UNIVERSITY OF CALIFORNIA
RIVERSIDE

Query Processing on Temporally Evolving Social Data

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Wenyu Huo

June 2013

Acknowledgements

This dissertation is based on my research work at UC Riverside. I have been very lucky to receive the support and guidance from many remarkable people. It is my pleasure to give thanks to them.

First and foremost, I would like to thank my advisor, Professor Vassilis J. Tsotras. I have had the very good fortune to work with and learn from him. He taught me how research is done and especially what makes an influential research. He spent numerous hours with me discussing research ideas and editing paper drafts. I have truly appreciated all the advice and support he has have given me throughout my research study.

I would also like to thank other committee members, Professor Eamonn Keogh and Professor Vagelis Hristidis, for their valuable feedbacks on this dissertation.

Many people helped to make my years at UCR a very enjoyable experience. I especially thank the Chinese Ph.D. folks at UCR CS, which include Qiang Zhu, Chen Huang, Jilong Kuang, Zi Feng, and Jianxia Ning. The weekly poker nights have made this Ph.D. career much more tolerable. My thanks also go to the members of the database research group, including Jian Wen, Michael Rice, Md. Mahbub Hasan, Marcos R. Vieira, and Mariam S. Salloum. They have offered valuable discussion and inspiring insights.

I would like to give my special thanks to my parents, Zhisheng Huo and Weihong Liu. They raised me up and always gave me unconditional support.

Finally, I am deeply indebted to my beautiful wife, Peihui Zhang. She willingly chose to marry and support me through all the ups and downs of graduate lift. Her love, care, and trust has been the source of my strength and courage. My son, Jiachen, was born during my Ph.D. years. His smile and happiness cheered me up every day.

ABSTRACT OF THE DISSERTATION

Query Processing on Temporally Evolving Social Data

by

Wenyu Huo

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2013
Dr. Vassilis J. Tsotras, Chairperson

The continuous growth of the internet and the popularity of social networks have created a huge amount of social media data. This includes social networks like users' friendships, as well as users' contributed content such as tags, blogs, posts, tweets, and etc. In addition, other collaborating applications also generate large data, such as the versioned textual documents created in a collaborative authoring environment like Wikipedia. In a dynamic world, the social media data is continuously evolving with time. In December 2004, Facebook had about 1 million users; but by October 2012, Facebook has over 1 billion active users. The dynamically changing and rapidly growing data bring us critical challenges: how to store, how to query, and how to use it in different application domains. This dissertation examines four related problems. First, we consider the large historical evolving graphs created from a social network, and examined various *temporal shortest-path* queries (e.g., find the shortest-path between two nodes as of certain time in the past). For this environment we proposed an efficient storage model, and fast query processing algorithms that take advantage of appropriate speed-up indexing techniques. For second problem examined, deals with *social tagging websites*, where users post and share items like bookmarks, videos, photos etc., along with comments and tags. Within this environment, we presented a study of top-k search that utilizes the temporal information as well as a user's participation in multiple social networks; our results show an improved search performance. Third, we examined the problem of *temporal top-k keyword search* in versioned textual collections; we compared different approaches and proposed novel methods that utilize multi-version access methods to improve the search. Finally, we considered applications that support multi-*version schema evolutions*; we explored scenarios for branching and merging, and proposed efficient indexing structures along with query processing optimizations.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The continuous growth of the internet and the popularity of social networks have created huge amount of data. Typical social media includes the social networks like users' friendships, as well as users' contributed contents such as tags, blogs, posts, tweets, and etc. Similarly, collaborating social applications also data, such as the versioned textual documents created in a collaborative authoring environment like Wikipedia. In a dynamic world, such data is continuously evolving with time. Taking the number of users as an example, in December 2004, Facebook had about 1 million users; by October 2012, Facebook has over 1 billion active users. The dynamic evolutions of social media data bring us critical challenges: how to store, how to query, and how to use it in different application domains. In this dissertation, we consider problems related to temporal querying over social data applications. The four problems we studied can be summarized as: (i) temporal shortest-path querying over evolving social graphs, (ii) top-k search in social tagging websites by using multiple networks and temporal information, (iii) temporal top-k keyword search in versioned textual collections from social collaborating applications, and (iv) temporal querying for applications supporting branched schema evolutions.

Query evaluation over *evolving social graphs* is important and challenging. Different from traditional studies of shortest-path queries on a single graph, our main objective is to efficiently answer temporal shortest-path queries within the evolving graph's history. Considering an evolving social graph over a large temporal period (years), an example query would be to find the shortest-path between two users as of some past time. Note that the evolving graph is not stored as a separate snapshot at each time (this would require even more space), neither as a sequence of deltas (which would result in long query times). Rather, the space used to store such a graph is typically linear to the changes in the graph evolution but can still support fast query times. Shortest-path queries are a basic component for many other graph-related queries (trend analysis etc.) For example, using temporal shortest path queries in an evolving social network we can discover how close two given users were in the past, and how this closeness was changed over time.

Our work on the temporal shortest-path query is distinguished from previous studies in four ways: (1) In order to reduce the storage overhead and to efficiently support time-interval querying as well, we store the graph evolution into one "integrated" temporal graph, instead of a sequence of snapshots or deltas. (2) Our temporal shortest-path queries can be specified for any given time-point or time-interval, while past works have considered querying over the whole graph life-time. (3) We explore preprocessing index techniques, which are very effective and efficient. (4) Further enhancements like temporal partitioning and their effects on the shortest-path query processing are discussed. To demonstrate our algorithms and optimizations, we do experimental evaluations on real-world social network datasets collected over long time periods.

With their increasing popularity, *social tagging sites* store valuable information like user-generated items, user social networks, and user tags. Such information can be used to improve services such as hot-lists, recommendations and web search; top-k search in social tagging sites has thus attracted research interest from both academia and industry.

Here we focus on temporal top-k search in social tagging sites. When compared to other works, our contributions are: (1) we apply multiple components to score an item with respect to a particular user's different social networks and assign weights to each component based on the classification of that user's participation in those networks. (2) We take into consideration the temporal information of tagging behaviors, in order to enhance popularity and freshness of the top-*k* results. (3) Last, we provide a variation of the classic top-*k* algorithm which works efficiently for our user-dependent temporal scoring functions. Experimental evaluations on real social tagging datasets show that our framework works well in practice.

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 a collaborative authoring environment, such as Wikipedia. If a text collection does not retain past documents, then a search query ranks only the documents as of the most current time. Even 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 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.

To answer that question, we need queries that can identify the top-k result with *both keyword and temporal constraints* over versioned textual documents. In particular: (1) We propose novel data organization and indexing solutions: The first approach partitions the temporal data based on their ranking positions, while the other maintains the full rank order using a multi-version ordered list. (2) In addition to top-k *time-point* keyword based search, we also consider two *time-interval* variants, namely "aggregation ranking" and "consistent" top-k querying. (3) We present experimental evaluations comparing our approaches to previous solutions, using large-scale real-world datasets.

Due to the collaborative nature of web applications, information systems experience evolution not only on their data content but also under *different schema versions*. For example, Wikipedia has experienced more than 170 schema changes in its 4.5 years of lifetime. In many applications, the schema may change into multiple branches. For instance, in a collaborative design environment, an initial schema may be branched into a number of parallel schemas whose data can evolve concurrently.

We address the issues to examine both data and schema evolution in a branched evolution environment. In particular: (1) We utilize a sharing strategy with lazy-mark updating, to save space and update time when maintaining the schema branching. (2) We employ branched temporal indexing structures and link-based algorithms to improve temporal query processing over the data. (3) Moreover, we propose various optimizations for two novel temporal queries involving multiple branches, the vertical and horizontal queries. (4) We further examine how to support version merging within the branched schema evolution environment. Our experiments show the space effectiveness of our sharing strategy while the optimized query processing algorithms achieve great data access efficiency.

Chapter 2

Efficient Temporal Shortest Path Queries on Evolving Social Graphs

Graph-like data is widely used in many applications, such as social networks, internet hyperlinks, roadmaps, bioinformatics, etc. In most of these applications, graphs are dynamic (evolving) as changes are applied through time. In this work, we study the problem of efficient shortest-path query evaluation on evolving social graphs. Our shortest-path queries are "temporal": they can refer to any time point or time interval in the graph's evolution, and corresponding valid answers should be returned. To efficiently support this type of temporal query, we extend the traditional Dijkstra's algorithm to compute shortest-path distance(s) for a time-point or a time-interval. To speed up query processing, we explore the bi-directional search method as well as preprocessing index techniques such as Contraction Hierarchies (CH) and Goal-directed Landmark-based A* search (ALT). Moreover, we examine how to maintain the evolving graph along with the indexing. Experimental evaluations on real world datasets demonstrate the feasibility and efficiency of our proposed algorithms and optimizations.

2.1 Introduction

Graphs have been used as a general data structure to model numerous modern applications, such as social networks, internet hyperlinks, roadmaps, bioinformatics, etc. For example, in a social network application like Facebook, registered users can be considered as vertices with edges representing friendships between them. In a dynamic world, users and friendships are continuously evolving with time. In December 2004, Facebook had about 1 million users; by October 2012, the number of active Facebook users had increased to 1 billion. Similarly, edges are continuously added or deleted as new friendships are formed or old ones are broken. This dynamically changing environment brings critical challenges: how to store the evolution of large-scale graphs and how to efficiently support query evaluations.

