Temporal Index Sharding for Space-Time Efficiency in Archive Search

Avishek Anand
Srikanta Bedathur
Klaus Berberich
Ralf Schenkel

MPI–I–2011–5–001 June 2011

Authors' Addresses

Avishek Anand
Max-Planck-Institut für Informatik
66123 Saarbrücken
Germany

Srikanta Bedathur
Max-Planck-Institut für Informatik
66123 Saarbrücken
Germany

Klaus Berberich
Max-Planck-Institut für Informatik
66123 Saarbrücken
Germany

Ralf Schenkel
Saarland University
66123 Saarbrücken
Germany

Abstract

Time-travel queries that couple temporal constraints with keyword queries are useful in searching large-scale archives of time-evolving content such as the web archives or wikis. Typical approaches for efficient evaluation of these queries involve *slicing* either the entire collection [19] or individual index lists [9] along the time-axis. Both these methods are not satisfactory since they sacrifice compactness of index for processing efficiency making them either too big or, otherwise, too slow.

We present a novel index organization scheme that *shards* each index list with *almost zero increase in index size* but still minimizes the cost of reading index entries during query processing. Based on the optimal sharding thus obtained, we develop a practically efficient sharding that takes into account the different costs of random and sequential accesses. Our algorithm merges shards from the optimal solution to allow for a few extra sequential accesses while gaining significantly by reducing the number of random accesses. We empirically establish the effectiveness of our sharding scheme with experiments over the revision history of the English Wikipedia between 2001-2005 ($\approx$ 700 GB) and an archive of U.K. governmental web sites ($\approx$ 400 GB). Our results demonstrate the feasibility of faster time-travel query processing with no space overhead.

Keywords

Time-Travel Text Search, Sharding, Slicing, Inverted Index, Web Archives

Contents

1 Introduction

Due to the ubiquitous access and ease of location using search engines, much of the human-generated content is available electronically over the Web. This content is constantly evolving with additions, deletions and modifications occurring at very high rates [4, 11, 12, 18]. Search engines continuously crawl and index the Web to keep up with its latest state.

There is a growing awareness, however, that this single-minded pursuit for the latest state of the Web results in losing access to the history of content which could be important in a number of advanced applications. As an answer to this, efforts such as the Internet Archive [3] and European Archive [2] store regular crawls of large portions of the Web for retrospective analysis. Effective querying over these collections requires the use of so-called *time-travel queries* that combine temporal constraints with standard keyword queries. These queries aim to return relevance ordered lists of documents (or, more precisely, document versions) that were "alive" during the specified time interval [9]. For example, on an archive of U.K. government websites, one may pose a query such as inheritance tax @ [2000-2002] to locate documents about inheritance taxation details prevalent during the years 2000-2002. Note that the same query without a temporal constraint would tend to generate results only about latest taxation details and not necessarily from the required time.

Efficient evaluation of these time-travel queries over archives of large collections (such as the Web) is a challenge that we address in this article. Earlier efforts combined the ideas of index organization in temporal databases with inverted indexes [6, 9, 19], which essentially meant that the time-enriched inverted indexes are *sliced* along the time-axis (or partitioned vertically). Each resulting partition of the index list contains index entries of only those document versions that have a *valid-time interval* overlapping with the time interval of the partition. Answering a time-travel query in such a temporally sliced index involves choosing the correct –much smaller– index partition (or partitions) to evaluate the query.

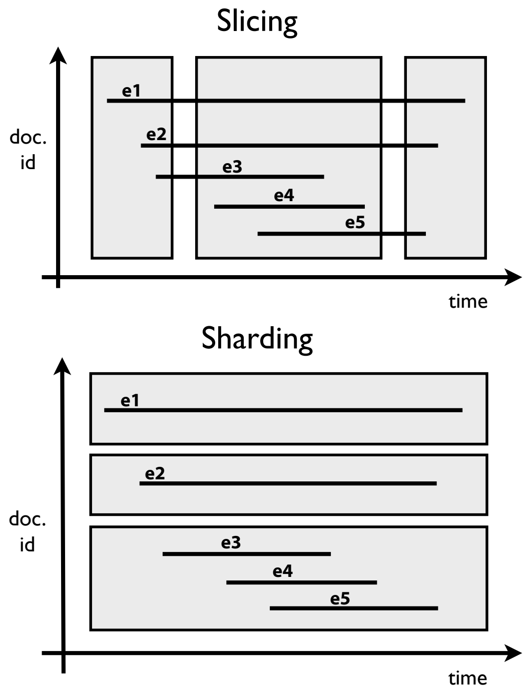


Figure 1.1: Slicing vs. Sharding of a Index list

Though attractive, such an index organization suffers from an index-size blowup incurred during the slicing process, since index entries whose valid-time interval spans across the slicing boundary are replicated. While a careful choice of the slicing boundaries can help to reduce the index blowup [9] in order to achieve acceptable levels of efficiency, 2 to 3 times index size increase is necessary.

1.1 Our Approach

In this article, we rotate the "knife" of partitioning by 90° and propose to *shard* –or horizontally partition– *each index list* along document identifiers, instead of time (cf. Figure 1.1). An immediate benefit of this reoriented partitioning is almost *no increase* in the overall index-size (as we show later, only overhead is to maintain a small set of location pointers in each partition). We develop a single-pass greedy algorithm that optimally shards the index list, minimizing the number of index entries read during query processing.

In the resulting sharded index, query processing proceeds by first *opening* all index list shards for each query term, and *seeking* to the right position inside each shard, and, next, sequentially scanning the shard from that position

–resulting in one random access followed by a sequential scan. As a random access is at least as expensive as a sequential read (as in disk- and network-based index storage), breaking the index list into too many shards actually degrades performance, even if the abstract processing cost says otherwise. Thus, the practical efficiency of the index organization is achieved only if it is sensitive to the cost ratio of random accesses to sequential accesses. We formulate an optimization framework for tuning the optimal sharding that takes into account the I/O cost ratio of the storage infrastructure and propose a heuristic to combine shards to gain practical runtime efficiency.

In summary, key contributions made in this article are:

1. A novel temporally sharded index organization for a time-enriched inverted index that overcomes the issue of index-size blowup.

2. An optimal greedy algorithm to shard the index list, so that no time-travel query reads more than the required index entries, thus achieving ideal query processing performance.

3. A framework that achieves practical runtime efficiency by tuning the number of shards that each index list is split into, taking into account the I/O cost ratio of the storage infrastructure. Note that the cost of this is *independent* of the dynamics in the query workload, as it depends only on the storage system parameters.

