A Comparison of Top-k Temporal Keyword Querying over Versioned Text Collections

Wenyu Huo and Vassilis J. Tsotras

Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA, USA
{whuo,tsotras}@cs.ucr.edu

Abstract. As the web evolves over time, the amount of versioned text collections increases rapidly. Most web search engines will answer a query by ranking all known documents at the (current) time the query is posed. There are applications however (for example customer behavior analysis, crime investigation, etc.) that would need to efficiently query these sources as of some past time, that is, retrieve the results as if the user was posing the query in a past time instant, thus accessing data known as of that time. Ranking and searching over versioned documents considers not only keyword constraints but also the time dimension, most commonly, a time point or time range of interest. In this paper, we deal with top-k query evaluations with both keyword and temporal constraints over versioned textual documents. In addition to considering previous solutions, we propose novel data organization and indexing solutions: the first one partitions data along ranking positions, while the other maintains the full ranking order through the use of a multiversion ordered list. We present an experimental comparison for both time point and time interval constraints. For time-interval constraints, different querying definitions, such as aggregation functions and consistent top-k queries are evaluated. Experimental evaluations on large real world datasets demonstrate the advantages of the newly proposed data organization and indexing approaches.

1 Introduction

Versioned text collections are textual documents that retain multiple versions as time evolves. Numerous such collections are available today and a well-known example is the collaborative authoring environment, such as Wikipedia [1], where textual content is explicitly version-controlled. Similarly, web archiving applications such as the Internet Archive [2] and the European Archive [3] store regular crawls (over time) of web pages on a large scale. Other time-stamped textual information such as, weblogs, micro-blogs, even feeds and tags, as also create versioned text collections.

If a text collection does not retain past documents, then a search query ranks only the documents as of the most current time. If the collection contains versioned documents, a search typically considers each version of a document as a separate document and the ranking is taken over all documents independently to the document's version (creation time). There are applications however, where this approach is not

S.W. Liddle et al. (Eds.): DEXA 2012, Part II, LNCS 7447, pp. 360–374, 2012.

adequate. Consider the following example: in order for a company to analyze consumer comments on a specific product before some event occurred (new product, advertisement campaign etc.), a temporal constraint may be very useful. For example, to view opinions on iphone4, a time-window within 06/07/2010 (announce date) and 10/04/2011 (announce date of iphone4s) could be a fair choice. Many investigation scenarios also require combining the keyword search with a time-window of interest. For example, while considering a financial crime, an investigator may need to identify what information was available to the accused as of a specific time instant in the past.

Providing "as-of" queries is a challenging problem. First is the data volume. Document collections like Wikipedia and Internet Archive, are already huge even if only their most recent snapshot is considered. When searching in their evolutionary history, we are faced with even larger data volumes. Moreover, how to quickly return the top-k temporally ranked candidates is another new challenge. Note that returning all qualified results without temporal constraints would not be efficient since two extra steps are required: (i) filtering out results later than the query specified time constraint, and, (ii) ranking the remaining results so as to provide the top-k answers.

In this paper we present an experimental evaluation of the top-k query over versioned text collections, comparing previously proposed as well novel approaches. In particular the key contributions can be summarized as:

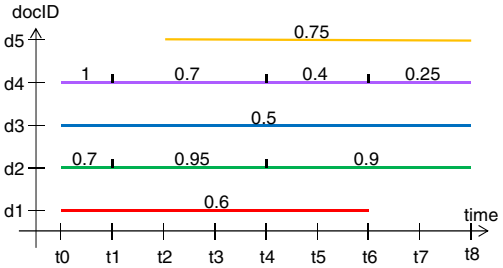
1. Previous methods related to versioned text keyword search are suitably extended for top-k temporal queries.
2. Novel approaches are proposed in order to accelerate top-k temporal queries. The first approach partitions the temporal data based on their ranking positions, while the other maintains the full rank order using a multiversion ordered list.
3. In addition to top-k time-point keyword based search, we also consider two time-interval (or time-range) variants, namely "aggregation ranking" and "consistent" top-k querying.
4. Experimental evaluations with large-scale real-world datasets are performed on both the previous and newly proposed methods.

The rest of the paper is organized as follows. Preliminaries and related work are introduced in section 2. Our novel approaches appear in section 3. Different query definitions of time-interval top-k queries are presented in section 4. All techniques are comprehensively evaluated and compared in a series of experiments in section 5 while the conclusions appear in section 6.

2 Preliminaries and Related Work

2.1 Definitions

The data model for versioned document collections was formally introduced in [10], and used by later works [9, 5, 6]. Let D be a set of n documents $d_1,d_2,\ldots,d_n$ where each document d_i is a sequence of m_i versions: $d_i = \{d_i^1, d_i^2, ..., d_i^{m_i}\}$. Each version has a semi-closed validity time-interval (or lifespan) $life(d_i^j) = [t_s, t_e)$. Moreover, it is



score-order

d4, 1, t0, t1
d2, 0.95, t1, t4
d2, 0.9, t4, t8
d5, 0.75, t3, t8
d2, 0.7, t0, t1
d4, 0.7, t1, t3
d1, 0.6, t0, t6
d3, 0.5, t0, t8
d4, 0.4, t3, t6
d4, 0.25, t6, t8

ts-order

d1, 0.6, t0, t6
d2, 0.7, t0, t1
d3, 0.5, t0, t8
d4, 1, t0, t1
d2, 0.95, t1, t4
d4, 0.7, t1, t3
d4, 0.4, t3, t6
d5, 0.75, t3, t8
d2, 0.9, t4, t8
d4, 0.25, t6, t8

Fig. 1. Example of versioned documents with scores for one term

assumed that different versions of the same document have disjoint life spans. An example of five documents and their versions appears in Fig. 1; each document corresponds to a colored line, while segments represent different versions of a document.

