6.1 Index Organization Methods

We now describe the different methods that we considered, including the one proposed in [5] and several new ones.

Interval partitioning: This is a version of the technique in [5], which uses a partitioning of the timeline into intervals. For example, for a period of 10 years, we might create 10 partitions, one for each year. We then take the documents and cut them into subdocuments that each fit into one of the partitions. Of course, there will be versions whose lifespan is split between two or more intervals; these will be replicated in each partition, with its lifespan pruned to the interval boundaries. We then assign document IDs to subdocuments such that consecutive docIDs are used within a partition; this organizes each inverted list into partitions that can be accessed separately. It is important to keep each list together in one place on disk – a method that builds independent inverted indexes for each partition would result in much higher disk access costs when queries overlap more than one interval.

Our implementation of this method uses two-level compression and the various other optimizations in our query processor, and thus we would expect faster results than in [5]. However, the method suffers from the same drawbacks as the *fix-time* method in the previous section in that it uses a one-size-fits-all approach for all documents. In addition, the replication of some versions in several intervals will further increase index size. On the plus side, given a query we can fetch all necessary parts of the inverted list in one disk access, by fetching only those partitions of the list with intervals overlapping the query range.

Stencil-Based Methods: To overcome the drawbacks of the *fix-time* and *interval partitioning* methods, we considered using a multi-level hierarchical partitioning of the timeline. Here, level L_0 contains the entire timeline, and we obtain level L_{i+1} from L_i by partitioning each interval in L_i into b subintervals, for some b. Figure 5 shows an example of a 4-level partitioning with $b = 2$ and a timeline of 32 days. We call such a partitioning a stencil; given this we now partition our documents so that each resulting subdocument resides in an interval that fits its range.

We look at two partitioning methods that use the *smart* approach from the previous section. The first one, *partition-first*, just uses the *smart* partitioning to make subdocuments, and then puts each subdocument into the deepest interval in the multi-level partitioning that it fits in. (Thus, we first cut, and then throw into the intervals, with no replication.)

In the second approach, *stencil-first*, we recursively cut documents into subdocuments using the stencil, as follows: Given our current subdocument, we find the deepest interval that it fits in, probably L_0 in the beginning. We then check if the product of lifespan and number of versions is less than p, where p is the parameter for the smart method. If yes, we do not cut it further, and keep it in the current interval. If no, then we cut it into subdocuments that exactly fit into the children of the current interval, and continue. In this

case, we need to replicate versions that cross boundaries, as in the single-level case, and we get a more snug fit than in the partition-first case. (Also note that we keep cutting until the product is less than p, while in *smart* we cut as soon as it becomes larger than p; in both cases the idea is to get a cut with product close to p, but now we may have to use a slightly larger p.)

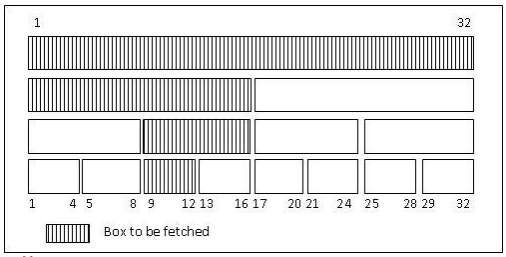

Figure 5: A 4-level stencil with $b = 2$ for a timeline of 32 days. The four shaded partitions need to be fetched for a query with range [9, 11].

Finally, we lay out each inverted list such that the partitions are ordered by an in-order traversal of the multi-level structure. We note that this places the partition corresponding to L_0 in the center, and this partition has to be fetched for any incoming query. Figure 5 shows the four partitions that are fetched for a query with range [9, 11].