We performed extensive empirical evaluation on large-scale versioned document collections using real-world keyword queries with temporal constraints at varying granularities. Results of these experiments indicate that our sharded temporal index has a negligible ($\approx 1\%$) increase in index size, but outperforms an optimized implementation of the previous best approach [9].

1.2 Organization

The remainder of this report is organized as follows: in Section 2, we briefly present the data and time-travel query model that we use in this report, as well as the time-travel inverted index [9] that we compare our work against. Next, in Section 3, we describe in detail the idea of sharding the inverted index and query processing over the resulting index organization. In Section 4 and in Section 5 we present the details of our idealized sharding and a sharding strategy sensitive to I/O cost ratio respectively. Details of our experimental setup are in Section 7 and the key results are in Section 8. Finally, we discuss previous related work in Section 9, before concluding in Section 10.

2 Preliminaries

2.1 Data and Query Model

We operate on a collection $\mathcal{D}$ of versioned documents. Each document d has a unique identifier id_d and consists of a sequence of its versions, $d = \langle d^1, d^2, \ldots \rangle$, where every version consists of terms drawn from a vocabulary $\mathcal{V}$, i.e., $d^i \subseteq \mathcal{V}$. Furthermore, each document version has an associated valid-time interval $I(d^i) = [\ begin(d^i),\ end(d^i)\ )$ that conveys when the document version d^i existed in the real world. We make the natural assumption that these valid-time intervals for any two versions of the same document are disjoint, i.e.,

$$\forall_{d^i, d^j, i \neq j} I(d^i) \cap I(d^j) = \emptyset\,.$$

A document version that still exists when it is added to the archive is called an *active* version and has $end(d^i) = \infty$.

A time-travel query over $\mathcal{D}$ consists of a set of terms $Q = \{q_1, \ldots, q_m\}$ and a time interval $[b_Q,\, e_Q]$. When evaluated, this query retrieves the set of document versions that satisfy the keyword query Q and existed at any time during the time interval $[b_Q,\, e_Q]$. When using the conjunctive boolean model for keyword retrieval, we can write the result as:

$$\begin{aligned} &R(\, Q@[b_Q,\, e_Q]\,) = \\ &\quad \left\{\, d^k \in \mathcal{D} \mid \forall q \in Q \,:\, q \in d^k \,\wedge\, I(d^k) \cap [b_Q,\, e_Q] \neq \emptyset \,\right\}\,. \end{aligned}$$

Queries for which $b_Q = e_Q$ holds, so that the query time interval collapses into a single time point, will be referred to as *time-point queries.*

2.2 Index Organization

The index used for handling time-travel queries is based on the established *inverted index* [21], where each term in the vocabulary is associated with a list of entries called an *index list.* Each entry in the index list, $\langle id_d, I(d^i), s\rangle$, consists of a document identifier id_d, the valid-time interval, $I(d^i) = [begin(d^i), end(d^i)]$, and a payload s. Based on the retrieval model employed, the payload can be empty (for boolean retrieval), a scalar value (tf in the document version) or even richer positional information in each version –the index organization supports all these settings. When no confusion arises, we simply use $begin(p)$ and $end(p)$ of an index entry p to refer to the valid-time interval boundaries of the corresponding document version.

This index further supports partitioning of each index list into smaller index lists. This partitioning can be done either along (i) the time-axis, i.e., have all index entries that overlap a contiguous segment of time together, or (ii) the document-identifier dimension. In this work, we call the former partitioning *slicing* and the latter *sharding.*

Index Slicing of [9]

In the time-travel indexing framework proposed by Berberich et al. [9], the time-enriched inverted lists are sliced into several smaller sublists each spanning a specific subinterval of the time axis. As we already mentioned, every entry whose valid-time interval spans the boundaries of more than one slice is replicated in all the slices which overlap (cf. Figure 1.1). In their paper, the authors also show that the optimal slicing (defined as the setting where any time-travel query reads only those entries that qualify by their valid-time interval) is impractical to achieve due to the resulting blowup (more than 100×) in index size. As an alternative, they formulate an optimization problem which takes as a user-given input, an upper bound κ on the tolerable index-size blowup, and outputs the best slicing satisfying this constraint. The quality of the slicing is determined by the expected number of index entries that are read for any time-travel query.

Another interesting aspect of the above framework is *temporal coalescing*, a way of compressing index lists either in a lossy or non-lossy way. Temporal coalescing merges temporally contiguous entries of the same document into a single index entry, as long as the relative change in the payload s is within a specified threshold ϵ. Although this is somewhat orthogonal to the issue of partitioning, the distribution of valid-time intervals of entries in the index list could vary significantly after performing coalescing.

3 Temporal Index Sharding

The index sharding proposed in this report, on the other hand, distributes the entries in an index list over disjoint subsets called *shards*, avoiding replication completely (cf. Figure 1.1). Entries in a shard (and in the original list) are ordered according to their begin times. For each shard we maintain an *impact list* as an additional access structure for efficiently determining the entries whose time segments overlap with a query time interval. This impact list maintains, for every possible begin time of a query time interval, the position in the shard of the earliest entry whose time segment contains the begin time. In other words, the impact list maintains pairs of all possible query begin times (key) and offsets (values) from the shard beginning. Given a query time interval, it is sufficient to start scanning the shard at the position (offset) stored in the impact list, instead of scanning the shard from the beginning. The overall size of each impact list can be reduced by storing only the distinct offset values rather than offsets for all possible query begin-times. A straightforward binary search over the query begin-times gives the correct offset location. For practical granularities of query begin-times such as days, impact lists for the complete index (i.e., for all shards of all terms) can be easily kept in memory.

Query processing follows the established term-at-a-time processing model where index lists are read one after the other and scores of a document version from different lists are merged in memory. For our sharded index, each query term is processed in a sequence of *open-skip-scan* operations, one for each shard:

1. **Open** – Open a shard for a query term.
2. **Skip** – Given the query begin time, lookup offset position from the impact list of the shard and perform a seek to that position. In practice we can combine the open and skip steps into an *open-skip* operation to save an extra I/O operation.

3. **Scan** – Perform sequential reads from this position and terminate when it is certain that rest of the entries do not overlap with the query time interval. As we read the entries sequentially in begin-time order, we can safely terminate when the begin-time of the next entry exceeds the query end-time.

Observe that all the shards for a given term are accessed independently of the temporal constraint in the time-travel query. For a sharded index, merging results from every shard of a index list is not expensive since the shards are disjoint; this is in contrast to the situation for temporally sliced index lists, where entries replicated in different slices must be detected and duplicates removed.