The inverted file index is the standard technique of text indexing for keyword queries, deployed in many search engines. Assuming a vocabulary V, for each term v in V, the index contains an inverted list L_v consisting of postings of the form (d, s) where d is a document-identifier and s is the so-called payload score. There are numerous existing relevance scoring functions, such as tf-idf [7], language models [13] and Okapi BM25 [14]. The actual scoring function is not important for our purposes; for simplicity we assume that the payload score contains the term frequency of v in d.

In order to support temporal queries, the inverted file index must also contain temporal information. Thus [10] proposed adding the temporal lifespan explicitly in the index postings. Each posting includes the validity time-interval of the corresponding document version: (d_i, s, t_s, t_e) where the document d_i had payload score s during the time interval $[t_s, t_e)$.

If the document evolution contains few changes over time, the associated score of most terms is unchanged between adjacent versions. In order to reduce the number of postings in an index list, [10] coalesces temporally adjacent postings belonging to the same document that have identical (or approximate identical) scores.

A general keyword search query Q consists of a set of x terms $q = (v_1, v_2, \ldots, v_x)$ and a temporal interval $[lb, rb]$. Without loss of generality, we use the aggregated score of a document version for keyword query q is the sum of the scores from each term v. The time-interval $[lb, rb]$ restricts the candidate document versions as a subset of the original collection: $D^{[lb,rb]} = \{d_i^j \in D \mid [lb, rb] \cap life(d_i^j) \neq \emptyset\}$. When $lb = rb$ holds, the query time interval collapses into a single time point t. For simplicity we first concentrate on time-point query and more complex time-interval queries are discussed in section 4 with related variations.

The answer R to a Top-K Time-Point keyword query **TKTP** = (q, t, k) over collection D is a set of k document versions satisfying: $\{d_i^j \in R \mid (\exists v \in q : v \in d_i^j) \wedge (d_i^j \in D^t) \wedge (\forall d' \in (D^t - R) : s(d_i^j) \geq s(d'))\}$ where $D^t = \{d_i^j \in D \mid t \in life(d_i^j)\}$. The first condition presents the keyword constraint, the second condition the temporal constraint, while

the third implies that the top-k scored document versions are returned. Now we present how to answer query TKTP using previous methods based on temporal inverted indexes.

2.2 Previous Methods

The straightforward way (referred to as **basic**) to solve query TKTP uses exactly one inverted list for each vocabulary term v with the posting (d_i, s, t_s, t_e). To answer the top-k queries, corresponding inverted lists are traversed and postings are fetched. When a posting is scanned, it is also verified for the time point specified in TKTP.

The sort-order of the index lists is also important. One natural choice is to sort each list in score order. This method (score-order) enables the classical top-k algorithms [11] to stop early after having identified the k highest scores with qualified lifespan. Another suitable sorting choice is to order the lists first by the start time t_s and then by score (t_s-order) which is beneficial for checking the temporal constraint. However, this approach is not efficient for top-k querying, especially when the query includes multiple terms. Fig. 1 shows the score-order and ts-order lists for a specific term.

Note that the efficiency of processing a top-k temporal query is influenced adversely by the wasted I/O due to read but skipped postings. We proceed with various materialization ideas of the slice the whole list of a term into several sub-lists or partitions thus improving processing costs.

Interval Based Slicing splits each term list along the time-axis into several sub-lists, each of which corresponds to a contiguous sub-interval of the time spanned by the full list. Each of these sub-lists contains all coalesced postings that overlap with the corresponding time interval. Note that index entries whose validity time-interval spans across the slicing boundaries are replicated in each of the spanned sub-lists.

The selection of the corresponding time-intervals where the slices are created is vital as discussed in [9, 5]. One obvious strategy is to eagerly slice sub-lists for all possible time instants (and adjacent identical lists can be merged). This will create one sub-list per time instant; this will provide ideal query performance for a TKTP query since only the postings in the sub-list for the query time point will be accesses. We refer to this method as **elementary**.

Note that the **basic** and **elementary** methods are two extremes: the former requires minimal space but requires more processing at query time since many entries irrelevant to the temporal constraint are accessed; the latter provides the best possible performance (for time-point query) but is not space-efficient (due to copying of entries among sub lists). To explore the trade-off between space and performance, [5] employs a simple but practical approach (referred to as **Fix**) in which a partition boundary is placed after a fixed time window. The window size can be a week, a month, a year, or other flexible choices. Fig. 2 shows the Fix-2 and Fix-4 sub-lists of our running example from Fig. 1, with the partition time window size as 2 and 4 time instants respectively. Nevertheless, all variations of the interval based slicing suffer from an index-size blowup since entries whose valid-time interval spans across the slicing boundaries are replicated.

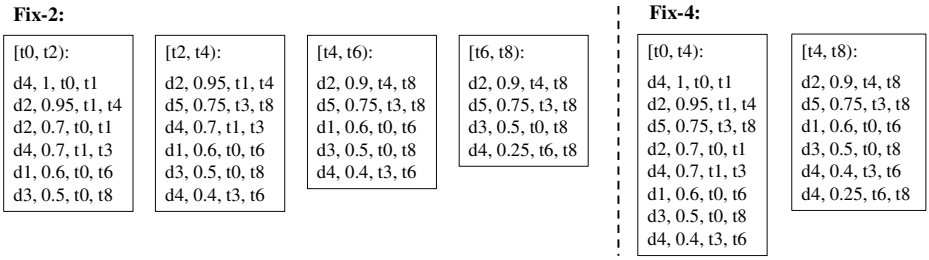


Fig. 2. Time Interval Based Slicing sub-list examples

Stencil Based Partitioning. Another index partitioning method along the time-axis was proposed in [12]. It is distinguished from the interval based slicing by using a multi-level hierarchical (vertical) partitioning of the lifespan. The inverted list of term v, at level L_0 contains the entire lifespan of this list, while level L_{i+1} is obtained from L_i by partitioning each interval in L_i into b sub-intervals. Such a partitioning is called a **stencil**; each index posting is placed into the deepest interval in the multi-level partitioning that fits its range. A stencil-based partition of three levels with $b = 2$ for the running example (from Fig. 1) is shown in Fig. 3.