The shortest-path query is among the fundamental operations on graph data, as the shortest-path distance is important in measuring "closeness" between nodes. In social networks, users may be comfortable with adding close users as their friends, and users may be interested in finding contents from users that are close to them in the social graph. Computing the shortest-path distances efficiently is thus crucial for a variety of applications.

Different from traditional studies of shortest-path queries on a single graph, our main objective is to efficiently answer *temporal shortest-path queries* within the graph evolving histories. Such temporal queries can be viewed as being issued on certain historical graph snapshot(s). This type of temporal query is not only essential for searching and retrieving histories, but also useful for trend analysis. For example, temporal shortest-path queries in a social network can discover how close two given users were in the past and how their closeness evolved over time. However, in many scenarios, even a single snapshot graph is already very large; maintaining the evolving graph history has much greater volume in data storage and brings more challenges in querying.

2.1.1 Related Work and Our Contributions

In recent years, plenty of research work has studied efficient shortest-path querying of large graph data. To improve query times, several preprocessing indexes have been proposed; a survey of route planning is provided by [18]. Nearly all of these techniques rely on some variant of the classical Dijkstra's algorithm [19]. These existing researches on preprocessing indexes can be classified into three general categories: *hierarchical* methods, *goal-directed* searches, and *combinations* of the two. Hierarchical methods (such as Highway Hierarchies [48], Transit Node Routing [7], and Contraction Hierarchies [22]) seek to order the nodes and/or edges within the graph into hierarchically nested levels. Goal-directed techniques (such as arc-flags [28] and ALT [24]) try to direct the shortest-path search toward certain explicit target nodes. However, most previous works focus only on a single (i.e. non-temporal) graph snapshot. There is also recent work on query processing techniques for time-dependent graphs [20] and dynamic graphs [14], but are different from our problem that computes temporal shortest-path distances on evolving graphs.

To the best of our knowledge, the most relevant works are [35, 45]. In particular, [35] addressed the problem of evaluating historical queries on graphs. Its temporal query types, namely the point and range queries, are very close to our time-point and time-interval query definitions. However, its storage model maintains the current graph and deltas to previous time snapshots; as a result, the first step of evaluating a historical shortest-path query is to first reconstruct the corresponding snapshot or snapshots that relate to the query's temporal predicate. Such a reconstruction phase can be costly; moreover, traditional speed-up preprocessing techniques such as CH and ALT are very difficult to incorporate in this storage framework.

Another storage approach was proposed in [45], namely, the historical evolving graph sequence (EGS). Various snapshots and deltas are explicitly stored, but in addition, temporally close snapshots are clustered together. Graph-based queries (like shortest-path and closeness centrality) are answered for the whole graph history (not a single time point or small time interval); this is done efficiently with the help of a Find-Verify-Fix (FVF) framework [45].

Our work is distinguished from previous studies in various ways: 1) In order to reduce storage overhead and support time-interval querying efficiently, we store the historical evolution in one "integrated" temporal graph instead of a sequence of snapshots or clusters and their deltas. 2) We explore preprocessing index techniques for the temporal evolving graph query processing, which are very effective and efficient. 3) We explore further enhancements like temporal partitioning.

The rest of this chapter is organized as follows. In section 2.2, the temporal evolving graph model is proposed along with temporal shortest-path querying definitions. In section 2.3, the fundamental solutions are explored as the extensions of Dijkstra's algorithms, while section 2.4 describes speedup techniques such as the bi-directional search method and preprocessing indexes like CH and ALT. Section 2.5 discusses further optimizations, in particular how temporal partitioning affects the processing of temporal queries. Section 2.6 presents our experimental analysis and section 2.7 concludes the chapter with future work.

2.2 Temporally Evolving Graph

2.2.1 Graph Data Model

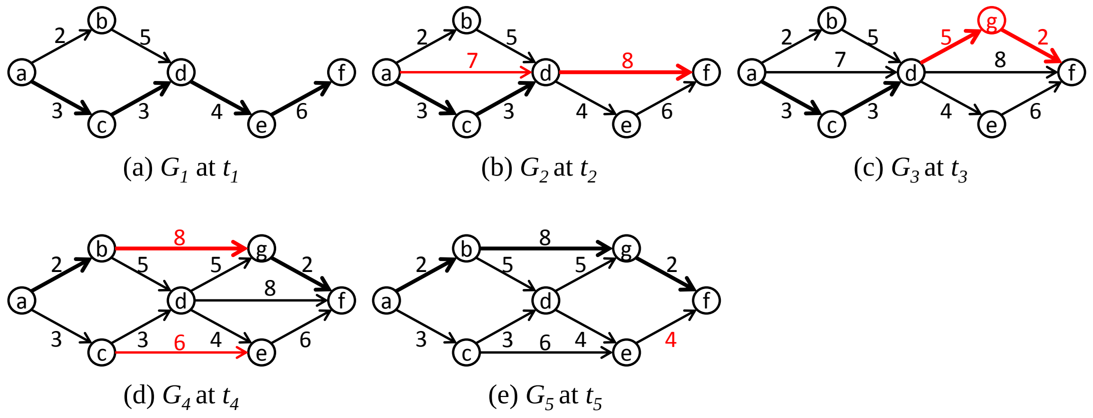
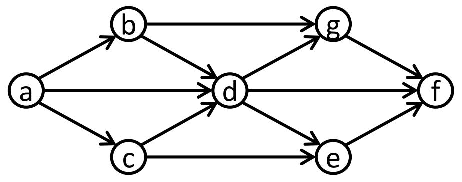


Figure 1: Example of temporal evolving graph

A single static graph, either directed or undirected, can be modeled as $G = (V, E)$, where V is the set of nodes and E is the set of edges. If G is a weighted graph, there is a weight function $w : E \rightarrow \mathbb{R}^+$ mapping edges in G to a positive, real-valued weight. An edge is represented as a triplet <n_1, n_2, w> : i.e., this edge is from node n_1 to node n_2 with weight w. If the graph evolves with time, a different graph snapshot exists logically at each time. For example, as shown in Figure 1(a), the graph G_1 at time t_1 had six nodes and six directed edges. From then, until the latest time t_5, there are five graph snapshots with four updates. Each update may contain multiple operations including: node insertion, node deletion, edge insertion, edge deletion, and edge weight adjustment. This graph evolution creates a Graph Sequence (GS), $GS = (G_1, G_2, G_3, G_4, G_5)$. To maintain this graph sequence in a space-efficient way, we use the Temporally Evolving Graph (TEG).

In a $TEG = (V, E, w, t_s, t_e)$, besides the nodes, edges and weights, we add two temporal attributes t_s and t_e to restrict the nodes and edges. Each node is represented in a triplet as <n, t_s, t_e> which implies that node n appears in the graph snapshots during the time interval [t_s, t_e). When a node is first created, its t_e is initialized with the special symbol "***now***", noting a currently existing ('alive') node in the current snapshot of the graph. Each edge in a *TEG* is represented as <n_1, n_2, w, t_s, t_e> noting that this edge runs from node n_1 to node n_2 with weight w during the time interval [t_s, t_e). Figure 2(a) shows the *TEG* of the graph sequence in Figure 1. Its nodes and edges are listed in Figure 2 (b) and (c). Given that most graphs do not change drastically over time, adding a temporal interval for each node and edge ever created allows the integrated temporal evolving graph to save storage space significantly.

(a) TEG



(b) Nodes

<a, t_1, now>
<b, t_1, now>
<c, t_1, now>
<d, t_1, now>
<e, t_1, now>
<f, t_1, now>
<g, t_3, now>

(c) Edges

<a, b, 2, t_1, now>
<a, c, 3, t_1, now>
<b, d, 5, t_1, now>
<c, d, 3, t_1, now>
<d, e, 4, t_1, now>
<e, f, 6, t_1, t_5>
<a, d, 7 t_2, t_4>
<d, f, 8, t_2, t_5>
<d, g, 5, t_3, now>
<g, f, 2, t_3, now>
<b, g, 8, t_4, now>
<c, e, 6, t_4, now>
<e, f, 4, t_5, now>

Figure 2: The TEG example and its nodes and edges

Note that in a TEG there may exist parallel edges connecting two nodes (such edges, however, have non-intersecting time intervals). For example, in Figure 2(a), between nodes *e* and nodes *f*, there are two separate edges <*e*, *f*, 6, t_1, t_4> and <*e*, *f*, 4, t_5, *now*>. However, between the same pair of nodes, there is only one unique valid edge at any given time point.

2.2.2 Temporal Query Definitions

In addition to the given source node n_s and target node n_t, a temporal shortest-path query requires a time constraint, such as a time-point t_q, or a time-interval $[ts_q, te_q)$, which restricts the candidate nodes and edges within a specific part of the whole temporal graph *TEG*.