Sort-Based Approaches: The next approaches are based on the idea of ordering subdocuments in the lists by assigning docIDs temporally. This is easy to do when each document has a single timestamp or a fixed lifespan t, as in the case of the NYT collection where every article was assumed to be valid for 90 days. If we then assign docIDs by start time, then for a range query $[l, r]$, we only need to fetch and traverse postings for documents with start times between $l - t$ and r, which are located in one consecutive area of the list. However, in our Wiki and IRE data sets, each version has a different lifespan, and of course we cannot afford to store each version by itself, as this would result in a much larger index size. Thus, our goal now is to again partition the documents into subdocuments, such that we can apply the above idea to decrease the amount of list data that we need to retrieve from disk. We look at three policies:

Time-sort is similar to *fix-time* in the previous section. Given a choice of t, we greedily cut into subdocuments with lifespan at most t. If a single version has lifespan larger than t, then we cut and replicate this version. After assigning docIDs based on the start time of each subdocument, given a range query $[l, r]$ we only have to fetch postings with start times between $l - t$ and r.

Smart-sort is based on the *smart* partitioning method from the previous section, that is, we greedily cut whenever the product of lifespan and number of versions in the subdocument becomes larger than some bound p. Then we assign docIDs in the same way as in time-sort. A problem with this is that there is no absolute upper bound t on lifespan, thus making it impossible to translate an incoming range query into a small subset of postings, and this means that usually the entire inverted list will be fetched. We can improve CPU cost by storing for each block of 128 docIDs in the first level of the index the maximum end time of any document in the block. This allows skipping of many such blocks.

Hybrid-sort partitions documents greedily as soon as either of the conditions in time-sort and smart-sort becomes true. Here the idea is to choose t and p such that usually the smart condition will make the cut, with the constraint based on t being triggered as a fuse when the lifespan of a subdocument becomes too large. Thus, we hope to preserve the advantages of smart partitioning while also keeping an absolute upper bound on document lifespan that can be used to fetch only relevant parts of a list from disk.

6.2 Experimental Evaluation

We now evaluate the index size, CPU cost, and disk cost of the six methods, interval, partition-first, stencil-first, time-sort, smart-sort, and hybrid-sort. We use the 2-DIFF compression method, though results for 2R-MSA are similar.

	Wikipedia			Ireland		
	cpu	disk	total	cpu	disk	total
stencil-first	11.4	41.6	53	0.75	27.9	28.7
part.-first	5.6	38.6	44.2	0.56	24.6	25.2
interval	5.9	34.5	40.4	0.57	25.4	26
smart-sort	4.6	61.6	66.4	0.55	32.1	32.7
time-sort	5.8	31.9	37.7	0.51	24.6	25.1
hybrid-sort	4.3	30.3	34.6	0.47	24.5	25

Table 7: CPU costs and disk costs (in ms) for the various methods with best parameter settings, for 30-day queries and 2-DIFF compression.

	Wikipedia			Ireland		
	docIDs	freq	total	docIDs	freq	total
stencil-first	4347	1649	5996	1055	557	1612
part.-first	5677	1365	7042	1552	405	1957
interval	6771	2312	9083	1717	524	2241
smart-sort	4890	1319	6209	1017	413	1430
time-sort	5365	1766	7131	1234	518	1752
hybrid-sort	5300	1738	7038	1224	515	1739

Table 8: Index size (in MB) for the various methods on Wiki and IRE data, using 2-DIFF compression.

Overview of Results: We start with the CPU cost, disk cost (without caching), and total cost per 30-day query, shown in Table 7. The numbers given are for the best choices of various parameters; we investigate some of these choices later. We observe that all methods except smart-sort achieve significant reductions in disk costs versus the earlier sections. The best results are obtained by time-sort and hybrid-sort, followed by interval, and then the multi-level methods partition-first and stencil-first. The interval method suffers from a fair amount of version replication that increases total index size (as shown later). The multi-level methods are limited by the need to fetch the postings located in higher-up intervals in the structure, in particular the root L_0 that is in the middle of the inverted list.