Comparing to the time interval based slicing, the stencil based partitioning has significant advantage in space because each posting falls into a single list, the deepest sub-interval that it fits. Nevertheless, for a time-point query stencil based partitioning has to fetch multiple sub-lists, one from each level.

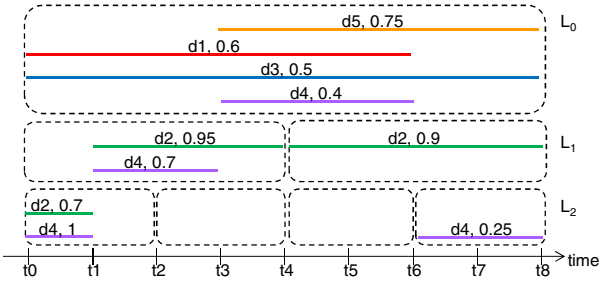
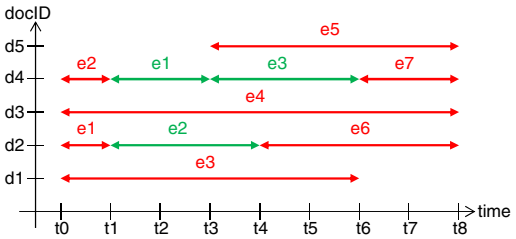


Fig. 3. Stencil-based partitioning with 3 levels and b = 2

The sort-order of each sub-list is again important. Since the temporal partitioning already shreds one full list into several sub-lists along the time-axis, a more appropriate choice for top-k queries is score-ordering.

Temporal Sharding. The approach proposed in [6] is to **shard** (or horizontally partition) each term list along the document identifiers instead of time. Entries in a term list are thus distributed over disjoint sub-lists called shards, and entries in a shard are ordered according to their start times t_s. So as to eliminate wasteful reads, within a shard g_i, entries satisfy a staircase property: $\forall p, q \in g_i, ts(p) \leq ts(q) \Rightarrow te(p) \leq te(q)$. An optimal greedy algorithm for creating this partitioning is given in [6]; an example of temporal sharding for the term list from Fig. 1, is shown in Fig. 4.



Shard_1:
e1: d2, 0.7, t0, t1
e2: d4, 1, t0, t1
e3: d1, 0.6, t0, t6
e4: d3, 0.5, t0, t8
e5: d5, 0.75, t3, t8
e6: d2, 0.9, t4, t8
e7: d4, 0.25, t6, t8

Shard_2:
e1: d4, 0.7, t1, t3
e2: d2, 0.95, t1, t4
e3: d4, 0.4, t3, t6

Fig. 4. Temporal sharding example

As with the stencil based approach, the space usage for temporal sharding is optimal since there are no replications of index entries. However, for query processing, all shards for each term need to be accessed, resulting in multiple sub-list readings. Moreover, the entries in each shard can only be time-ordered (based on start time t_s). Thus the benefit of score-ordering for ranked queries cannot be achieved, because all temporal valid entries have to be fetched.

3 Novel Approaches

A common characteristic of existing works is that they only consider the versioned documents on the time- and docID-axes, and try to partition the data along either direction. Instead, we view the index entries from a new angle -- namely, their score over time, and create index organizations to improve the performance of top-k querying. The *score-time* view of the example from Fig. 1 is shown in Fig. 5.

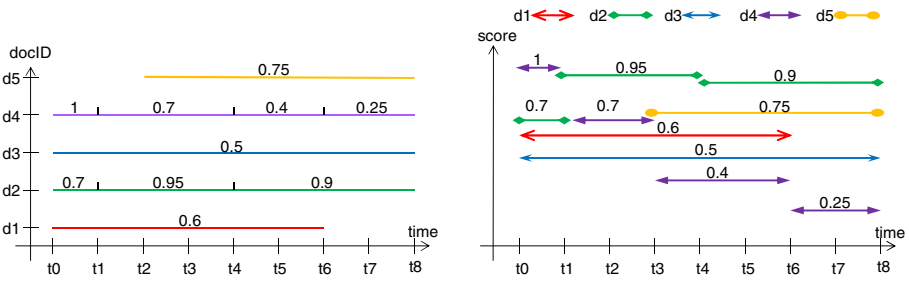


Fig. 5. The Score-Time view of the versioned documents

Recall that to answer a TKTP query we should be able to quickly find the top-k scores of a term at a given time instant. The main idea behind the score-time view is to maintain an index that will provide the top scores per term at each time instant. For example, at time t_0, the term depicted in Fig.5 had scores 1 (from d_4), 0.7 (from d_2), 0.6 (from d_1) and 0.5 (from d_3). These orderings change as time proceeds; for example at time t_2, the top score is 0.95 from d_2, etc. In *rank-based* partitioning (section 3.1), we first discuss a simplistic approach (SPR) where an index is created for each rank position of a term. For example, there is an index that maintains the top score over time,

then one for the second top score, etc. More practical is the group ranking approach (GR) where an index is created to maintain the group of the top-g scores (g is a constant), then the next top-g etc. We also consider temporal indexing methods (section 3.2). One solution is to use the Multiversion B-tree and maintain the whole ranked list in order over time. We realize however that these ranked lists are always accessed in order, so a better solution is provided (multiversion list) that links appropriately the data pages of the temporal index, without overhead of the index nodes.