Definition 1. The sub-graph of a temporal evolving graph $TEG = (V, E, w, t_s, t_e)$ for a time-point constraint t_q is defined as ***sub-TEG***$(TEG, t_q) = (sub\text{-}V, sub\text{-}E, w, t_s, t_e)$, where $\forall v \in sub\text{-}V : (v \in V \land t_s(v) \le t_q \land t_e(v) > t_q)$ and $\forall e \in sub\text{-}E : (e \in E \land t_s(e) \le t_q \land t_e(e) > t_q)$, representing the graph snapshot at time t_q. Similarly, the sub-graph of a temporal evolving graph *TEG* for a time interval constraint $[ts_q, te_q)$ is defined as ***sub-TEG***$(TEG, ts_q, te_q) = (sub\text{-}V, sub\text{-}E, w, t_s, t_e)$, where $\forall v \in sub\text{-}V : (v \in V \wedge t_s(v) < te_q \wedge t_e(v) > ts_q)$ and $\forall e \in sub\text{-}E : (e \in E \wedge t_s(e) < te_q \wedge t_e(e) > ts_q)$, representing the graph snapshots during time interval $[ts_q, te_q)$.

Definition 2. A **T**ime **P**oint **S**hortest **P**ath query **TPSP**(TEG, n_s, n_t, t_q) returns the distance of a path $p(e_1,\ldots,e_k)$ for query time t_q, which is the shortest-path from a source node n_s to a target node n_t, and all edges in p are valid at query time t_q. In another words, path p satisfies: $n_1(e_1) = n_s \wedge n_2(e_k) = n_t \wedge \forall e_i \in p : (t_s(e_i) \leq t_q \wedge t_e(e_i) > t_q)$; and $\forall p' \subseteq$ sub-TEG(TEG, qt) from n_s to n_t : dist$(p') \geq$ dist(p).

For any time point shortest-path query, since the corresponding historical graph snapshot is unique, there is a single distance returned. However, for the time interval query, the distance from source to target may change within this time interval. For example, during time interval $[t_2, t_5)$, the shortest-path from node *a* to node *f* has three different distances, namely: $\{14, (a, c, d, f), [t_2, t_3)\}$, $\{13, (a, c, d, g, f), [t_3, t_4)\}$, and $\{12, (a, b, g, f), [t_4, t_5)\}$. We thus define two different time interval queries for shortest paths in a TEG: the first variation returns all the shortest distances during the time interval, while the second query variation returns an aggregated result over these distances.

Definition 3. A **T**ime **I**nterval **S**hortest **P**ath "**all**" query **TISP-all**$(TEG, n_s, n_t, ts_q, te_q)$ returns a set of distances for paths $P = \{p_1,\ldots,p_m\}$ which contains all the shortest distance paths from source node n_s to target node n_t during the query time interval $[ts_q, te_q)$. Each path $p_i \in$ P is associated with a time interval [ts_{pi}, te_{pi}) and there is no other path shorter than p_i from n_s to n_t during this time interval [ts_{pi}, te_{pi}).

For the aggregated time interval query, we could utilize the *minimum*, *maximum*, or *average*; without loss generality, here we outline the minimum as a representative.

Definition 4. A time interval shortest path "**min**" query **TISP-min**(*TEG*, n_s, n_t, ts_q, te_q) returns the path *p* which is the minimum shortest path from source node n_s to target node n_t during the query time interval [ts_q, te_q).

Based on definitions 3 and 4, it is clear that: TISP-min(*TEG*, n_s, n_t, ts_q, te_q) = *min*(TISP-all(*TEG*, n_s, n_t, ts_q, te_q)).

2.3 Fundamental Solution

Dijkstra's algorithm [19] is the classic solution for the point-to-point shortest path query. Here, we discuss how to process a temporal shortest path query by extending the traditional Dijkstra's algorithm.

2.3.1 Dijkstra's Algorithm for Time Point SP Queries

For the time point shortest path (TPSP) query on a temporal evolving graph, there is a straightforward adaption of Dijkstra's algorithm using a priority queue *PQ*, as presented in Algorithm 1. In particular, we need to verify if an edge *e*'s time interval [t_s(e), t_e(e)) is valid at time t_q (i.e., $t_s(e) \leq t_q < t_e(e)$) before relaxing this edge in the search (Algorithm 2.1, line 13).

Algorithm 2.1: TPSP-Dijkstra(TEG, n_s, n_t, t_q)

Input: Temporal evolving graph $TEG = (V, E, w, t_s, t_e)$,

n_s, $n_t \in sub\text{-}V(t_q)$, and query time t_q

Output: Distance of the shortest path $p \subseteq$ sub-TEG(TEG, t_q)

```
1:  $PQ \leftarrow \emptyset$
2:  for all $v \in V \wedge t_s(v) \le t_q \wedge t_q < t_e(v)$ do
3:     $d[v] \leftarrow \infty$
4:  end for
5:  $d[n_s] \leftarrow 0$
6:  $PQ.Insert(n_s, d[n_s])$
7:  while !$PQ.empty()$ do
8:     $u \leftarrow PQ.ExtractMin()$
9:     if $u = n_t$ then
10:       return $d[n_t]$
11:    end if
12:    for all $e = (u, v) \in E$ do
13:       if $t_s(e) \le t_q \wedge t_q < t_e(e) \wedge d[u] + w(e) < d[v]$ then
14:          $d[v] \leftarrow d[u] + w(e)$
15:          if $v \notin PQ$ then
16:             $PQ.Insert(v, d[v])$
17:          else
18:             $PQ.DecreaseKey(v, d[v])$
19:          end if
20:       end if
21:    end for
22: end while
23: return $\infty$
```

An optimization we used here is to store the adjacent edges of a given node sorted first by their target and then by their start time. This is helpful in pruning temporally invalid edges; when a target node or an edge is accessed whose start time is later than the query time-point, any remaining edges can be skipped. This edge pruning optimization can be utilized for time-interval queries as well.

2.3.2 Dijkstra's Algorithm on Time Interval SP Queries

For the time interval shortest path "all" (TISP-all) query, the naïve method is to perform the TPSP-Dijkstra for all time points within the query interval [*qts*, *qte*). Therefore, for a query time interval with *k* time instants, this approach would run TPSP-

Dijkstra *k* times, which will not be as efficient. An improved approach is to run Dijkstra's algorithm once and return all the qualified answers for the TISP-all query.

Algorithm 2.2: TISP-all-Dijkstra($TEG, n_s, n_t, ts_q, te_q$)

Input: Temporal evolving graph $TEG = (V, E, w, t_s, t_e)$,
$sou, tar \in sub\text{-}V(ts_q, te_q)$, and time interval $[ts_q, te_q)$
Output: All distances of the shortest path set $P \subseteq$ sub-TEG(TEG, ts_q, te_q)

```
1:  $[ts_{qin}, te_{qin}) \leftarrow [t_s(n_s), t_e(n_s)) \cap [t_s(n_t), t_e(n_t)) \cap [ts_q, te_q)$
2:  if $\emptyset = [ts_{qin}, te_{qin})$ then return $\infty$
3:  $d_{out} \leftarrow \infty$ with time interval(s) $[ts_q, te_q)$ - $[ts_{qin}, te_{qin})$
4:  $PQ \leftarrow \emptyset$; $done \leftarrow \emptyset$
5:  for all $v \in V \wedge t_s(v) < te_{qin} \wedge t_e(v) \geq ts_{qin}$ do
6:      $D[v] \leftarrow \{d[v, ts_{qin}, te_{qin}] \leftarrow \infty\}$
7:  end for
8:  $D[n_s] \leftarrow \{d[n_s, ts_{qin}, te_{qin}] \leftarrow 0\}$
9:  $PQ.Insert(<n_s, ts_{qin}, te_{qin}>, d[n_s, ts_{qin}, te_{qin}])$
10: while !$PQ.empty()$ do
11:     $<u, ts_{ui}, te_{ui}> \leftarrow PQ.ExtractMin()$
12:     if $u = n_t$ then
13:         $Done \leftarrow Done + [ts_{ui}, te_{ui})$
14:         if $Done = [ts_{qin}, te_{qin})$ then
15:             return $D[n_t] \cup d_{out}$
16:         end if
17:     end if
18:     for all $e = (u, v) \in E$ do
19:         if $t_s(e) < te_{ui} \wedge t_e(e) > ts_{ui}$ then        // $[t_s(e), t_e(e))$ overlaps with $[ts_{ui}, te_{ui})$
20:             $[t_l, t_r) \leftarrow [t_s(e), t_e(e)) \cap [ts_{ui}, te_{ui})$
21:             for all $d[v, ts_{vj}, te_{vj}] \in D[v]$ and $ts_{vj} < t_r \wedge te_{vj} > t_l$ do
22:                 if $d[u, ts_{ui}, te_{ui}] + w(e) < d[v, ts_{vj}, te_{vj}]$ then
23:                     Updating
24:                 end if
25:             end for
26:             for all $d_j[v, ts_{vj}, te_{vj}] \in D[v]$, ordered by $ts_{vj}$ do
27:                 if $d_j = d_{j-1}$ then
28:                     merge $d_{j-1}$'s time interval into $d_j$; remove $d_{j-1}$
29:                 end if
30:             end for
31:         end if
32:     end for
33: end while
34: return $D[tar] \cup d_{out}$
```

The TISP-all-Dijkstra's algorithm, as presented in Algorithm 2.2, is different than the traditional TPSP in three aspects. First, the algorithm cannot stop until the confirmed shortest path distance covers the whole query interval. As a result, we need to record the parts that are done as well as undone. Second, the distance from the source to a given node *v* within the query interval is not a single value *d*, but a set of values *D* with different time aspects (due to parallel edges). Last, updating the distance set *D* and priority queue *PQ* is more complex. We present the details in Algorithm 2.3.