Note that we use a single disk access to fetch all needed partitions of an inverted list; this is the best for our data sets. To do so, we interleave the two index levels. This is simple for stencil and interval, which assign subdocuments to discrete subsets: We just concatenate the first and second level of each subset. For the sort-based methods, we have a slightly more complicated structure. Consider time-sort with parameter t: We store inverted lists in blocks of 128 integers, and interleave blocks from both levels by ordering them by minimum start time. We group the resulting sequence of blocks into chunks of x blocks, and create a table storing for each chunk the minimum start and maximum start of any subdocuments in it. For range query [l,r], we fetch in one access all chunks with minimum start larger than r and maximum start larger than $l-t$ (including any non-qualifying chunks in-between). We define z as the maximum number of first-level blocks between any consecutive second-level blocks. We choose $x >= z$. In all our data we observe $z <= 25$, and we set $x = 40$ in our experiments. Thus, each chunk has at least one first-level and one second-level block. For large z, replicating selected first-level blocks leads to even better results.

It is possible, however, that using more than one disk access in the multi-level methods would result in better performance for even larger index sizes (due to either a larger document collection or inferior index compression techniques).

	Wikipedia			Ireland		
	7	30	365	7	30	365
stencil-first	41.4	41.6	43.6	27.8	27.9	28.2
part.-first	38.3	38.6	42.9	24.4	24.6	25
interval	33.9	34.5	39.5	25.3	25.4	25.7
smart-sort	61.6	61.6	61.6	32.1	32.1	32.1
time-sort	30.6	31.9	36.6	24.6	24.6	25.1
hybrid-sort	29.6	30.3	35.1	24.5	24.5	24.9

Table 9: Disk cost per query for 7-day, 30-day, and one-year queries, on Wiki and IRE with 2-DIFF compression.

	Wikipedia			Ireland		
	7	30	365	7	30	365
stencil-first	10.8	11.4	16.3	0.74	0.75	0.91
part.-first	5.1	5.6	11.1	0.54	0.56	0.81
interval	5.4	5.9	13.7	0.52	0.57	0.84
smart-sort	4.3	4.6	8.9	0.53	0.55	0.86
time-sort	5.2	5.8	11.9	0.49	0.51	0.85
hybrid-sort	3.9	4.3	10.6	0.46	0.47	0.82

Table 10: CPU cost per query for 7-day, 30-day, and one-year queries, on Wiki and IRE with 2-DIFF compression.

Looking at CPU costs in Table 7, we note that most of the techniques also outperform the results in previous sections. In particular, the methods using smart partitioning (smart-sort and hybrid-sort) do extremely well, probably due to a more clustered access pattern into the index during query processing resulting in fewer blocks of OPT-PFD-compressed data to be uncompressed on both index levels.

Next, we look at compressed index size for the various methods, shown in Table 8. We see that interval has the largest index size on both data sets, while stencil-first has the smallest on Wiki (but high CPU and disk costs). All the sort-based methods also perform well. Overall, the results of the two tables suggest that hybrid-sort is the best overall performer, followed by time-sort. However, smart-sort is a good choice when the index is completely in main memory, having lower CPU cost than the smart partitioning in Section 5 with slightly larger index size.

Now we look at how disk costs vary as range query size changes from 7 days to 30 days to 365 days, shown in Table 9. Note that the index structures were optimized for the case of 30 days. As such, we cannot expect queries to become much faster for smaller query ranges. On the other hand, cost increases only moderately for one-year queries. Also, note that for IRE, costs are almost independent of range size as disk costs are dominated by random access times. Corresponding results for CPU cost are shown in Table 10.

Choosing Parameters: Next, we investigate some of the parameter choices we made in the above tables in more detail. For the stencil-based methods, we had to choose the base b and depth d of the multi-level partitioning. We tried various values of b and the simple case of $b = 2$ did overall quite well. In Figure 6 we show the performance of the stencil-first and partition-first methods as we vary the depth d; results suggest choosing 4 or 5 levels for the larger Wiki set and 3 levels for IRE. In Figure 7 we see the total cost as we change the parameter t (the maximum lifespan of a subdocument) used in three methods. We note that t should ideally be chosen somewhat larger than the query size (30 days in this case). For IRE, it can be chosen much larger, but note that the overall timeline in IRE is more than ten years, with fewer versions per document than Wiki.