3.1 Rank Based Partitioning

The **Single Position Ranking (SPR)** approach creates a separate temporal index for each ranking position of a term. Thus, for the i-th ranking position (i = 1,2,…), a sub-list is maintained that contains all the entries that ever existed on position i over time. Together with each entry we maintain the time interval during which this entry occupied that position. All sub-list entries are ordered based on their recorded starting time; a B+tree built on the start times can easily locate the appropriate entry at a given time. The SPR of our running example (from Fig. 1) is shown in Fig. 6(a). Space can be saved by using only the start time of each entry but for simplicity we show the end times as well (the end time is needed only if there is no entry in a particular position, but this is true only at the last position).

Using the SPR approach, to process a TKTP query about time t, the first k sub-lists have to be accessed for each relevant term; from each sublist the B+tree will provide the appropriate score (and document id) of this term at time t. If each sub-list has m items on average, the estimated time complexity is $O(k \cdot log_B m)$ (here B corresponds to the page size in records). Many sub-list accesses can degrade querying performance; moreover, in this simple SPR method the same posting can be duplicated in multiple ranking position sub-lists. This unavoidable replication may result in storage overhead.

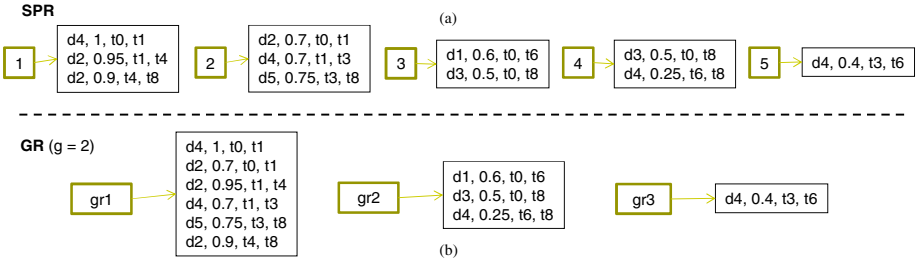


Fig. 6. Ranking position based partitioning

Group Ranking (GR). In order to save space and improve querying performance, GR maintains an index not for a single ranking position, but for a group of positions. Let the group size be g. For example, the first g ranked elements are in group gr_1, the next g ranked elements are in group gr_2, etc. Thus, compared to the n sub-lists maintained in SPR for n ranking positions, GR uses instead n/g sub-lists. With respect to the I/O of top-k querying, we only need k/g random accesses (each of them still logarithmic).

As with the SPR each member within a group also records the time interval that the member was in the group. For example, assume that group gr_i maintains ranking positions *(i-1)g+1* through *ig*. If at time t_s the score of a particular term falls within these positions, this score is added to the group, with an interval starting at t_s. As long as this score falls within the ranking positions of this group, it is considered part of the group; if at time t_e it falls out of the group, the end time of its interval is updated to t_e.

To save on update time, within each group we do not maintain the rank order. That is, each group is treated as an unordered set of scores that evolves over time. To answer a TKTP query that involves a particular group *gr* at time *t*, we need to identify what members group *gr* had at time *t*. Since the size of the group is fixed, we can easily sort these member scores and provide them to the TKTP result in rank order. However, it is guaranteed that given time *t*, the members in gr_i have no lower scores than those in group gr_j where $1 \le i < j \le (n/g)$. The GR approach for the above example (from Fig. 1) with $g = 2$ is shown in Fig. 6(b).

An interesting question is what index to employ for maintaining each group over time. Different than SPR, each group at a given time may contain multiple entries; thus a B+index on the temporal start times is not enough. Instead, temporal index structures that maintain and reconstruct efficiently an evolving set over time, like the snapshot index [15] can be used to accelerate temporal querying.

Note that when implementing GR in practice, each group may have a different size *g*. It is preferable to use smaller *g* for the top groups and larger *g* for the lower groups (since the focus is on top-k, the few top groups will be accessed more frequently and thus we prefer to give faster access). For simplicity however, we use the same *g* for all groups.

3.2 Using a Multiversion List

Consider the ordered list of scores that a term has over all documents at time *t*; as time evolves, this list changes (new scores are added, scores are promoted, demoted or even removed, etc). Temporal indexing methods have addressed a more general problem: how to maintain an evolving set of keys over time. This set is allowed to change by adding, deleting or updating keys; the main temporal query supported is the so called: temporal-range query: "given *t*, provide the keys that were in the set at time *t*, and are within key range *r*". The Multiversion B-tree (**MVBT**) proposed in [8], is an asymptotically optimal (in terms of I/O accesses under linear space) solution to the temporal range query. Assuming that there were a total of *n* changes that occurred in the set evolution, then the MVBT uses linear space ($O(n/B)$). Consider a range temporal query that specifies range *r* and time *t*, and let a_t denote the number of keys that were within range *r* at time *t* (i.e., the number of keys that satisfy the query); the MVBT answers the above query using $O(\log_B n + a_t/B)$ page I/Os, which is optimal in linear space [8].

In order for the MVBT to maintain order among the keys, it uses a B+tree to index the set. As the set evolves, so does the B+-tree. Conceptually the MVBT contains all B+-trees over time; for a given query time *t* the MVBT provides access to the root of the appropriate B+-tree, etc. Of course, the MVBT does not copy all B+-trees (as this would result in quadratic space). Instead it uses clever page update policies.