Algorithm 2.3: Updating

This is the updating function in Algorithm 2.2, line 23. The new distance of node *v* with time interval $[t_l, t_r)$ is $d_{new} = d[u, ts_{ui}, te_{ui}] + w(e)$. The overlapped previous distance with a larger value holds time interval $[ts_{vj}, te_{vj})$.

```
 1: if $t_l \le ts_{vj} \wedge t_r \ge te_{vj}$ then
 2:     $d[v, ts_{vj}, te_{vj}] \leftarrow d_{new}$
 3:     $PQ.DecreaseKey(<v, ts_{vj}, te_{vj}>, d[v, ts_{vj}, te_{vj}])$
 4: else if $t_l \le ts_{vj} \wedge t_r < te_{vj}$ then
 5:     $D[v] = D[v] + \{d[v, ts_{vj}, t_r] \leftarrow d_{new}\}$
 6:     $PQ.Insert(<v, ts_{vj}, t_r>, d[v, ts_{vj}, t_r])$
 7:     $D[v] = D[v] + \{d[v, t_r, te_{vj}] \leftarrow d[v, ts_{vj}, te_{vj}]\}$
 8:     $PQ.Insert(<v, t_r, te_{vj}>, d[v, t_r, te_{vj}])$
 9:     $D[v] = D[v] - \{d[v, ts_{vj}, te_{vj}]\}$
10:     $PQ.Delete(<v, ts_{vj}, te_{vj}>, d[v, ts_{vj}, te_{vj}])$
11: else if $t_l > ts_{vj} \wedge t_r \ge te_{vj}$ then
12:     $D[v] = D[v] + \{d[v, t_l, te_{vj}] \leftarrow d_{new}\}$
13:     $PQ.Insert(<v, t_l, te_{vj}>, d[v, t_l, te_{vj}])$
14:     $D[v] = D[v] + \{d[v, ts_{vj}, t_l] \leftarrow d[v, ts_{vj}, te_{vj}]\}$
15:     $PQ.Insert(<v, ts_{vj}, t_l>, d[v, ts_{vj}, t_l])$
16:     $D[v] = D[v] - \{d[v, ts_{vj}, te_{vj}]\}$
17:     $PQ.Delete(<v, ts_{vj}, te_{vj}>, d[v, ts_{vj}, te_{vj}])$
18: else if $t_l > ts_{vj} \wedge t_r < te_{vj}$ then
19:     $D[v] = D[v] + \{d[v, lt, rt] \leftarrow d_{new}\}$
20:     $PQ.Insert(<v, lt, rt>, d[v, lt, rt])$
21:     $D[v] = D[v] + \{d[v, ts_{vj}, t_l] \leftarrow d[v, ts_{vj}, te_{vj}]\}$
22:     $PQ.Insert(<v, ts_{vj}, t_l>, d[v, ts_{vj}, t_l])$
23:     $D[v] = D[v] + \{d[v, t_r, te_{vj}] \leftarrow d[v, ts_{vj}, te_{vj}]\}$
24:     $PQ.Insert(<v, t_r, te_{vj}>, d[v, t_r, te_{vj}])$
25:     $D[v] = D[v] - \{d[v, ts_{vj}, te_{vj}]\}$
26:     $PQ.Delete(<v, ts_{vj}, te_{vj}>, d[v, ts_{vj}, te_{vj}])$
27: end if
```

At the beginning, we consider the "inter" query time interval $[ts_{qin}, te_{qin})$, which is the intersection of the time intervals of source node n_s, target node n_t, and query time interval $[ts_q, te_q)$. Outside of this "inter" query time interval, there is no valid path from n_s to n_t

within the original query time interval. Hence, we focus on the “inter” interval to compute the shortest-path distances, which can reduce the search space. When we extract a node u from the priority queue, and u is the target node n_t, we check if the whole “inter” query interval is done (Algorithm 2.2, line 12). If not, the search for the “all” shortest-path distances should be continued.

As shown in Algorithm 2.2 (lines 18-32), when relaxing a valid edge $e(u, v)$ from the node u (whose time-interval is $[ts_{ui}, te_{ui})$), it may contribute a new distance(s) value for node v within an intersected time-interval $[t_l, t_r)$. For any temporally overlapped distance of node v, if the new distance $d_{new} = d[u, ts_{ui}, te_{ui}] + w(e)$ is smaller than the previous value (whose time-interval is $[ts_{vj}, te_{vj})$), we need to update the distance set of v and the priority queue.

As demonstrated in Figure 3 and Algorithm 3, there are three different cases to be considered. First, if the new distance’s time interval covers the whole old distance’s time interval (Figure 3(a)), we replace the old value with the new and decrease its key in the priority queue (Alg. 3, lines 1-3). Second, if the new distance’s time interval covers the head (or tail) of the old distance’s time interval (Figure 3(b, c)), we split the old interval into two parts. The covered one is updated with the new value while the uncovered one retains the old value (Alg. 3, lines 4-17). Third, if the new distance’s time interval is totally inside of the old one’s (Figure 3(d)), the old interval is split into three parts. The middle part is updated with the new value while the head and tail retain the old value (Alg. 3, lines 18-26). Each time, we only update one temporally overlapped distance value $d[v, ts_{vj}, te_{vj}]$. After all distances are updated, we run a post-process to merge the adjacent identical distances for node v.

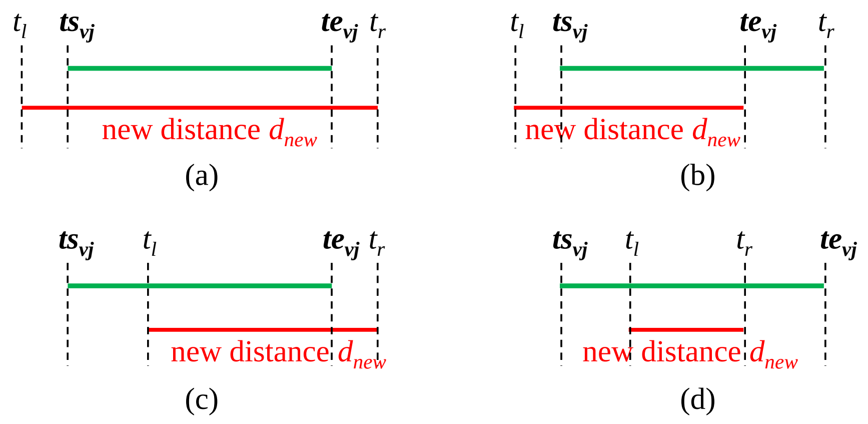


Figure 3: Different scenarios for updating the previous distance and priority queue

For the TISP-min query, the evaluation process is similar to the TISP-all-Dijkstra's algorithm. The only difference is that the algorithm can be stopped once the first shortest path is settled for the target node, without exploring all the candidates. The first discovered shortest path is guaranteed to be the minimum due to the Dijkstra's algorithm.

2.4 Speed-up Techniques

Here, we propose some speed-up techniques for the temporal shortest path algorithms. Besides the commonly used bidirectional search approach, we analyze the utilization of preprocessing indexes, such as hierarchical methods and goal-directed search algorithms.

2.4.1 Bidirectional Search

The bidirectional search method [42] utilizes the Dijkstra's algorithm for both forward and backward searches, and proceeds in two phases. In the first phase, we alternate between two unidirectional searches: one forward search from source *s* growing a spanning tree *S*, and the other backward search from target *t* growing a spanning tree *T*. When the forward and backward searches reach the same vertex v_0, we move on to the second phase, in which the shortest path is found.

For the TPSP query, the bidirectional Dijkstra's algorithm can be straightforwardly extended from the unidirectional TPSP-Dijkstra. However, the case of the TISP-all query requires attention. In the unidirectional TISP-all-Dijkstra algorithm, we start by finding the "inter" interval $[ts_{qin}, te_{qin})$ which is the intersection of source and target node time intervals with the query interval $[ts_q, te_q)$. In phase 1, we can adopt the TISP-all-Dijkstra for both the forward and backward search alternately. When we meet a vertex labeled in both *S* and *T*, we can move on to phase 2. A shortest path distance for the intersection time interval can be found if the intersection is not empty. Then we go back to phase 1 and continue the bidirectional search until the whole "inter" interval $[ts_{qin}, te_{qin})$ is done in phase 2.

2.4.2 Contraction Hierarchies

Hierarchical methods (such as HH [48], TNR [7], and CH [22]) seek to order the nodes and/or edges within the graph to hierarchically nested levels, based on some measure of overall graph structure. One of the most efficient methods to date is the contraction hierarchies (CH [22, 23]). The effectiveness of the CH search technique comes from the use of the newly-added shortcut edges, which allow Dijkstra's search to effectively bypass irrelevant nodes during the search, without invalidating correctness.

Node Contracting. Certain absolute ordering ϕ of the vertices is established in the graph, according to some notion of relative importance; then the CH is constructed by "contracting" one vertex at a time in increasing order. When a vertex v is contracted, it is removed from the current graph. For any pair of remaining vertices, u and w, adjacent to v in the original graph whose only shortest u-w path is $<u, v, w>$, a so-called ***shortcut*** edge (u, w) must be added with the weight of the original shortest path cost through v. A *local witness search* for v (from and to all its neighbors) is required to determine the shortcuts.