Impact of Caching: Recall that in real systems, even if the index does not fit in main memory, some significant

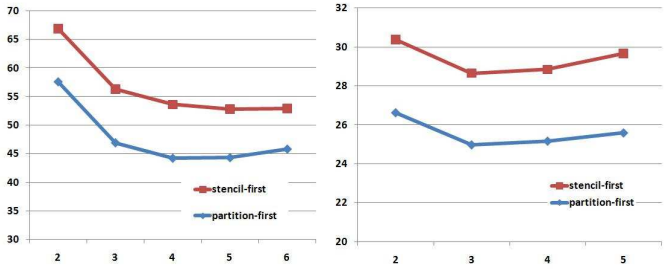

Figure 6: Total cost (in ms) per query for partition-first and stencil-first on the y-axis, as we vary the depth of the partitioning on the x-axis, for 30-day queries for Wiki (left) and IRE (right).

amount of memory is available for list caching, and we saw in Section 4 that this significantly reduces disk costs. In Figure 8, we compare the complete cost per query (CPU cost plus disk cost) of all the algorithms in this section on different cache sizes. We see that, as expected, all methods obtain significant benefits from caching. Overall, hybrid-sort performs best, followed by time-sort and interval. Note that for a cache size of 1024 MB, realistic on most current machines, hybrid-sort is more than three times as fast as the baseline 2-DIFF method from Section 4.

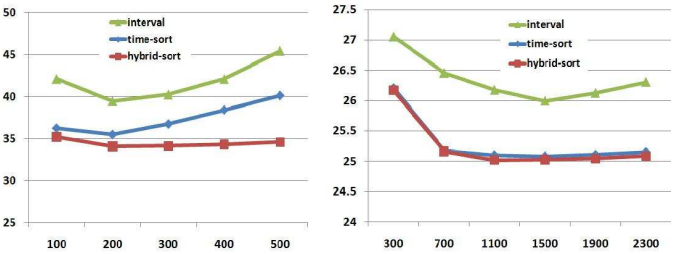

Figure 7: Total cost (in ms) per query for interval, time-sort, and hybrid-sort on the y-axis, as we vary the maximum lifespan (in days) of a subdocument on the x-axis, for Wiki (left) and IRE (right).

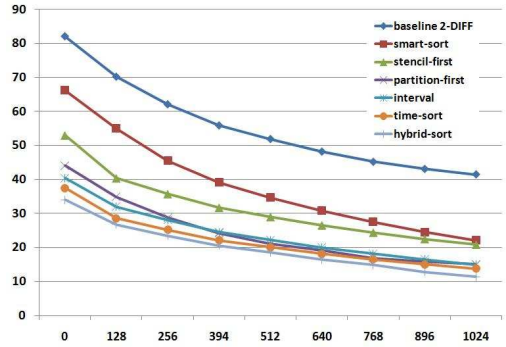

Figure 8: Total cost (CPU plus disk, in ms) per query on the y-axis, for cache sizes ranging from 0 to 1024 MB on the x-axis.

7. AGGREGATE QUERY PROCESSING

Recent work in [22] proposed a new aggregate operator for versioned document collections called Stable Top-k. Given a keyword query, a query range, and a value x, this operator returns all documents that were in the top-k results during at least x% of the query range. In other words, were are looking for documents that were among the most relevant during most of the query range. This is an example of an operator that performs an aggregate computation over several versions that were valid during the query interval; other examples might be documents that are ranked high on av-

erage, or documents that scored above a certain threshold score most or all of the time.

U et al. [22] proposed several algorithms for this problem, with the fastest one using an specialized index structure based on R*-trees. In order to explore the efficiency of our range search methods, we decided to implement Stable Top-k on top of our generic mechanism. That is, we use our methods to find all versions that satisfy the keyword and range constraints, and then run a separate (but fairly simple) Stable Top-k algorithm on this result set, which typically consists of a few thousand or few ten thousand versions.