In particular, when a key k is added to the evolving set at time t a record is inserted in the (leaf) data page whose range contains k; this record stores key k and a time interval of the form: [t, *). The '*' denotes that key k has not been updated yet. If later at time t' this key is removed from the set, its record is not physically deleted. Instead this change is represented by changing the '*' to t' in this record's interval. A record is called "alive" for all time instants in its interval. Given a query about time t, the MVBT tree identifies all data pages that contain alive records for that time t. In contrast to a regular B+-tree that deals with pages that get underutilized due to record deletions, the MVBT pages cannot get underutilized because no record is ever deleted. Like the B+-tree pages can get full of records and need to be split (*page overflow*). However, the MVBT needs to also guarantee that the number of "alive' records in a page do not fall below a lower threshold l (*weak version underflow*) and also do not go over an upper threshold u (*strong version overflow*)- note that l and u are $O(B)$. If a page overflows, a time-split occurs, that copies the alive records of the overflown page (at the time of the overflow) to a new page. If there are too few alive records, the page is merged with a sibling page that is also first time-split. If there are too many alive records, a key split is first applied (among the alive records) [8].

Using the MVBT for our purposes means that the scores play the role of "keys". That is, the MVBT will maintain the order of scores over time. Since however term records are accessed by the docID they belong to, a hashing index is also needed that, for a given docID, it provides the leaf page that holds the record with this term's current score. This hashing scheme need only maintain the most current scores (i.e., it does not need to maintain past positions).

Nevertheless, the above MVBT approach has a significant overhead. In particular, it is built to answer queries about any range of scores. This is achieved by starting from an appropriate root of the MVBT and follow index nodes until the leaf data pages in the query range are accessed. For top-k processing however, we only access scores in decreasing order, starting with the largest score at a particular time instant. Thus, what we actually need, is a way to access the leaf page that has the highest scores at a particular time, and then follow to its sibling leaf page (with the next lower scores) at that time, etc. We still maintain the split policies among the leaf pages, but we do not use the MVBT's index nodes. Effectively we maintain a **multiversion list** (**MList**), i.e., of the leaf data pages over time.

To access the leaf data page that has the highest scores at a given time, we maintain an array A with records of the form (t, p) where t is a time instant and p is a pointer to the leaf page with the highest scores at time t. If later at time t' another page p' becomes the leaf page with the highest scores, array A is updated with a record (t', p'). If this array becomes too large for main memory, it can easily be indexed by a B+-tree on the (ordered) time attribute.

For the above "list of leaf pages" idea to work, each leaf page needs to "remember" the next sibling leaf page (with lower scores) at each time. (Note: the MVBT does not require the sibling pointers, since access to siblings is done through the parent index nodes). One could still use the array approach (one array responsible to keep access to the second leaf page, one for the third etc.) but this would require many array lookups at query time (each such lookup taking $O(log_B n)$ page I/Os. Instead, we propose

to embed these arrays within the page structure. That is, within each leaf page, we allocate a space of c records (where c is a constant) for the sibling page pointer records (also of the form (t,p)). As a result, each leaf page has now space for B-c score records. Since however, the sibling page can change over time, it is possible that for a leaf page p the sibling will change more than c times. If this happens at time t, page p is "time split", that is, a new leaf page p' is created containing only the currently alive records of page p and with an empty array for sibling pointers. Moreover, p' replaces p in the list. If before t, the list of leaf pages contained pages (in that order) $m \rightarrow p \rightarrow v$, a new record (t, p') is added in the array of page m, and the array of page p' is initialized with a record (t,v). If p was the first page, the record (t,p') is added to array A.

The advantage of the **Mlist** approach is apparent at query processing time. A search is first performed within array A for time t (in $O(log_B n)$ page I/Os). This will provide access to the page with the highest scores at time t. Find the next sibling page at time t however will be provided by looking among the c records of this page, etc. That is, the top-k scores at time t will be accessed in $O(log_B n + k/B)$ page I/Os. The justification is that after the access to array A, each leaf page (except possibly the last one) will provide $O(B)$ of the top-k scores (since we are using the MVBT splitting policies within the B-c space of each leaf page and c is a constant, each page is guaranteed to provide at least $l=O(B)$ scores that were valid at the query time t.

4 Top-k Time Interval Queries

Until now we focused on the top-k time point (TKTP) querying, and analyzed different index structures for solving it. We proceed with the time interval top-k query. The main difference is that in the TKTP, each document has at most one valid version at the given time point t; while for an interval querying, each document may have multiple versions valid during the given time interval $[lb, rb]$. As a result, there are different variations, depending on how the top-k is defined (which of the valid scores per document participate in the top-k computation). Here, we summarize the different definitions of top-k time-interval queries and discuss how to process them efficiently within the proposed index structures.

4.1 Classic Top-k Time-Interval Query

This query definition is a straight forward extension from the top-k time point query. For a Top-K Time Interval keyword query **TKTI** = (q, lb, rb, k) over collection D, we require the answer R be a set of k document versions satisfying: $\{d_i^j \in R \mid (\exists v \in q : v \in d_i^j) \wedge (d_i^j \in D^{[lb,rb]}) \wedge (\forall d' \in (D^{[lb,rb]} - R) : s(d_i^j) \geq s(d'))\}$ where $D^{[lb,rb]} = \{d_i^j \in D \mid [lb, rb] \cap life(d_i^j) \neq \emptyset\}$. This definition only changes the time constraints from a time point t to a time range $[lb, rb]$. The returned top-k answers are different versions, which may be from the same document, that is, we consider each document version as an independent object.