Querying. Once all necessary shortcuts E' are added to the graph G for a given ordering, shortest path queries may then be carried out using a bidirectional Dijkstra search variant which performs a simultaneous forward search in the upward graph $G_{\uparrow} = (V, E_{\uparrow})$, where $E_{\uparrow} = \{(v, w) \in E \cup E' \mid \phi(v) < \phi(w)\}$, and backward search in the downward graph $G_{\downarrow} = (V, E_{\downarrow})$, where $E_{\downarrow} = \{(u, v) \in E \cup E' \mid \phi(u) > \phi(v)\}$. A tentative shortest path cost is maintained and is updated only when the two search frontiers meet to form a shorter path. Once the minimum key from the priority queue exceeds the distance of the best path for both directions, the search is finished.

Node Ordering. Note that a good node ordering is one of the most crucial aspects of CH. The computation of an optimal node ordering (i.e. shortcut minimal or query search space minimal) is NP-hard. The heuristic solution here is to consider several different ordering metrics, along with several different combinations of weighted coefficients for each metric tested. Work [23] establishes several metrics including edge difference, contracted neighbors, original edges, and so on.

Incorporating CH into TEG

Here, we analyze how to incorporate the contraction hierarchies into our temporal evolving graph. Since the CH indexing adds shortcuts on the original graph, in a form of extra edges, we can extend the contraction hierarchies by adding temporal information on the shortcut edges as well. Now we discuss how to construct a global CH for the whole TEG based on a node ordering within the whole graph lifetime.

When contracting a vertex v, one way we can do it is to perform a local witness search for each pair of neighbors, which is a total of $|I_v^{\downarrow}|*|O_v^{\uparrow}|$ separate local searches (where $I_v^{\downarrow} = \{(u, v) \in E : \phi(u) > \phi(v)\}$ and $O_v^{\uparrow} = \{(v, w) \in E : \phi(v) < \phi(w)\}$). In practice, a better way is to perform a single forward shortest-path search from the source node u of each incoming edge $e_{\downarrow} = (u, v) \in I_v^{\downarrow}$, ignoring node v until all nodes in the set $W = \{w \in V \mid (v, w) \in O_v^{\uparrow}\}$ have been settled. When a target node w is "settled", it means its distances are settled for the whole "local search time-interval" $[ts_{e\downarrow}, te_{e\downarrow})$ of the incoming edge $e_{\downarrow}$. We can also stop the search from u when it has reached a distance of $w(e_{\downarrow}) + max\{w(e_{\uparrow}) \mid e_{\uparrow} \in O_v^{\uparrow}\}$.

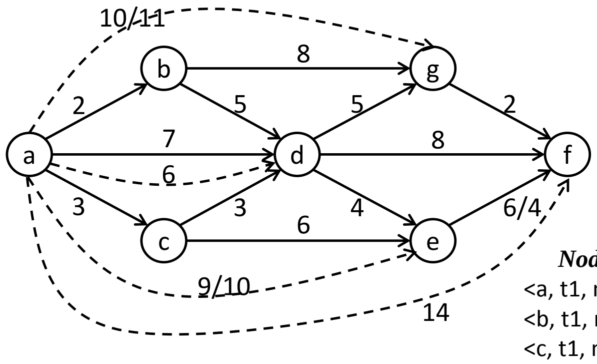
This task can be achieved efficiently with the help of our TISP-all-Dijkstra query processing algorithm without a specified target, and all the "witness" shortest-path distances are stored in set D. Then we compare them against the distance of path $u \rightarrow v \rightarrow w$ as $dist = w(e_{\downarrow}) + w(e_{\uparrow})$ within time-interval $[ts_{e\downarrow}, te_{e\downarrow})$. If $dist$ is smaller than some value in D of a time interval $[ts_i, te_i)$, then we need to add a shortcut (u, w) with a weight $dist$ and a time interval $[ts_i, te_i)$. More details about node contracting in CH on TEG are shown in Algorithm 2.4.

Algorithm 2.4: TEG-CH-Contraction(TEG, ϕ)

Input: Temporal evolving graph $TEG = (V, E, w, t_s, t_e)$,
and node ordering function ϕ: $V \rightarrow \{1, \ldots, |V|\}$
Output: Augmented temporal evolving graph $TEG' = (V, E \cup E', w, t_s, t_e)$, where E' represents newly added shortcut edges

```
1:  $TEG' \leftarrow TEG$ ; $E' \leftarrow \emptyset$
2:  for all $v \in V$ ordered by $\phi$ do
3:    for all $e_{\downarrow} = (u, v) \in E \cup E'$ : $\phi(u) > \phi(v)$ do
4:      $maxOutDist \leftarrow 0$; $W \leftarrow \emptyset$
5:      for all $e_{\uparrow} = (v, w) \in E \cup E'$ : $\phi(v) < \phi(w)$ do
6:        if $\emptyset \neq [ts_{e\downarrow}, te_{e\downarrow}) \cap [ts_{e\uparrow}, te_{e\uparrow})$ then
7:          $W \leftarrow W \cup \{w\}$
8:          $maxOutDist \leftarrow max(w(e_{\uparrow}), maxOutDist)$
9:        end if
10:     end for
11:     $TEG'_{v} \leftarrow TEG'[\{z \in V \mid \phi(v) < \phi(z)\}]$
12:     do local search $D \leftarrow$ TISP-all-Dijkstra($TEG'_{v}$, $u$, $\emptyset$, $ts_{e\downarrow}$, $te_{e\downarrow}$) until
        $W$ fully settled for $[ts_{e\downarrow}, te_{e\downarrow})$ or distance $w(e_{\downarrow})+maxOutDist$ reached
13:     for all $d(w_i) \in D$ with time-interval $[ts_{wi}, te_{wi})$ do
14:       for all $e_{\uparrow} = (v, w_i)$ overlaps with time-interval $[ts_{wi}, te_{wi})$ do
15:         $dist \leftarrow w(e_{\downarrow}) + w(e_{\uparrow})$
16:         if $dist < d(w_i)$ then
17:           $e' \leftarrow (u, w_i)$ ; $w(e') \leftarrow dist$
18:           $[ts_{e'}, te_{e'}) \leftarrow [ts_{e\downarrow}, te_{e\downarrow}) \cap [ts_{e\uparrow}, te_{e\uparrow}) \cap [ts_{wi}, te_{wi})$
19:           $E' \leftarrow E' \cup \{e'\}$
20:           $TEG' \leftarrow TEG' \cup E'$
21:         end if
22:       end for
23:     end for
24:   end for
25: end for
26: return $TEG'$
```

The CH on TEG of our running example in Figure 1 is shown in Figure 4, based on an example importance ordering as b < c < d < a < e < f < g. Since the shortcuts are also certain types of "edges" in the pre-processed graph, the shortcuts of CH on TEG have two new features inherited from the properties of TEG's edges: i) each shortcut has a time interval validity [t_s, t_e], and ii) parallel shortcuts are supported as well. There are two pairs of parallel shortcuts in our TEG-CH example: <a, e, 10, t1, t3, d> / <a, e, 10, t1, t3, d> and <a, g, 11, t3, t3, d> / <a, g, 10, t4, now, b>.



Importance ordering:
b < c < d < a < e < f < g

Nodes
<a, t1, now>
<b, t1, now>
<c, t1, now>
<d, t1, now>
<e, t1, now>
<f, t1, now>
<g, t3, now>

Edges
<a, b, 2, t1, now>
<a, c, 3, t1, now>
<b, d, 5, t1, now>
<c, d, 3, t1, now>
<d, e, 4, t1, now>
<e, f, 6, t1, t5>
<a, d, 7 t2, t4>
<d, f, 8, t2, t5>
<d, g, 5, t3, now>
<g, f, 2, t3, now>
<b, g, 8, t4, now>
<c, e, 6, t4, now>
<e, f, 4, t5, now>

Shortcuts
<a, d, 6, t1, now>
<a, e, 10, t1, t4>
<a, f, 14, t2, t3>
<a, g, 11, t3, t4>
<a, e, 9, t4, now>
<a, g, 10, t4, now>

Figure 4: Example of contraction hierarchies on a temporally evolving graph

Once the construction phase of CH on TEG is finished, the shortest path queries can be carried out. The algorithm employed on the corresponding CH for TEG is similar to the bidirectional Dijskra's algorithm on CH for traditional shortest path queries. For each upward and downward search, our proposed TPSP-Dijkstra and TISP-Dijkstra algorithms can be utilized. For example, consider the time-interval "all" query of $[t_2, t_5)$ from *a* to *f*. The upward search from *a* extracts the following distances in order: $<e, 9, t_4, t_5>$, $<e, 10, t_2, t_4>$, $<g, 10, t_4, t_5>$, $<g, 11, t_3, t_4>$, and $<f, 14, t_2, t_3>$, while the downward search from *f* only extracts the distance $<g, 2, t_3, t_5>$. So the "all" shortest path distances from *a* to *f* with $[t_2, t_5)$ are: $<14, t_2, t_3>$ $(a \rightarrow f)$, $<13, t_3, t_4>$ $(a \rightarrow g \rightarrow f)$, and $<12, t_4, t_5>$ $(a \rightarrow g \rightarrow f)$.