4 Idealized Index Sharding

As the entries in each shard are ordered according to their begin times, with the help of the impact list we can easily avoid wasteful reading of entries whose valid-time interval, $I(d^i)$, is before the query begin time. However, this does not guarantee the elimination of all wasteful reads. Consider a situation when two index entries p and q contained in a shard such that p completely subsumes q, i.e.,

$$begin(p) \leq begin(q) \ \wedge \ end(p) > end(q).$$

Now, the queries with $end(q) \leq b_Q \leq end(p)$ will wastefully read q. This can arbitrarily degrade performance if there are many such entries, for instance, when there is an entry which spans long intervals.

We can avoid any wasteful reads of entries such that no pair of entries from a shard form such a subsumption pattern. In other words, we require that entries in a shard satisfy *staircase property*, defined as follows:

Definition 1 (Staircase Property)**.** *Given a shard g_i, if we have*

$$\forall p, q \in g_i, begin(p) \leq begin(q) \Rightarrow end(p) \leq end(q) \ ,$$

then we say the shard g_i has the staircase property. □

Clearly, it may be possible to shard a given index list in many different ways so that the staircase property (cf. Figure 4.1) is satisfied. Since query

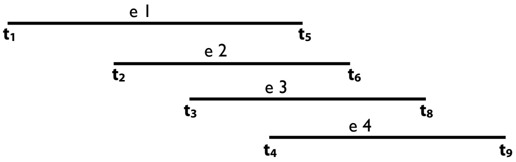


Figure 4.1: Staircase property

processing proceeds by open-skip operations for all shards of a term, it is desirable to minimize the number of idealized shards. This can be cast into an optimization problem where, given a set of time intervals (corresponding to the valid-time intervals of index entries), minimize the number of shards. Formally this is defined as follows:

Definition 2 (Idealized Sharding)**.** *Given a set of entries L_v for a term v, partition L_v into a* minimal set *of shards $G = \{g_1, \ldots, g_m\}$, $g_i \subseteq L_v$ where:*

$$g_i \cap g_j = \emptyset \qquad \forall g_i, g_j \in G,\ i \neq j \tag{4.1}$$

$$\bigcup_i g_i = L_v \tag{4.2}$$

$$\forall g_i \in G \ :\ g_i \text{ satisfies the staircase property.} \tag{4.3}$$

□

4.1 Optimal Algorithm

Algorithm 1 Idealized Sharding Algorithm

```
Input: $L_v$ sorted in increasing order of begin times
$G_{opt} = \emptyset$    // Idealized sharding

for $i = 1 \,..\, |L_v|$ do
  // Iterate over all entries in the index list for $v$
  if $\neg\exists g \in G_{opt} : g.end \leq end(L_v[i])$ then
    create new shard $g_{new}$
    $g_{new}.end = 0$
    $G_{opt} = G_{opt} \cup \{g_{new}\}$
  end if
  $g_t = \underset{g \in G_{opt}}{argmin}\ (end(L_v[i]) - g.end)$
  $g_t.end = end(L_v[i])$ // Update the end-time of the shard
  $g_t = g_t \cup \{L_v[i]\}$ // Include the current entry into the shard

end for

Output: $G_{opt}$ is the idealized sharding.
```

Let each shard g_i associated with an index list of begin-time sorted entries have an end-time $g_i.end$, defined as the latest end-time of all the entries in g_i.

We employ the greedy algorithm given in Algorithm 1, in which we process all entries of a list L_v in increasing begin-time order. At each iteration, we include an entry e into an existing shard if its inclusion does not violate the *staircase property* of the shard. If there are multiple shards to which e can be assigned, we assign it to the shard with the minimum gap, $end(e) - g_i.end$. If there are currently no shards to which e can be assigned, we start a new shard with e in it. It turns out that the idealized sharding problem maps directly to the problem of decomposing a set of points in a plane into minimum number of ascending chains and essentially uses the same algorithm as in [20].

4.2 Proof of Optimality

We develop the proof for the optimality of Algorithm 1 by first proving a set of three lemmas about key properties of the generated shards.

The first lemma states that the algorithm produces only shards that have the staircase property.

Lemma 1 (Staircase Property)**.** *When Algorithm 1 terminates, every shard created by the algorithm has the staircase property.*

Proof. We show this by contradiction. Assume that there is one shard that does not have the staircase property. This means that there are two entries p and q in this shard such that $begin(p) \leq begin(q)$, but $end(p) > end(q)$. Since $begin(p) \leq begin(q)$, p was added to the shard before q. But when p was added, end time of the shard was set to $end(p)$, so q could not have been be added to the same shard, which contradicts our assumption that p and q are in the same shard. □

For the next lemma, let the shards created by Algorithm 1 for a list L_v be numbered by their order of creation.

Lemma 2 (Temporal Subsumption of Entries)**.** *For every entry in a shard* g_i $(i > 1)$ *there exists an entry in* g_{i-1} *which completely subsumes it.*

Proof. Any entry e when added to g_i equals the end of a shard when it is added (line 7). At this stage it is subsumed by the entry e' which forms the end of g_{i-1}. This is because e' has an earlier begin time than e because of its early addition. □

We introduce the notion of a *stalactite set* of time intervals. A stalactite set S consists of time intervals such that,

$$\forall p, q \in S, begin(p) \leq begin(q) \Rightarrow end(p) > end(q).$$

There may be many such stalactite sets that can be formed using entries from a given index list, L_v. Let us denote the stalactite set of maximum cardinality as $S_{max} : L_v$.

Lemma 3 (Stalactite property)**.** *The number of shards created by Algorithm 1 for a list L_v is equal to $|S_{max} : L_v|$.*

Proof. We prove the lemma using contradiction. Assume that the new entry to be placed e_{new} is not a part of $|S_{max} : L_v|$. We also assume that its addition creates a new shard $g_{|S_{max}|+1}$ which is more than the claimed $|S_{max}|$ shards. Since the newly placed entries arrive in begin time sorted order, $begin(e_{new})$ is more than any of the previously placed entries. This means that $end(e_{new}) < end(g_{|S_{max}|})$ for it to start a new shard $g_{|S_{max}|+1}$. Now Lemma 2 says that there exists an entry in $g_{|S_{max}|}$ which subsumes e, making e a part of a larger stalactite set of cardinality $|S_{max}| + 1$ which is contrary to initial assumption that S_{max} is the maximal stalactite set. □

Now, we can state and prove the optimality of the algorithm.