Processing a TKTI query is similar to processing a TKTP query. For some of the described index methods, multiple sub-lists have to be accessed instead of one.

For example in time interval based slicing and stencil based partitioning, all the sub-lists (or stencils) overlapping with the query time-interval should be checked in order to find the correct top-k results. The multiple parallel sub-lists can be accessed in a round-robin fashion which is compatible with top-k algorithms.

4.2 Document Aggregated Top-k Time-Interval Query

Another possibility is to treat each document as one object, that is, a document appears at most once in the result. There are various approaches in aggregating relevance scores of the document versions that existed at any point in the temporal constraint [*lb*, *rb*] to obtain a document relevance score *drs*(d_i, *lb*, *rb*). Three aggregation relevance models are mentioned in [10]:

MIN. This model judges the relevance of a document based on the minimum score. It is formally defined as: $drs(d_i, lb, rb) = \min\{s(d_i^j) \mid [lb, rb] \cap life(d_i^j) \neq \emptyset\}$. The MIN scores of our five-document example for interval [t_0, t_8) are d_2=0.7, d_3=0.5, d_4=0.25, d_1=0, d_5=0.

MAX. In contrast, this model takes the maximum score as an indicator. It is formally defined as: $drs(d_i, lb, rb) = \max\{s(d_i^j) \mid [lb, rb] \cap life(d_i^j) \neq \emptyset\}$. MAX scores of our five-document example for interval [t_0, t_8) are d_4=1, d_2=0.95, d_5=0.75, d_1=0.6, d_3=0.5.

TAVG. Finally, the TAVG model assigns the score to each document using a temporal average among all its valid versions. Since score $s(d_i^j)$ is piecewise-constant in time, *drs*(d_i, *lb*, *rb*) can be efficiently computed as a weighted summation of these segments. TAVG scores of our five-document example for interval [t_0, t_8) are d_2=0.89, d_4=0.51, d_3=0.5, d_5=0.47, d_1=0.45.

After the aggregation mechanism has been defined, one can consider the Aggregated Top-K Time-Interval keyword query $\mathbf{TKTI^A}$ = (*q*, *lb*, *rb*, *k*) over collection D, that finds the top k documents with aggregated scores over all their valid document versions. To process the aggregated top-k time-interval query, we need to extend the traditional top-k algorithms (such as TA and NRA) by recording the bookkeeping information and computing the scores and thresholds with candidates at document-level. The relevance score of a document in the query temporal-context depends on the scores of its version that are valid during this period.

4.3 Consistent Top-k Time Range Query

The consistent top-k search finds a set of documents that are consistently in the top-k results of a query throughout a given time interval. The result of this query has size 0 to *k*; queries can have empty results if *k* is small or the rankings change drastically. A relaxing consistent top-k query utilizes a relax factor *r*, $0 < r <= 1$, and seeks for documents that are in the top-k for at least $r \times (rb - lb)$ time in the [*lb*, *rb*] interval. For a Consistent Top-K Time Interval keyword query $\mathbf{TKTI^C}$ = (*q*, *lb*, *rb*, *k*) over collection *D*, the documents in the answer *R* are in the top-k for at least $r \times (rb - lb)$ time in the [*lb*, *rb*] interval. The consistent top-3 query of our five-doc example for time-interval [t_0, t_8) has only one result as d_2 if $r = 1$, and has three results as d_1, d_2 and d_5 if $r = 0.6$.

In [16] several algorithms were introduced to answer the consistent top-k query; the most efficient ones are based on the assumption that there is a list containing all versions satisfying the keyword and time interval constraints and the list is ordered by score. This assumption coincides with the purpose of our proposed index structures, thus we can access the qualified entries and execute the consistent top-k time interval query using the proposed approaches in [16].

5 Experimental Evaluations

Dataset Description and Methods Implemented: We used news-like articles as our primary versioned document collection. We collected US and world-wide English newspaper websites and treated each URL as a single document. Then their historical homepage versions were retrieved by crawling the Internet Archive [2] from 1997.1.1 until 2011.12.31. We created two different datasets with daily unit time granularity. The US based news had many frequent updates. The size of raw data is about 0.2 TB, with 12,649 documents and 1,542,893 versions; thus on average there are 122 versions per document in the US dataset. For the world-wide news websites, the size of raw data is about 50 GB, with 5,046 documents and 275,981 versions, so on average there are 55 versions per document. Previous related works create query workloads by extracting frequent queries from the AOL query logs. In addition to this traditional query workload, we use popular keywords (such as "twitter", "iphone", "lady gaga" etc.) from the Google Zeitgeist [4] annual reports from 2001 until 2011. Overall, we formed 200 queries with 265 terms for both classic and popular keywords.

We organize the data into term inverted list(s) using the previous and novel approaches. In the basic method with score-ordering (referred to as **Basic-s**) we create one inverted list per term. The second method is elementary time-interval slicing with a merging of adjacent identical sub-lists (**Ele**). For the Fix approach we used a time-window length of 30 days (**Fix-30**). The stencil based partitioning was implemented with 3 levels and $b = 4$ (**Stencil**). Temporal sharding is referred as **Shard**, while the single position ranking model appears as **SPR**. Two group ranking methods were implemented with group sizes of 25 and 50 (**GR-25** and **GR-50**). For comparison purposes we also included the **MVBT** index (with the appropriate hashing secondary index).The multiversion list approach (**MList)** uses a factor $a = c / B$ to present the ratio of the number of pointer records to the number of all records in a page.