2.4.3 Landmark-based A* Search

Goal-directed search techniques (such as arc-flags [11] and ALT [5]) try to "direct" the shortest-path search toward some explicit target node (i.e., the "goal"), in order to speed up the overall query time. One of the most effective goal-directed search techniques is the ALT algorithm [5], using A* search in combination with Landmarks and the Triangle inequality.

The ALT algorithm is primarily based on A* search [12], which works like Dijkstra's algorithm except that at each step it selects a labeled vertex v to scan, with the smallest value of $k(v) = d_s(v) + \pi_t(v)$, where the potential function $\pi_t(v)$ gives an estimate on the distance from v to the search target t. For bidirectional A* search, we assume π_t and π_s give lower bounds to the target and from the source, respectively. As suggested, we use an average potential function defined as $p_t(v) = (\pi_t(v) - \pi_s(v)) / 2$ for the forward computation and $p_t(v) = (\pi_s(v) - \pi_t(v)) / 2 = - p_t(v)$ for the reverse one. They are feasible and consistent for bidirectional A* search.

ALT involves preprocessing, which selects a small set of vertex as landmarks L, and for each vertex in the graph, pre-computes the shortest-path distance to and from every landmark. For any node v, with target node t, the triangle inequality provides two lower bounds for each landmark, $l \in L$: $d(l, t) - d(l, v) \leq dist(v, t)$ and $d(v, l) - d(t, l) \leq dist(v, t)$. The maximum of these lower bounds over all landmarks is used to get the tightest lower bound. The original implementation of ALT uses, for each shortest path querying, only a subset of active landmarks, those that give the best lower bounds on the s-t distance.

Incorporating ALT into TEG

The ALT method also can be extended for the temporal shortest path problems in TEG by solving the two key steps in preprocessing phase: **landmark selection** and **distance computation**.

Finding good landmarks is critical for the overall performance of lower-bounding algorithms. Selecting the optimal set of landmarks is an NP-hard problem [44]; however, several strategies are described in [24]. The simplest way to select landmarks is at *random*. The *farthest* greedy algorithm works as follows. Pick a start vertex at random and find a vertex v that is farthest from it. Add v to the set of landmarks. Proceed in iterations, always adding to the set the vertex that is farthest from it.

The main problem of previous landmark selection strategies all focus on a static graph and measure the distance based on the graph structure. If we just simply utilize them, the selected landmarks may not be suitable for the general temporal shortest path querying. For example, at a given query time, there may be few landmarks valid at all, which could downgrade the performance of ALT. Therefore, we should take consideration of the temporal information of the vertex. Without loss of generality, we adopt the farthest landmark selection in this work and extend it by choosing a set of global landmarks with a combination of “farthest” and “longest.” The “longest” term refers to the lifetime of the selected nodes.

In the distance computation step, for each vertex in the graph, we calculate the shortest-path distances to and from every landmark in the whole TEG graph lifetime. The computation can be efficiently achieved with the help of the proposed TISP-all-Dijkstra algorithm and its bidirectional version. Thus for each landmark, the distances are stored along with their time intervals. When computing the lower bounds in ALT search algorithm for time-point querying, we pick the unique temporally valid distance for the corresponding querying time point. When computing the lower bounds for time-interval queries, due to multiple temporally valid distances, we choose the minimum triangle inequality lower bound value to and from each landmark, and then use the maximum of these lower bounds over all landmarks as the tightest estimation.

2.5 Temporal Partition

2.5.1 Storage Graph Model

For historical evolving graphs, as we mentioned earlier, there are two data models. One is Graph Sequence (GS), storing all the graph snapshots for each time instance, and the other is our Temporal Evolving Graph (TEG) model with a super-graph containing all histories. The GS model is optimal for time-point querying, but it causes huge storage overhead and is not efficient for time-interval querying. On the other hand, the TEG model is optimal in space saving and efficient for time-interval querying (especially large intervals); however, its time-point querying performance is downgraded due to skipping plenty of temporally invalid edges. The trade-off solution is to make temporal partitions for one huge TEG along the time axis. For example, if the whole TEG has n time instances, and we create a partition for each m time instance, then we will get n/m partitions, as shown in Figure 5.

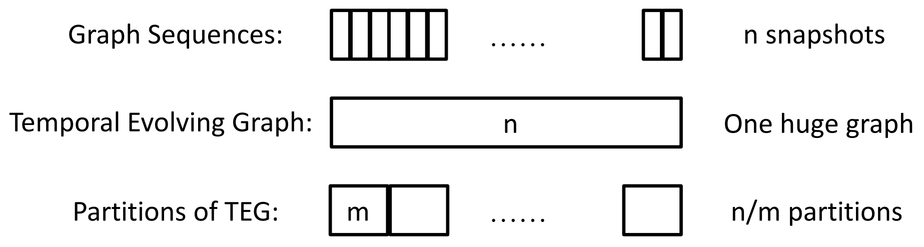


Figure 5: Example of a temporal partition for the storage graph model

The temporal partitioning indeed brings some duplicates between the partitions, but it reduces the size of queried TEG(s). The temporal shortest path queries (either a time-point or a time-interval) are issued on the corresponding TEG partition(s) instead of the original "super" TEG, which may speed up the query process. However, a long time-interval query may go across multiple partitions, which results in multiple runs of TISP-all querying.

For temporal partitioning, how to split is important. Here, we propose a simple and efficient split strategy called fixed-time-window (**fix**): each partition has a time-window with a fixed length. For example, as we presented in Figure 5, the fixed time-window length is *m*. The advantage of fix strategy is that: for a time-window length *m* any time-interval query with a length *l*, we need to access at least $\lceil l/m \rceil$ and at most $\lceil l/m \rceil$+1 partitions.

Another applicable split strategy is called graph-edit-distance (**ged**). This is borrowed from a clustering idea in [45]: similar snapshots are grouped together based on the symmetric difference of the graph's edge sets. During some time the graph may change more dramatically than other times, so the ged strategy may result in more balanced partitions from a storage point of view. However, ged's time-lengths of partitions is different compared to a fixed one in fix.

The basic temporal partition approach to split the whole graph into a set of disjointed adjacent partitions has an obvious drawback: even for a small time-interval query, if it goes over the borders of the partitions, we still need multi-partition accesses. For example, for a fix-10 temporal partition with time-window length as 10 (shown in Fig. 6), small time-interval query q_1 and q_2 both have a length of 5; since q_2 is over the border between partition p_2 and p_3 in the disjointed partition set (set-1), we need to access both p_2 and p_3 to get the correct "all" query answers. Therefore, we propose an **overlapped partition** solution: for partition p_i (i>1), it is overlapped with previous partition pi-1 with a factor f ($0 \leqslant f < 1$). For example, in Fig. 6, we make another set of "overlapped" partitions (set-2) with f = 50%. We can see that partition p_2 starts from time t_6 by overlapping half of p_1. Thus, the partition borders are covered by the overlaps. For a small time-interval like q_2 (across the partition borders in set-1), it fits in one partition p_4 of the set-2.

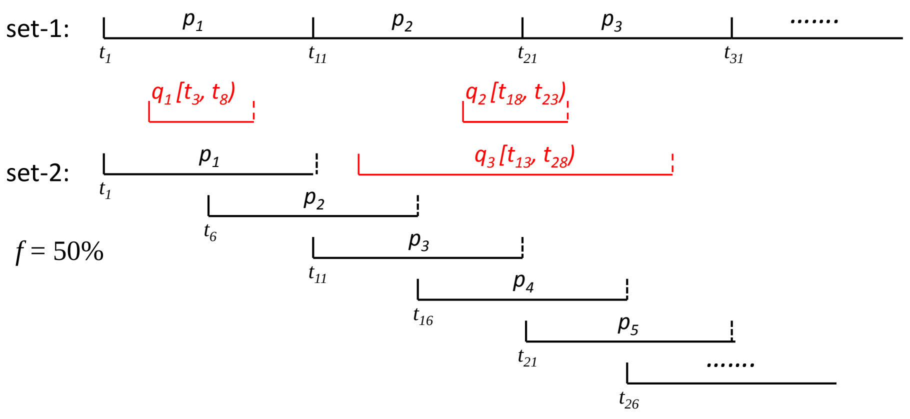


Figure 6: Example of an overlapped solution for a temporal partition

The overlapped partition solution increases the storage space by a factor of 1/(1-f); however, it improves the performance for time-interval queries by reducing the probability of multi-partition accessing for small time-interval queries. For example, assume a fix temporal partitioning with time-window length of m; we create the overlapped partition set by using f = 50% (overlap half). For any time-interval query length $l <= (m/2)+1$, we just need to access one partition, while in the original disjointed partitioning, we have the probability of $(l-1)/m$ to access two partitions. For time-interval query length $(m/2)+1 < l <= m$, we have the probability of $[(l-1)-(m/2)]/m$ to access two partitions, which is much smaller than the probability of $(l-1)/m$ in a disjointed partition set. Even for longer time-interval query whose $l > m$, we still have a larger probability to access fewer numbers of partitions in the overlapped partition solution than in disjointed partitioning. This is because the overlapped partition set with f = 50% is a superset of the disjointed adjacent partition set. For example, in Fig. 6, for a long time-interval query q_3 of 15-day length, it can be processed by accessing two partitions p_3 and p_5 (excluding partition p_4).