In particular, we implemented a version of the Top-k Bands algorithm in [22], and ran it for different query ranges on Wiki. The results are in Table 11, where we show the additional cost (in ms) for Top-k Bands. We see that this cost is in fact rather small. As it is independent of the method used in the range search, we can get the complete cost by adding the best numbers from this paper. Thus, if the index is completely in main memory, we need to add 4.3 ms for 30-day queries (Table 10), and if it is completely on disk, we need to add 34.4 ms (Table 7). Overall, these results seem to outperform those for the specialized methods in [22], on a data set that is significantly larger. We conclude that generic range search methods can indeed efficiently support aggregate operators such as Stable Top-k.

	stable top-10	stable top-100
7 days	0.6	0.7
30 days	1.3	1.5
1 year	9.5	10.8

Table 11: Additional cost (in ms) of running Top-k Bands on top of our range search, for different query sizes and for $k = 10$ and $k = 100$.

8. CONCLUDING REMARKS

In this paper, we have described and evaluated techniques for efficient temporal range queries in versioned document collections. We evaluated some simple methods to provide a realistic baseline for improvements, and then showed how to decrease CPU and disk costs through suitable index organizations. Finally, we showed that our techniques are fast enough to efficiently support classes of aggregate queries over such collections.

There are several interesting problems that are left open by us. One is about partitioning methods that attempt to minimize index size by maximizing similarity within subdocuments. When cutting a document, we essentially waste the similarity between the two versions separated by the cut. This suggests that we should try to cut whenever a document undergoes significant change. However, this needs to be balanced with the other concerns, the number of versions and the lifespan of the resulting subdocuments, and preliminary experiments by us did not give any nontrivial benefits. A more careful partitioning method based on a suitable model might give some improvements.

While we focused here on versioned collections, we believe that improvements in practice are also possible for non-versioned collections where each document has a timestamp or lifespan associated with it. Finally, we did not consider early-termination techniques for versioned collections, which could compute top-k results without scoring all versions satisfying the keyword and range constraints.

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.

9. 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 queries over versioned text. In *Proc. of ACM CIKM Conf.*, 2010.

[3] A. Anand, S. Bedathur, K. Berberich, R. Schenkel, and C. Tryfonopoulos. EverLast: a distributed architecture for preserving the web. In *Proc. of ACM/IEEE JCDL Conf.*, 2009

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

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

[6] R. Blanco and A. Barreiro. Document identifier reassignment through dimensionality reduction. In *Proc. of European Conf. on Information Retrieval*, 2005.

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

[8] A. Broder, N. Eiron, M. Fontoura, M. Herscovici, R. Lempel, J. McPherson, R. Qi, and E. Shekita. Indexing shared content in information retrieval systems. In *Proc. of EDBT Conf.*, 2006.

[9] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *Proc. of ACM SIGMOD Conf.*, 2006.

[10] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. In *Proc. of Conf. on Very Large Data Bases*, 2009.

[11] M. Fontoura, R. Lempel, R. Qi, and J. Zien. Inverted index support for numeric search. In *Internet Mathematics*,3(2), 153-185, 2006.

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

[13] J. He, J. Zeng, and T. Suel. Improved index compression techniques for versioned document collections. In *Proc. of ACM CIKM Conf.*, 2010.

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

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

[16] C. B. Jones, A. I. Abdelmoty, D. Finch, and G. Fu. The spirit spatial search engine: Architecture, ontologies and spatial indexing. In *Proc. of Conf. on Geographic Information Science*, 2004.

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

[18] 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.

[19] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of European Conf. on Information Retrieval*, 2007.

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

[21] S. Song, J. Jaja. Archiving Temporal Web Information: Organization of Web Contents for Fast Access and Compact Storage. In Technical Report UMIACS-TR-2008-08.

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

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

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

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

[26] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *Proc. of ACM CIKM Conf.*, 2005.

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

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