Comparison Results: First, the space usage for all implemented methods on both the US-news and World-news datasets is presented in Table 1. The page size is 4 Kbytes while $B = 100$ records. The table presents the space consumed (in GB) to implement the index methods for the 256 terms used in our experiments. Clearly, the elementary time-interval slicing has a huge space overhead while the Stencil and Shard methods present substantial space savings. As expected, the Basic-s approach has the minimal space requirements. Fix-30 uses more space since a record may appear in more partitions while in Stencil and Shard, each record appears once. The additional space that Stencil and Shard use wrt Basic-s is due to the additional structures they utilize. Among the rank-based partitioning methods, SPR uses more space than the GR approaches; this is because the SPR approach has to maintain one index per ranked position. GR-25 uses more space than GR-50 since it uses more index structures

(one per group). For the MList method, we show the results of a = 7% and a = 10% (referred to MList-7 and MList-10). The MList approaches also use linear space (but due to the copying of records at page splits, the space is more than the Stencil and Shard approaches). MList uses slightly more space than the MVBT because of the use of sibling pointers and the splits they create.

Table 1. The space usage (in GB) for the 256 terms used in the queries

Methods	Basic-s	Ele	Fix-30	Stencil	Shard	SPR	GR-25	GR-50	MVBT	MList-7	MList-10
US	**1.93**	213.34	4.26	2.24	2.31	6.65	6.12	5.93	3.79	4.02	3.95
World	**0.35**	38.6	0.78	0.41	0.43	1.21	1.12	1.06	0.69	0.75	0.78

The top-k temporal queries include both time-point (in our dataset this corresponds to one day) and time-interval queries. For each temporal keyword query, we randomly choose 50 time constraints from the 15-year lifespan from 1997 to 2011, and record the average performance. For TKTI^A, we use TAVG scoring; for TKTI^C, we use r = 1. The page I/O costs for top-20 queries using the US-news dataset are shown in Table 2 (the best performance for each query is shown in bold). For time interval queries, the time-interval lengths used were 15 days, 30 days, and 60 days. We also present the I/O costs for top-100 queries on both US-news and World-news datasets in Table 3 for both time-point query and 30-day time-interval queries.

Table 2. The page I/O cost of top-20 temporal keyword queries for US news

Methods	Basic-s	Ele	Fix-30	Stencil	Shard	SPR	GR-25	GR-50	MVBT	MList-7	MList-10
TKTP	49.16	**3.74**	7.44	11.16	87.5	33.24	6.26	7.8	5.34	5.58	5.02
TKTI-15	65.32	56.22	13.9	16.5	90.64	40.74	14.92	17.26	11.46	11.7	**11.18**
TKTI-30	81.76	108.7	16.48	20.42	93.58	45.9	19.32	22.88	15.74	16.22	**15.28**
TKTI-60	105.6	195.82	31.26	35.8	95.22	49.66	23.06	26.14	22.38	22.9	**21.7**
TKTIA-15	74.16	67.84	20.8	24.12	96.54	48.38	20.42	22.8	18.68	19.54	**16.92**
TKTIA-30	89.84	126.4	23.18	27.84	98.3	50.1	25.78	26.2	21.9	23.84	**21.42**
TKTIA-60	112.96	209.56	41.06	46.76	103.86	60.22	31.14	33.84	30.32	30.82	**29.68**
TKTIC-15	68.48	60.6	17.42	19.48	92.82	44.34	16.68	19.12	14.04	14.58	**13.74**
TKTIC-30	83.52	110.58	19.5	22.38	96.04	47.48	21.5	24.04	18.18	20.36	**17.44**
TKTIC-60	108.34	201.42	35.74	39.22	98.72	53.82	26.7	27.98	25.6	26.24	**24.18**

Table 3. The page I/O cost of top-100 temporal keyword queries for US and World news

US	Basic-s	Ele	Fix-30	Stencil	Shard	SPR	GR-25	GR-50	MVBT	MList-7	MList-10
TKTP	93.4	**10.14**	25.74	38.7	102.68	162.4	29.64	21.18	20.72	21.84	19.12
TKTI-30	157.84	315.3	48.62	70.22	114.2	233.94	92.82	62.94	46.92	49.38	**46.24**
TKTIA-30	171.8	336.44	53.5	79.18	118.24	241.48	115.74	75.32	52.1	55.92	**51.48**
TKTIC-30	163.52	324.86	50.26	73.42	115.7	236.5	101.36	67.28	49.06	52.06	**48.2**
World	**Basic-s**	**Ele**	**Fix-30**	**Stencil**	**Shard**	**SPR**	**GR-25**	**GR-50**	**MVBT**	**MList-7**	**MList-10**
TKTP	85.44	**9.96**	23.36	37.52	98.8	156.44	27.5	20.84	20.12	18.22	18.84
TKTI-30	143.32	306.58	45.74	69.62	110.28	228.36	83.34	54.62	45.32	**44.78**	45.1
TKTIA-30	152.7	322.36	50.82	77.84	115.66	237.02	103.6	70.7	51.16	**49.82**	50.56
TKTIC-30	147.24	311.92	47.78	72.16	112.72	231.84	91.76	62.58	47.24	**46.3**	46.82

The elementary time-interval slicing has the best snapshot querying performance for both top-20 and top-100 queries. This is to be expected since the answer is basically prepared for each time instant (at the cost of huge storage requirements). Among the other methods, the newly proposed approaches (GR, MList) outperform

the previous methods (Stencil and Shard). The best performance is provided by the MList-10 method. It has better performance than the MVBT given it accesses the answer faster (by avoiding the MVBT index traversal). Considering its low space requirements, this approach provides the overall best performance for TKTP queries.