Theorem 1. *Algorithm 1 creates an optimal sharding.*

Proof. It follows from Lemma 1 that every element in $S_{max} : L_v$ has to be a part of a new shards which lower bounds the number of shards to $|S_{max} : L_v|$. Lemma 3, on the other hand, upper bounds the number of shards to $|S_{max} : L_v|$ which proves the optimality of Algorithm 1. □

Further, we show that the algorithm can be implemented efficiently, by making use of the following property of the sharding at any stage during the algorithm:

Lemma 4 (Descending End Times)**.** *If Algorithm 1 created a shard g_i before g_{i+1}, then $g_i.end > g_{i+1}.end$.*

Proof. We prove this property by induction over increasing number of entries added.

$i = 1$: For the first entry e_1 the property holds since there are no earlier shards.

$i \rightarrow i + 1$: Let there be n existing shards $G = \{g_1 \cdots g_n\}$. Depending on the end time of the $i + 1$th entry ,i.e., $end(e_{i+1})$ we consider two cases:

If $end(e_{i+1})$ is less than all the existing shard ends ($end(e_{i+1}) < g_k.end$, $\forall g_k \in G$), then e_{i+1} forms a new shard and the end of the shard is less than all the existing ends. This proves the claim.

Due to the induction hypothesis, the end times of shards are sorted in the descending order i.e., $g_1.end > g_2.end > \cdots > g_n.end$. If $end(e_{i+1})$ is greater than any of the shards, then by definition (line 6), it will have to be assigned to the shard which minimizes the difference of their end times. It is easy to see that this shard is the earliest shard which can accommodate e_{i+1} since any other group will either violate the staircase property or have a greater difference. This proves the claim. □

Owing to such an ordering of shard ends, entries can be placed efficiently via a binary search over the shard ends.

5 Cost-Aware Merging of Shards

Depending on the distribution on valid-time intervals, the idealized sharding introduced in the previous section might generate a large number of shards requiring one *open-seek* operation, involving a random access, for each shard. If the cost of such a random access is high and if the distribution of time intervals gives rise to many idealized shards, query processing performance can degrade. In such cases, it might actually be beneficial to relax the idealized sharding, and reduce the number of shards at the cost of allowing some wasted reads.

In this section, we present an I/O cost-aware technique to selectively merge idealized shards allowing for a controlled amount of wasted reads while reducing the number of random accesses. We introduce a cost model which limits sequential wasted reads due to merging of a set of idealized shards by taking into consideration costs of random accesses and sequential reads of the underlying index infrastructure.

5.1 Cost Model

Let the cost of a random access be C_r, and that of a sequential read be C_s. We allow for a penalty function for a set of idealized shards $\{g_i\}$ as $P(\{g_i\})$ and require it to be bounded by C_r/C_s if the shards should be considered for merging. We refer to this as the *threshold criterion*, i.e,

$$P(\{g_i\}) \leq \frac{C_r}{C_s}$$

An example of such a penalty function is *mean wasted reads*, which is defined as the number of wasted reads incurred during query processing, averaged over all possible query begin time points. Under this penalty function, g_i and g_j can be merged when the mean wasted sequential reads in the *merged shard* is less than the overhead incurred in an open-seek operation. In other words, an open-seek operation to one merged shard accompanied by a few sequential wasted reads is cheaper than two open-seek operations to idealized shards without any wasted reads. As an example, if $C_r/C_s = 100$, then the wasted reading of less than 100 entries, that do not qualify by the temporal constraint, on an average would be more beneficial than performing a random access to an additional shard.

It turns out that due to the specific organization of time intervals in the idealized shards, the total penalty, $P(\{g_i\})$, of merging a set of idealized shards, $\{g_i\}$, can be easily computed from the pairwise penalties of participating idealized shards $P(g_i, g_j)$'s. Also note that $P(g_i, g_i) = 0$. Now, for merging a set of idealized shards $m = \{g_i, g_j, \cdots\}$ the overall penalty $P(m)$ is given by

$$P(m) = \sum_{g_k \in m} P(first(m), g_k)$$

We retain the order in which idealized shards were created, i.e., shards created early have a lower index. Thus $first(m)$ denotes the shard which was created first in m. This can be shown as follows:

For a pair of idealized shards (g_i, g_j) where $i < j$, and for a given query begin-time t, let the set of wasted entries read be $w_{i,j}(t)$. It is easy to see from Lemma 2 that $w_{j,k}(t) \subset w_{i,k}(t)$. Hence, for a set of idealized shards $g_1, g_2, \cdots, g_n$ the $w_{1,2,\cdots,n}(t) = w_{1,n}(t)$, which means that aggregation of multiple idealized shards depends on the idealized shard which is created first. As an example, the penalty incurred due to merging idealized shards {7,10,3,12} would be $P(3, 7) + P(3, 10) + P(3, 12)$.

Computing pairwise penalties: Consider a pair of entries p and q, which belong to different idealized shards, where p is subsumed by q. A read is said to be wasted for each query begin time point lying in the interval $[end(p), end(q))$. The total cost of merging two shards g_i and g_j is the sum of all wasted reads caused from all such pairs of subsumed entries. The average penalty $P(g_i, g_j)$ hence is the total cost of merging the pair of groups divided by all possible query begin time points. Computing wasted reads at each time point can be efficiently implemented by interleaving computation of pairwise wasted reads within Algorithm 1.

5.2 Problem Statement

Given a set of idealized shards G_{opt}, pairwise penalty values $P(g_i, g_j)$ and a threshold C_r/C_s find a set of disjoint merged shards M of *minimum cardinality* such that each of the merged shards respects the *threshold criterion.*

Disjoint merged shards require that no pair of merged shards have an idealized shard in common. We formally define the optimization problem as

$$\min |M| \quad \text{s.t.}$$
$$P(m) \leq C_r/C_s\,,\ m \in M\,,$$

where m is a merged shard. Observe that the disjoint merging is encoded in the objective function $min|M|$. We refer to the process as **relaxed sharding**.

5.3 Algorithm For Merging Optimal Groups

We present a heuristic algorithm which is shown to perform well in practice in our experimental evaluation. As inputs we expect the set of the idealized shards and the ratio C_r/C_s. We retain the order in which idealized shards were created, i.e., earlier created shards have a lower index.

The pseudo code for merging the idealized shards is presented in Algorithm 2. Every iteration employs a two stage greedy process. The first stage is an *ascending choice phase* in which it chooses all the unmerged/available idealized shards in ascending order of their index until the threshold constraint is violated (lines 11 to 20).

The second stage is a greedy phase (lines 23 to 27) where the remaining capacity is greedily chosen with smallest unmerged shard first (as in the standard greedy approach to the knapsack problem).