2.5.2 Indexing

The temporal partition idea can be used on preprocessing indexes as well. For both CH and ALT, their performance is highly related to certain key feature in their construction phase, like the node ordering for CH and the landmark selection for ALT. One global choice may not be the best for any single temporal query. Therefore we explore the opportunities to maintain different index structures for different time period partitions.

To implement temporal partition on indexing, there are two options: i) “partition both graph and index” by splitting the index along with the temporal evolving graph together; or ii) “partition only index” without actual graph-level splitting. For “partition both graph and index” option, we first partition the TEG based on certain splitting strategy. Then for CH (or ALT), we compute the node ordering and construct local CH (or select the landmarks and calculate the local distances) for each sub-TEG partition.

The "partition only index" option saves space for actual graph-level splitting; we need to access the original edges from the global TEG. For ALT, since the landmarks and their pre-computed distances are a set of separated structures from the original graph, this option seems favorable. For the whole graph lifetime, we split it into multiple sub-time-intervals, one for each virtual partition. Then for each sub-time-interval, we select its local landmark set and pre-compute the local distances from and to those landmarks. Multiple local landmark sets can achieve better querying performances than the single global landmark set. For CH, we can implement this option in a similar way. In addition to the original TEG-graph and newly-added temporal partitioned shortcut sets, we need to store extra information such as the partitioning sub-time-intervals along with the node ordering for each partition.

For temporal index partitioning of CH or ALT, in addition to fix and ged, other sophisticated split strategies can be explored, such as shortcut-edit-distance (sed) based on the symmetric differences among CH's shortcut sets or landmark-edit-distance (led) based on the symmetric differences of the selected landmarks.

2.6 Experimental Evaluations

All experiments have been done on an Intel® Core™ i5-2400S CPU at 2.50GHz with 8 GB RAM. Our implementation was written in C++ and compiled by gcc version 4.4.3.

2.6.1 Datasets

In our experiments, we used social network graphs from YouTube and Flickr, as provided by socialnetworks.mpi-sws.org. The properties of the real datasets are given in

Table 1. The storage space for different data models are listed in that table as well. In addition to Graph Sequence and Temporally Evolving Graph, we also present the storage space of the snapshot-delta model proposed in FVF work [45] as a comparison.

Table 1: Statistics of real datasets

Dataset	**YouTube**	**Flickr**
Graph type	Undirected	Directed
Number of snapshots	165	104
Date of last snapshot	2007-07-23	2007-05-08
|V| of first Snapshot	1,402,949	1,620,392
|V| of last Snapshot	3,218,658	2,570,535
Vertex growth	129%	58%
|E| of first snapshot	6,783,917	17,034,807
|E| of last snapshot	18,524,095	33,140,018
Edge growth	173%	63%
|E|/|V| of first snapshot	4.84	10.51
|E|/|V| of first snapshot	5.75	12.89
Size of TEG	451.2 MB	776.4 MB
Size of FVF	997.7 MB	1.6 GB
Size of GS	49.7 GB	59.7 GB

2.6.2 Setup of CH and ALT Indexing

For CH, node ordering is important. In this work, we consider three classic ordering metrics from [23]: *edge difference*, *contracted neighbors*, and *original edges*, and two novel priority terms: *lifetime length* and *new parallel-edge*. Lifetime represents the time interval length of a node; new parallel-edge represents the number of new parallel-edges introduced during the contraction of a node. For both the YouTube and Flickr dataset, we achieve fairly good performances by using edge difference and original edges with weight 2 and 1 respectively. Therefore, we use this setup in the following experiments.

To speed up the preprocessing phase of CH, especially for local witness searches, we use the hop-limit optimization: limit the depth of the shortest-path tree of the local search to 5. Note that this has no influence on the correctness of CH as long as we make sure to always insert a shortcut when we have not found a path witnessing that the shortcut is unnecessary. Meanwhile, we also use the *core nodes* optimization to reduce the preprocessing time. Node contracting is stopped when the number of remaining un-contracted nodes reaches a threshold, and the un-contracted nodes are left as core nodes. The size of core nodes we used here is 10k.

For ALT, landmark selection is crucial. In this work, we use our temporal optimized "farthest + longest" algorithm. Meanwhile, for a set of landmarks we use 32 nodes, and for each individual temporal query, we choose at most 6 active landmarks as a subset. A larger number of landmarks can gain better querying performance, but it also results in considerable storage overhead. For example, the storage space for 64 landmarks is about double the size of that for 32 landmarks.

2.6.3 Experimental Results

Time-Point Shortest-Path Query. For time-point queries, we get the average query performance time by running the shortest-path algorithms on every dataset day. And for each tested day, we choose 1000 uniformly random s-t pairs. The results are reported in Table 2. We can see that both CH and ALT index get more improvement in querying performance than bidirectional search. Meanwhile, CH is better than ALT with much smaller extra space usage by using more time in preprocessing. The YouTube and Flickr datasets have similar result patterns and we will show the results on the YouTube dataset for the following experiments.

Table 2: Preprocessing time, extra space, and performance of time-point querying

YouTube	TPSP-Dijkstra	Bidir	CH	ALT
Preprocessing	0	0	3h47m	1h12m
Extra Space (MB)	0	0	54.1	724.5
Query Time (ms)	2159	1283	340	384
Flickr	TPSP-Dijkstra	Bidir	CH	ALT
Preprocessing	0	0	4h19m	1h26m
Extra Space (MB)	0	0	97.2	817.5
Query Time (ms)	3647	1994	620	672

Table 3: Performance of time-interval querying

"all" query	Multi-TPSP	One-TISP	Bidir	CH	ALT
5-day	10.8s	6.3s	3.9s	1056ms	1163ms
15-day	32.4s	15.1s	9.4s	3039ms	3375ms
25-day	54.1s	25.9s	16.7s	4961ms	5428ms
"min" query	Multi-TPSP	One-TISP	Bidir	CH	ALT
5-day	10.8s	5.9s	3.4s	962ms	1039ms
15-day	32.4s	13.7s	7.8s	2665ms	2981ms
25-day	54.1s	23.4s	13.8s	4308ms	4846ms

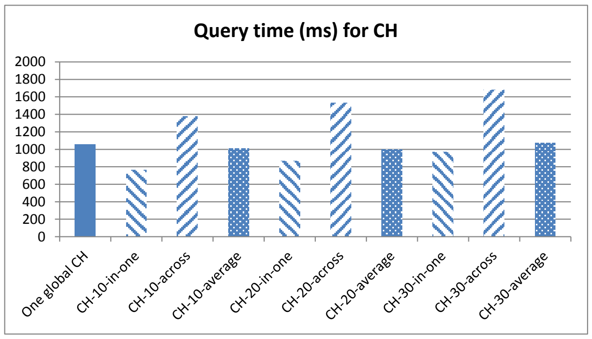
Time-Interval Shortest-Path Query. For time-interval querying, we tested on different query interval lengths of 5-day (3% of graph lifetime), 15-day (9% of graph lifetime), and 25-day (15% of graph lifetime). For each length, we randomly chose 100 time-intervals within the dataset lifetime. The querying performance time is also averaged by 500 uniformly random s-t pairs for each query time-interval. The results on the YouTube dataset are reported in Table 3. It can be seen that, for both time-interval "all" and time-interval "min" querying, the one-time run of TISP-Dijkstra's algorithm is much better than multiple runs of TPSP-Dijkstra's algorithm. And the average performance can be further improved by using CH or ALT index.

Temporal Partitioning. First, we demonstrate some results by utilizing the fixed-time-window (*fix*) split strategy. Different time-window lengths are tested as 10 days, 20 days, and 30 days. Therefore, for the YouTube dataset, the numbers of partitions are 17, 9, and 6, respectively. The last partition does not necessarily have a full fix-time length. Here, for both CH and ALT, we use the "partition only index" option. The preprocess time, extra space (refers to the space of index, while graph space is 451.2MB), and performance of time-point queries for different time-window lengths (CH-30 stands for CH index with 30-day time-window partition) are presented in Table 4. For temporal partitioning, the extra storage spaces have increased while time-point querying performances have improved. And smaller time-window length can bring in more time-point querying benefit, by requiring more space usage as well.

Table 4: Preprocessing time, extra space, and time-point querying for temporal partition

YouTube	Preprocessing	Extra space (index)	Query time
Bidir	0	0	1283 ms
CH	3h47m	54.1 MB	342 ms
CH-30	17h42m	234.7 MB	315 ms
CH-20	26h3m	362.4 MB	280 ms
CH-10	49h37m	687.0 MB	247 ms
ALT	1h12m	724.5 MB	384 ms
ALT-30	6h7m	6.8 GB	353 ms
ALT-20	9h11m	9.7 GB	311 ms
ALT-10	17h25m	18.6 GB	269 ms

We also test the temporal partition for time-interval "all" and "min" queries ("min" queries have similar results). Since one query time-interval may go across multiple partitions, we report the query performance by issuing multiple sub-queries (one for each overlapped partition) and merging the final results. The results of 5-day time-interval queries for CH temporal partitioning and ALT temporal partitioning are shown in Figure 7. We can see that, if the query time-interval is inside of a single partition, then the querying performance is better than using one global index. However, if the query time-interval goes across multiple partitions, the query performance is worse than on a global index solution due to multiple runs of all candidate partitions.