For time-interval queries, the Ele method's performance degrades drastically, especially for longer time-interval. The group ranking method's performance is related to its group size g as it relates to k. For top-20 querying, a group size of 25 works better than a group size of 50 (the answer can be found by accessing the first group only); while for top-100 querying, GR-50 is a better choice (only two groups need to be accessed instead of four for GR-25, thus less index accesses). For top-20 interval queries, the MList-10 had consistently the best performance for each query.

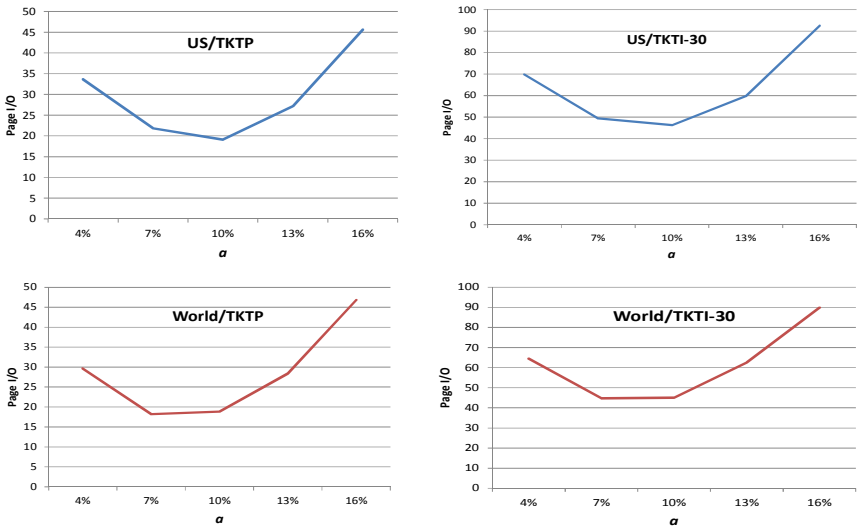


Fig. 7. The Multiversion list method for different ratio a using the US and World news datasets

Interestingly, for the top-100 interval queries the MList-7 shows better performance for the World-news dataset. The reason for that is that this dataset has fewer updates. As a result, there will be fewer pointer changes in the ordered list, thus a smaller a will provide enough space to hold the pointer structure. This can also be seen in the space requirements for this dataset: the fewer pointer splits mean that MList-7 uses less space than MList-10 (and thus the lists are shorter and the query performance better). The above observation implies that the performance of the multiversion list method is related to the value of a. There are two opposing factors affecting the query performance with respect to a. For a given page size, a small a implies that the area allocated to sibling pointers is small; thus few sibling page changes can cause the page to split. More splits use more space and the query time increases. On the other hand, a large a implies that the space allocated for the regular records in a page is small, thus the page can split faster due to the record updates. This also increases space and query time. The optimized value of a depends on the dataset characteristics. Figure 7 depicts the page I/O for the top-100 results returned by point

(TKTP) and interval (TKTI-30) queries for the US and World-news datasets. For the US-news dataset, a = 10% has the best average performance for both time-point querying (TKTP) and 30-day time-interval querying (TKTI-30) while for the World-news dataset (which has less update frequency), the performance is optimized for a = 7% .

6 Conclusion

We presented an experimental comparison of indexing methods over versioned text collections for top-k temporal keyword queries. In addition to previous methods, we proposed novel solutions that partition the data along the score-time axes. Among all methods, the multiversion list provided the most robust performance considering space usage and query time efficiency for both time-point and time-interval queries. We examined variations of the time-interval queries, including the document-level aggregated top-k queries and consistent top-k queries. The performance of the multiversion list is affected by the value of a, the percentage of a data page allocated to hold sibling pointers. As future work, we plan to devise a model that can optimize the value of a based on the frequency of updates, the size of the page and other factors.

Acknowledgements. This work is partially supported by NSF grant IIS-0910859.

References

[1] Wikipedia, `http://en.wikipedia.org/`
[2] Internet Archive, `http://www.archive.org/`
[3] European Archive, `http://www.europarchive.org/`
[4] Google Zeitgeist, `http://www.google.com/zeitgeist/`
[5] Anand, A., Bedathur, S., Berberich, K., Schenkel, R.: Efficient Temporal Keyword Queries over Versioned Text. In: CIKM (2010)
[6] Anand, A., Bedathur, S., Berberich, K., Schenkel, R.: Temporal Index Sharding for Space-Time Efficiency in Archive Search. In: SIGIR (2011)
[7] Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley (1999)
[8] Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion B-tree. VLDB Journal (1996)
[9] Berberich, K., Bedathur, S., Neumann, T., Weikum, G.: A Time Machine for Text Search. In: SIGIR (2007)
[10] Berberich, K., Bedathur, S., Weikum, G.: Efficient Time-Travel on Versioned Text Collections. In: BTW (2007)
[11] Fagin, R., Lotem, A., Naor, M.: Optimal Aggregation Algorithms for Middleware. J. Comput. Syst. Sci. 66(4), 614–656 (2003)
[12] He, J., Suel, T.: Faster Temporal Range Queries over Versioned Text. In: SIGIR (2011)
[13] Ponte, J.M., Croft, W.B.: A Language Modeling Approach to Information Retrieval. In: SIGIR (1998)
[14] Robertson, S.E., Walker, S.: Okapi/keenbow at TREC-8. In: TREC (1999)
[15] Tsotras, V.J., Kangelaris, N.: The Snapshot Index: an I/O Optimal Access Method for Snapshot Queries. Information System 20(3), 237–260 (1995)
[16] U, L.H., Mamoulis, N., Berberich, K., Bedathur, S.: Durable Top-k Search in Document Archives. In: SIGMOD (2010)