Algorithm 2 Cost Aware Group Merging

```
*Input:* $G_{opt}$ and $P(g_i, g_j)$
$M = \emptyset$    // Merged shards

**for** $i = 1 \,..\, |G_{opt}|$ **do**
   Let $g_i \in G_{opt} \wedge g_i \notin M$ be next shard in order
   **create** new shard $r_i$
   $r_i = r_i \cup \{g_i\}$
   $capacity \ = \frac{C_r}{C_s}$

   //ascending choice phase
   **for** $j = i + 1 \,..\, |G_{opt}|$ **do**
      **if** $(P(g_i, g_j) \leq capacity) \ \wedge \ (g_j \notin M)$ **then**
         $capacity \ = capacity - \ P(g_i, g_j)$
         $r_i = r_i \cup \{g_j\}$
      **else**
         **if** $(g_j \notin M) \ \wedge \ (g_j \notin r_i)$ **then**
            **break**
         **end if**
      **end if**
   **end for**

   // smallest size first
   **while** $capacity > 0$ **do**
      $g_{min} = \underset{(g \in G_{opt}) \cap (g \notin M)}{argmin} \{P(g_i, g)\}$
      $capacity \ = capacity - \ P(g_i, g_{min})$
      $r_i = r_i \cup \{g_{min}\}$
   **end while**
   $M = M \cup r_i$
**end for**

*Output:* $M$ is the set of merged shards.
```

6 Dealing with Index Updates

After the initial index building, the archive may grow over time, so that the index needs to be updated. This growth usually comes from a crawler fetching new versions of documents already in the archive, or documents that were unknown before. We therefore make the realistic assumption that the vast majority of updates will either replace an existing active version (setting its end time to the crawl time) or add a new active version (whose begin time is the crawl time, and whose end time is ∞), and assume that the begin time of inserted versions is not smaller than begin times of any version already in the index. Insertions of older versions will be a rare exception, for example caused by merging our archive with another archive; in these cases, we resort to recomputing the index.

We will now explain how the index is maintained under such an update load, mainly by explaining to which (idealized or relaxed) shard a new version is added. Note that the actual techniques for adding new index entries to shards are the same used for updating standard inverted lists, for example in-place updates or logarithmic merging; see [10] for a thorough discussion of such techniques.

We first show how to update idealized shards. Let us assume for the moment that there are no active versions in the index. We then sort the versions to insert by their begin time and append them to the index by applying Algorithm 1 with them as input, updating and extending the current set of shards. Note that while running the algorithm, we update the (in-memory) impact lists, which can be flushed to disk from time to time by a background process. The resulting set of idealized shards is optimal as the output of the algorithm is the same as if we ran it for all versions (those already in the archive and those added to it) with an initially empty set of shards. Notice that the algorithm consumes versions in ascending order of their begin timestamps, and all begin timestamps in the archive are strictly smaller than the begin timestamps of newly added versions.

Active versions make the process more complicated when they are replaced by a newer version; otherwise, they are not touched by the update. Consider an active version d^i of a document d in shard g that is overwritten by a newer version d^{i+1} at time t, setting $end(d^i) = t$. If we just updated $end(d^i)$ and there was at least one other active version in the same shard, we would incur a wasted read of d^i for any query time greater than t. To avoid this, we split shard g into two pieces: The first new shard gets all versions v from g with $begin(v) \leq begin(d^i)$, the second new shard gets the remaining versions. Since versions in shards are sorted by ascending begin timestamp, this can usually be done by simply setting shard boundaries accordingly, without actually moving index entries (which would only be necessary when there are multiple versions with the same begin timestamp, which is very unlikely if the begin time is set to the crawl time). We have to update the impact lists of both new shards, which can be efficiently done on this in-memory data structure. Since versions in the first new shard are unchanged, no wasted reads occur by construction. In the second new shard, wasted reads could only occur when querying for timestamps after $end(d^i)$. Since all begin times of versions other than d^i are larger than $begin(d^i)$, the impact list will point to an index entry after d^i for these queries, avoiding wasted reads. After having updated all end times of active versions that are overwritten, we insert the versions replacing them using the algorithm for updating idealized shards without active versions.

Note that this maintenance procedure can be applied either for batched updates or after each (new or updated) version is read by the crawler.

Updating relaxed shards can be done as follows: When we initially construct the relaxed shards, we store from which idealized shards they were built. When we now insert a (new or updated) version into an idealized shard, we insert it at the same time into its corresponding relaxed shard. This may, of course, result in a set of relaxed shards where for some relaxed shards our cost bound is violated (because the updates incurred too many wasted reads). We therefore compute regularly, for each relaxed shard m, its current aggregated penalty $P(m)$, and recompute all relaxed shards when the maximal or average aggregated penalty gets too large, for example when it exceeds $2 \cdot C_r/C_s$.

7 Evaluation Framework

In this section, we present our experimental setup and the datasets that we use to evaluate our approach in terms of query-processing performance and space consumption.

7.1 Setup

We implemented the sharded index and all our algorithms using Java 1.6. Additionally, we obtained the latest implementation of the sliced index from its authors. All experiments were conducted on Dell PowerEdge M610 servers with 2 Intel Xeon E5530 CPUs, 48 GB of main memory, a large iSCSI-attached disk array, and Debian GNU/Linux (SMP Kernel 2.6.29.3.1) as operating system. Experiments were conducted using the Java Hotspot 64-Bit Server VM (build 11.2-b01).

7.2 Datasets

For our experiments we use the following two real-world datasets.

WIKI The *English Wikipedia revision history* [1], whose uncompressed raw data amounts to 0.7 TBytes, contains the full editing history of the English Wikipedia from January 2001 to December 2005. We indexed all versions of encyclopedia articles excluding versions that were marked as the result of a minor edit (e.g., the correction of spelling errors etc.). This yielded a total of 1,517,524 documents with 15,079,829 versions having a mean (μ) of 9.94 versions per document at standard deviation (σ) of 46.08.

UKGOV This is a subset of the European Archive [2], containing weekly crawls of eleven governmental websites from the U.K. We filtered out documents not belonging to MIME-types `text/plain` and `text/html` to obtain a dataset that totals 0.4 TBytes. This dataset includes 685,678 documents with 17,297,548 versions ($\mu = 25.23$ and $\sigma = 28.38$).

Note that the two datasets represent realistic classes of time-evolving document collections. WIKI is an explicitly versioned document collection, for which all its versions are known. UKGOV is an archive of the evolving Web, for which, due to crawling, we have only incomplete knowledge about its versions. For ease of experimentation, we rounded timestamps in both datasets to day granularity.

7.3 Index Management

Both the sliced and sharded indexes are stored on disk using flat files containing both the lexicon as well as the sliced or sharded index lists. At run time, the lexicon and impact lists are read completely into main memory, and for a given query the appropriate slices or shards are retrieved from the index flat file on disk. For compression we employ 7-bit encoding [16]. Note that such variable-byte encoding is complementary to other compression methods like *temporal coalescing* [9]. We compare the following types of indexes in our experimentation:

Sharded Indexes We consider idealized sharding (IS) and three cost-aware *relaxed sharding* variants with relaxation parameters 10.0, 100.0 and 1000.0 referred to as RS-10, RS-100 and RS-1000. Recall that the relaxation parameter reflects the I/O cost ratio introduced in Section 5. The cost-aware metric is based on the penalty function, *mean wasted reads*, for merging the idealized shards(IS).

Sliced Indexes We consider instances of the sliced index that are partitioned following the *space-bound* approach [9]. The parameter κ denotes the space restriction that models the maximum blowup in index size relative to a non-partitioned index. For our experiments we consider four variants of the space-bound approaches i.e., parameter values for $\kappa = \{1.5, 2.0, 2.5, 3.0\}$. These variants are denoted subsequently in the text as SB-1.5, SB-2.0, SB-2.5 and SB-3.0.

Naïve Unpartitioned Index As a second competitor, we build an unpartitioned index with provision for impact lists over ordered begin times referred to as RS-inf. This serves as a proof that our techniques are effective not only because of the impact list construction and a global begin time order.

The average length of the index lists in our sharded index were 496,259 for WIKI and 945,044 for UKGOV. Since sharding causes each index list to be split into multiple shards we briefly look at the size distribution of the above mentioned sharded indexes. For WIKI, idealized sharding (IS) resulted in 79.75 shards per index list (shards/list). This reduces to 32.5 shards/list for RS-10 and further to 11.78 shards/list (RS-100) and 4.71 shards/list (RS-1000). The shard sizes in IS varies from 2% - 8% of the entire index list size, while for RS-10 its 7%-20% and 35%-50% for RS-1000. For UKGOV, IS resulted in 17.86 shards/list. The number of shards/list reduce to 14.36 (RS-10), 9.9 (RS-100) and 4.76 (RS-1000). The shard sizes as a fraction of the entire list vary from 1%-2% (IS) to 7%-8% (RS-10) and finally 4%-44% (RS-1000).

We also evaluate the effect of temporal coalescing [9] on index size and query processing. To this effect we build sharded and sliced indexes with application of temporal coalescing using a parameter $\epsilon = 0.01$. Other than that, we use the same choice of parameters as in the experiments without temporal coalescing.

7.4 Query Workloads and Execution

We compiled two dataset-specific query workloads by extracting frequent queries from the AOL query logs, which were temporarily made available during 2006. For the WIKI dataset we extracted 300 most frequent queries which had a result click on the domain `en.wikipedia.org` and similarly for UKGOV we compiled 50 queries which had result click on `.gov.uk` domains. Both the sliced and sharded index structures are built for terms specific to the query workload. Using these keyword queries, we generated a time-travel query workload with 5 instances each for the following 4 different temporal predicate granularities: day, month, year and queries spanning the full lifetime of the respective document collection.

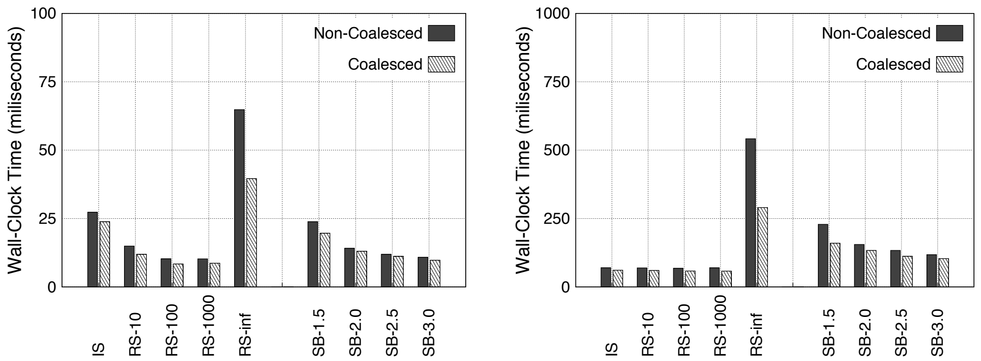
For query processing we employed conjunctive query semantics i.e., query results contain documents that include all the query terms. We use wall-clock times (in milliseconds) to measure the query processing performance on warm caches using only a single core. Specifically, each query was executed five times in succession and the average of the last four runs was taken for a more stable and accurate runtime measurement.

8 Experimental Results

8.1 Sharding vs Slicing

In the first set of experiments, we compare the performance of sharding and slicing on different query granularities. In the plots presented, runtimes of different variants of both sharded and sliced partitions are shown in milliseconds. Each plot corresponds to a given query granularity – day, month, year or the full life time of the respective collection. C_r/C_s values of disks usually vary in the order of 100 and 1000. Since runtimes for both RS-100 and RS-1000 show low variance throughout all experiments we chose RS-1000 as a reasonable representative for sharding for this comparison.

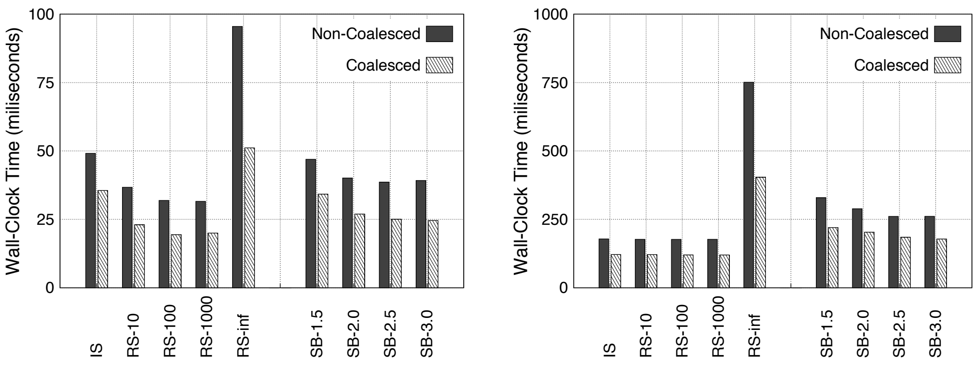
SB-3.0 is optimized for short time interval queries, because of a higher degree of partitioning, and as expected it performs better than any of its other sliced counterparts ($\kappa = \{1.5, 2.0, 2.5\}$) for day and month queries. In case of WIKI, we see a low difference in query processing times between SB-3.0 (10.63 ms) and RS-1000 (10.26 ms) in case of day queries (Figures 8.1(a)). The difference is notable for month queries with RS-1000 exhibiting a 19.5%



(a) Wikipedia

(b) UKGOV

Figure 8.1: Wall-clock times for day queries



(a) Wikipedia

(b) UKGOV

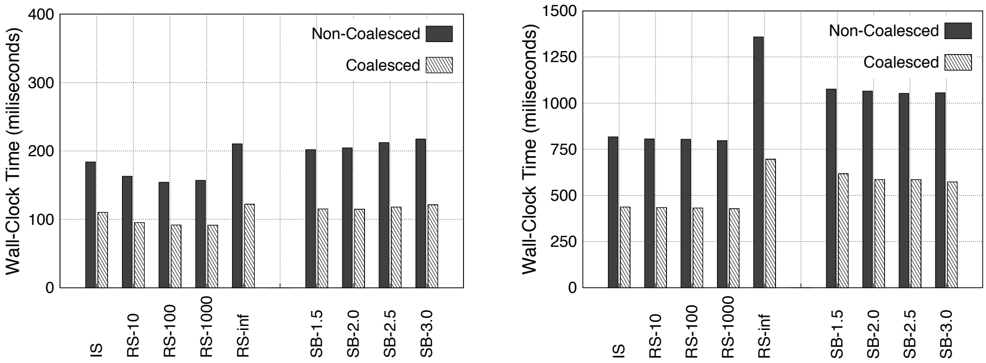
Figure 8.2: Wall-clock times for month queries

improvement over SB-3.0 (Figure 8.2(a)). In UKGOV, RS-1000 takes 69.88 ms to process day queries which is almost 40% improvement over SB-3.0 which takes 117.9 ms (Figure 8.1(b)).

While SB-3.0 is efficient for short time interval queries, they understandably perform worse for longer time intervals because of having to read a larger number of replicated entries inflicted by a higher degree of partitioning. This is shown in Figures 8.3(a), 8.4(a), 8.3(b) and 8.4(b) where smaller κ valued sliced indexes ($\kappa = \{1.5, 2.0, 2.5\}$) perform better than SB-3.0. In case of WIKI, comparing SB-1.5, which has the best runtimes for year queries and full life time queries, with RS-1000 shows that the latter consistently outperforms in both cases. Query processing times drop by 22.2% (Figure 8.3(a)) for year queries and by 19% for full lifetime queries (Figure 8.4(a)). In UKGOV, there is an improvement of almost 29.9% or 279.26 ms(RS-1000 vs SB-1.5) 8.3(b) for year queries. The full time queries are faster by 931 ms or 21%(RS-1000 vs SB-1.5) as shown in Figure 8.4(a) and 8.4(b) which in absolute terms seems to be a considerable difference.

8.2 Effect of Cost Aware Merging of Shards

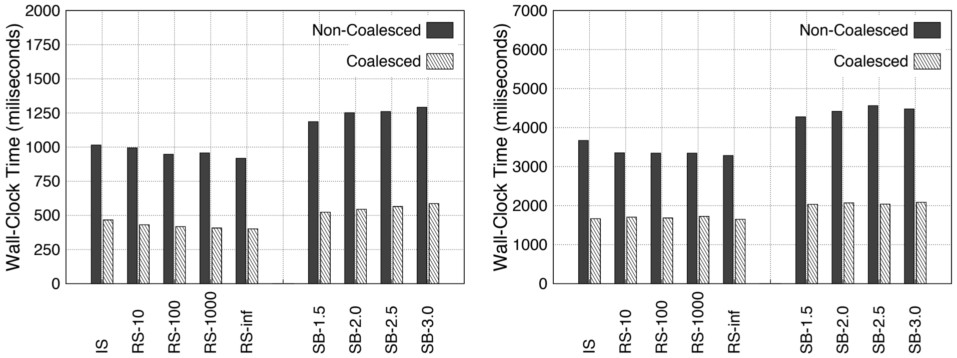
The next set of experiments present the effectiveness of cost aware merging of shards. Idealized sharding or IS might be characterized by a high number of shards for certain distributions of document life times. Although query processing on IS results in reading only the relevant entries intersecting with the query time interval, they suffers from inefficiencies due to a large number of random accesses. Especially for disks with $C_r >> C_s$, the open-seek operation on idealized shards might result in considerable overheads.



(a) Wikipedia

(b) UKGOV

Figure 8.3: Wall-clock times for year queries



(a) Wikipedia

(b) UKGOV

Figure 8.4: Wall-clock times for full-lifetime queries

In WIKI, we see a consistent improvement from IS to RS-1000. This is because idealized sharding admits a fairly large number of shards in this case and thus the I/O costs are dominated by initial random accesses to access these idealized shards. Improvements result from the reduction in the number of shards due to careful merging of idealized shards as presented before in Section 5. Although these reductions might not be significant for queries with larger time intervals, but they reduce query processing time by a sizable fraction for smaller interval queries – day queries improve by 62%(Figure 8.1(a)) and month queries by 35%(Figure 8.2(a)). Unlike WIKI, UKGOV does not show any considerable difference in performance which is due to the already low number of initial idealized shards. This indicates that cost-aware merging can be applied as a self-organizing approach depending on the distribution of initial shards.

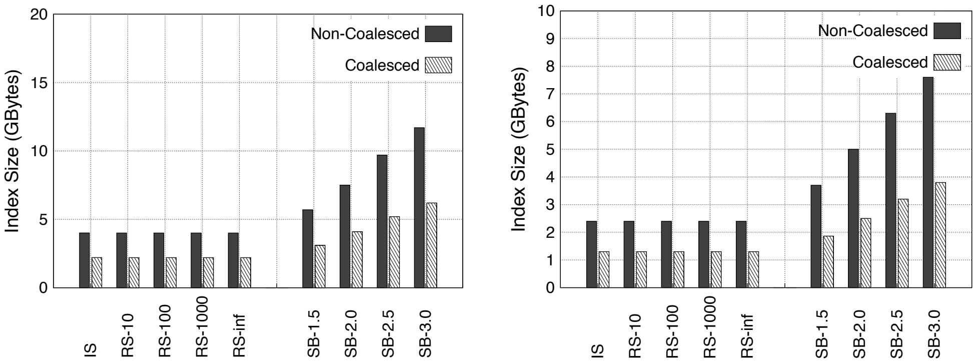
RS-1000 however outperforms RS-inf by a fairly large margin in all query granularities excepting full-lifetime ones showing the effectiveness of careful sharding of entries. The behavior for full lifetime queries is to be expected because all entries in RS-inf become relevant for such kind of queries and have to be subsequently read.

8.3 Index Sizes

As expected we observe that the size of sharded indexes remain the same as the unsharded index. This is due to the fact that sharding divides the index entries of the unpartitioned lists in a disjoint manner. On the contrary the index entries in a sliced index are subject to replication across slices. The size of the sliced index structures show a direct correlation with the input parameter κ as shown in Figure 8.5(a). As discussed before, κ regulates the upper bound to the index blowup. The higher the κ the more efficient is the performance of short time interval queries at the expense of a larger index size. However, longer time interval queries are impacted by the higher κ valued indexes due to wasted amount of reads of replicated index entries. Thus the sliced index has to be carefully tuned depending on the time interval of the query workload trading off index size and query efficiency. This is not the case with the sharded index where the tradeoff is between number of random accesses and sequential reads, which are local tunable parameters depending on physical characteristics of disks (where the index is stored) irrespective of the query workload. From our experiments we observe that the time taken to build a sharded index is roughly twice the time taken for the standard unpartitioned inverted index. Since the sharded index building process can be easily parallelized, one can efficiently build sharded indexes using a distributed processing platform (e.g., Hadoop).

8.4 Effect of Coalescing

Our experimental results with temporal coalescing of index entries lead to similar results as our experiments on the original, uncoalesced indexes. Independent of the partitioning/sharding used, index size is much smaller than with uncoalesced indexes—up to an order of magnitude for UKGOV and up to a factor of 2 for Wikipedia—and indexes created with sharding are always smaller than those with slicing (Figure 8.5). The runtime of all methods with temporal coalescing is depicted in Figures 8.1(a) through 8.4(b). It is evident that query performance improves with temporal coalescing, with longer time interval queries gaining more than those with smaller time intervals as more



(a) Wikipedia

(b) UKGOV

Figure 8.5: Index sizes

entries need to be read. Sharding is still more efficient than slicing, especially with cost-aware merging, and achieves this advantage with indexes of a smaller size.

9 Related Work

Temporal information associated with documents has recently seen increasing attention in information retrieval. One of the earliest known efforts in this direction is by Anick and Flynn [7] who developed a framework for versioning the complete index for historical queries. Recently, Alonso et al. [5] give an overview of relevant research directions. The work on time-travel text search by Berberich et al. [9] is closest to our work in this report. It introduced the notion of time-travel queries, limiting its discussion to time-point queries. To support this functionality efficiently, index lists from an inverted index are temporally partitioned, in our terminology into slices, providing guarantees on either the space used by the index or query performance. Index entries whose valid-time interval overlaps with multiple of the determined temporal slices are judiciously replicated and put into multiple index lists, thus increasing the overall size of the index. The work by Herscovici et al. [15] focuses on exploiting the redundancy commonly seen in versioned documents to compress the inverted index. Similarly, He et al. [13, 14] consider the problem of efficiently storing inverted indexes on disk using compression; these are orthogonal to our work and could be combined with our sharding techniques.

Research in temporal databases has taken a broader perspective beyond text documents and targeted general class of time-annotated data. Index structures tailored to such data like the Multi-Version B-Tree [8] or LHAM [17] are related to our work, since they also, implicitly or explicitly, rely on a temporal partitioning and replication of data. It is therefore conceivable to apply our proposed techniques in conjunction with one of these index structures.

10 Conclusion

This work presents a novel method of index organization based on sharding for solving time-travel queries. Previous approaches traded off space and efficiency of query processing resulting in an index size blowup. Also, there was no single partitioning scheme for a given query time interval granularity. Unlike those, we look at minimizing the I/O during query processing without incurring any overhead in space and are independent of the query workload. We also propose a cost model, which takes into account different I/O costs of the local storage media where the index is stored. The resulting sharding balances sequential read costs and random access costs for efficient query processing. We carried out extensive experiments and show that with no space overhead we consistently perform better than the state-of-art sliced indexes, upto a factor of 2.

Bibliography

[1] English Wikipedia. http://en.wikipedia.org/.

[2] European archive. http://www.europarchive.org.

[3] Internet archive. http://archive.org.

[4] Eytan Adar, Jaime Teevan, Susan T. Dumais, and Jonathan L. Elsas. The web changes everything: understanding the dynamics of web content. In *WSDM*, 2009.

[5] Omar Alonso, Michael Gertz, and Ricardo Baeza-Yates. On the value of temporal information in information retrieval. *SIGIR Forum*, 41(2):35–41, 2007.

[6] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Efficient Temporal Keyword Search over Versioned Text. In *CIKM*, 2010.

[7] Peter G. Anick and Rex A. Flynn. Versioning a full-text information retrieval system. In *SIGIR*, 1992.

[8] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4), 1996.

[9] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A Time Machine for Text Search. In *SIGIR*, 2007.

[10] Stefan Büttcher, Charles L.A. Clarke, and Gordon V. Cormack. Dynamic inverted indices. In *Information Retrieval - Implementing and Evaluating Search Engines*, chapter 7, pages 228–256. The MIT Press, 2010.

[11] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey Mogul. Rate of change and other metrics: a live study of the world wide

web. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

[12] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet Wiener. A large-scale study of the evolution of web pages. In *WWW*, 2003.

[13] Jinru He, Hao Yan, and Torsten Suel. Compact full-text indexing of versioned document collections. In *CIKM*, 2009.

[14] Jinru He, Junyuan Zeng, and Torsten Suel. Improved index compression techniques for versioned document collections. In *CIKM*, 2010.

[15] Michael Herscovici, Ronny Lempel, and Sivan Yogev. Efficient indexing of versioned document sequences. In *ECIR*, 2007.

[16] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to information retrieval.* Cambridge University Press, 2008.

[17] Peter Muth, Patrick E. O'Neil, Achim Pick, and Gerhard Weikum. The LHAM Log-Structured History Data Access Method. *VLDB J.*, 8(3-4):199–221, 2000.

[18] A. Ntoulas, J. Cho, and C. Olston. What's New on the Web?: The Evolution of the Web from a Search Engine Perspective. In *WWW*, 2004.

[19] Narayanan Shivakumar and Hector Garcia-Molina. Wave-indices: indexing evolving databases. In *SIGMOD*, 1997.

[20] Kenneth J. Supowit. Decomposing a set of points into chains, with applications to permutation and circle graphs. *Information Processing Letters*, 21(5):249 – 252, 1985.

[21] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

Below you find a list of the most recent research reports of the Max-Planck-Institut für Informatik. Most of them are accessible via WWW using the URL http://www.mpi-inf.mpg.de/reports. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
– Library and Publications –
Campus E 1 4

D-66123 Saarbrücken

E-mail: library@mpi-inf.mpg.de