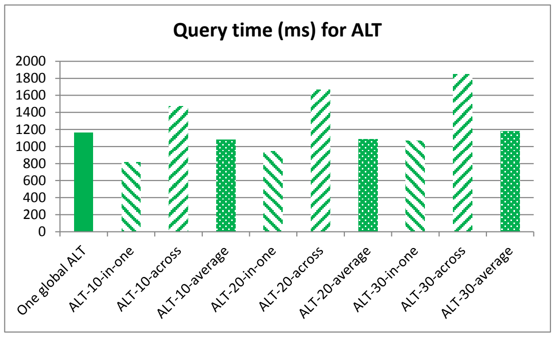


Figure 7: Performance of 5-day time-interval all queries for CH and ALT partitions

Since the 5-day query has a relatively small query interval length, its average performance is improved for all three different partition lengths. For each query, we need to visit at most two partitions. However, if the query interval is longer, it would have more chances to go across several partitions. The average results for 15-day queries (9% of the graph lifetime) are shown in Figure 8. We can see that, for 15-day all queries, the temporal partitioned indexes have worse performances on average than one global index without partitioning and small time-window length (10-day) partitioning, which is best for time-point querying but has the worst performance for both CH and ALT.

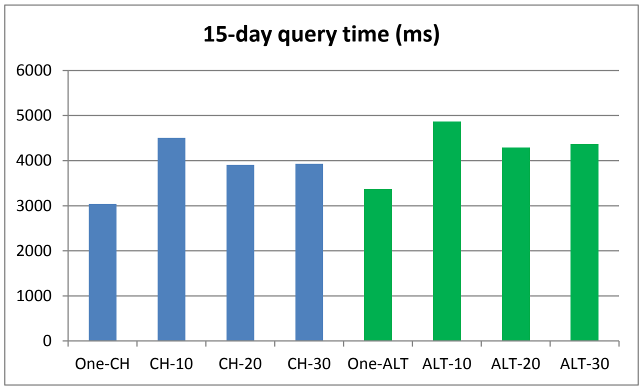


Figure 8: Average performance of 15-day temporal queries for CH and ALT partitions

Then, we compare the two different partition-level options: "partition only index (*oi*)" and "partition both graph and index (*gi*)" on fixed-time-window (*fix*) splitting for both CH and ALT speed-up techniques. Average results for time-point, time-interval "all", and time-interval "min" queries are reported in Table 5. We can see that partition option *gi* gains a little improvement in average querying performance rather than *oi* by using much more space to store the partitioned graph and index structures (especially for CH). Thus, for our YouTube dataset, the "partition only index" may be the desirable option.

Table 5: Comparing different partition-level options for CH and ALT

YouTube	CH	CH-20-oi	CH-20-gi	ALT	ALT-20-oi	ALT-20-gi
Preprocess	3h47m	26h3m	27h18m	1h12m	9h11m	10h26m
Total space	504.8MB	813.6MB	3.3GB	1.2GB	10.1GB	12.6 GB
TPSP	341ms	280ms	264ms	384ms	311ms	288ms
5-day-all	1056ms	999ms	928ms	1163ms	1087ms	1019ms
5-day-min	962ms	902ms	844ms	1039ms	984ms	933ms
15-day-all	3039ms	3905ms	3525ms	3375ms	4284ms	4007ms
15-day-min	2665ms	3280ms	2974ms	2981ms	3790ms	3525ms

Next, we compare different splitting strategies on the "partition only index" option. For fixed-time-window (*fix*), we use the 20-day length, resulting in 9 partitions, for both CH and ALT. The graph-edit-distance (*ged*) is also used for both CH and ALT by generating 9 partitions. Meanwhile, shortcut-edit-distance (*sed*) is used for CH and landmark-edit-distance (*led*) is used for ALT, both with 9 partitions. Since the time to compute each snapshot CH or ALT is too long, our *sed* or *led* splitting strategies compute the CH or ALT for every 5-day period; this results in 33 preprocessing computations. The average time-point and time-interval "all" querying results are shown in Table 6. Since our YouTube dataset has a smooth evolving pattern (without dramatic changes), the different splitting strategies have very similar performances. By considering the preprocessing time, fix split strategy may be the best choice.

Table 6: Comparing different split strategies for CH and ALT

YouTube	CH-fix	CH-ged	CH-sed	ALT-fix	ALT-ged	ALT-led
Preprocess	26h3m	27h37m	138h49m	9h11m	10h18m	46h20m
Total space	813.6MB	732.5MB	747.2MB	10.1GB	10.6GB	11.3GB
TPSP	280ms	264ms	272ms	311ms	295ms	302ms
5-day-all	999ms	937ms	952ms	1087ms	1004ms	1025ms
5-day-min	902ms	843ms	859ms	984ms	889ms	906ms
15-day-all	3905ms	4008ms	4061ms	4284ms	4332ms	4308ms
15-day-min	3280ms	3392ms	3424ms	3790ms	3872ms	3839ms

At last, we test the "overlapped partition" solution for fix-time-window splitting strategy. When we set overlap factor $f = 0$, it is the same as disjointed partitioning we used above. The results for "partition index only" option on CH indexing with time-window length as 10-day and 20-day, along with overlap factor f as 0, 30% and 50%, are shown in Table 6. The extra space here refers to the space for index storage (while the graph space is 451.2MB). We can see that when set f as 30%, it is only good for time-point and small time-interval queries; when we set *f* as 50%, we can get fairly good performances for both small time-interval and large time-interval queries.

Table 7: Overlapped temporal index partitioning for CH

	CH	10-0%	10-30%	10-50%	20-0%	20-30%	20-50%
# of partitions	0	17	24	33	9	12	17
Preprocess	3h47m	49h37m	73h29m	99h42m	26h3m	37h40m	51h18m
Extra space	54.1MB	687.0MB	892.2MB	1.2GB	362.4MB	530.4MB	774.8MB
TPSP	341ms	247ms	245ms	240ms	280ms	276ms	271ms
5-day-all	1056ms	1007ms	902ms	761ms	999ms	887ms	865ms
10-day-all	2042ms	2542ms	2932ms	1946ms	2285ms	1945ms	1674ms
15-day-all	3039ms	4504ms	5973ms	3960ms	3905ms	4510ms	3905ms

2.7 Conclusion

In this work, we studied the problem to answer temporal shortest-path distance queries on historical evolving graphs. Based on our newly proposed data model and query definitions, we extended the traditional Dijkstra's algorithm for both time-point and time-interval queries, in order to process the shortest-path querying efficiently. Moreover, we investigated how to incorporate preprocessing index structures such as CH and ALT to speed-up query processing. To analyze trade-offs and explore further optimizations, we proposed temporal partitioning, with multiple split strategies and partition-level options. To demonstrate our algorithms and optimizations, we performed experimental evaluations on real-world social-network datasets.

Chapter 3

Temporal Top-k Keyword Search in Social Tagging Websites Using Multiple Social Networks

The advent of Web 2.0 has facilitated the growth of online communities and applications such as blogs, wikis, online social networks and social tagging sites. In social tagging sites, users are provided easy ways to create social networks, to post and share items like bookmarks, videos, photos or articles, along with comments and tags. In this chapter, we present an experimental study of top-*k* search in social tagging sites by utilizing multiple social networks and temporal information of tagging behaviors. In particular, besides the global connection, we consider two main social networks, namely the friendship and common interest networks in our scoring functions. Based on the degree of participation in various networks, users can be categorized into specific classes that differ in their weights on each scoring component. Temporal information, usually ignored by previous works, can enhance the popularity and freshness of the ranking results. Experiments and evaluations on real social tagging datasets show that our framework works well in practice and give useful and intuitive results.

3.1 Introduction

With the advent and popularity of Web 2.0, the World Wide Web has become increasingly open for everyone. Successful Web 2.0 applications include blogs, wikis, online social networks, and social tagging sites. In social tagging sites, such as del.icio.us, Flickr and CiteULike (Table 8), user-generated data is the core feature. Once a user is logged in, he/she can easily edit his/her own personal profile, build social networks with friends, and contribute content by posting bookmarks, videos, photos, or articles. He/she can also annotate those items with arbitrary labels— the so-called tags. Social tagging sites are free, fun, and functional, attracting more and more people to register as users.

Table 8: Popular social tagging websites

	del.icio.us	flickr	citeulike
URL	www.delcious.com	www.flickr.com	www.citeulike.org
Type	Online social bookmarking	Photo sharing and Photo networking	Social bookmarking of academic articles
Owner	Yahoo! Inc	Yahoo! Inc	Oversity Ltd
Launched	Sep 2003	Feb 2004	Nov 2004
Statistics	Over 180 million bookmarked URLs	Over 6 billion images	Over 3 million articles bookmarked

With their increasing popularity, social tagging sites have formed and stored valuable information like user-generated items, user social networks, and user tags. How to make good use of this information to improve services such as hot-lists, recommendations and web search is an open and attractive challenge for both academia and industry.

In this work, we focus on temporal ranking and personal search in social tagging sites. When compared to other works, our contributions are: First, we apply multiple components to score an item with respect to a particular user's different social networks and assign weights to each component based on the classification of that user's participation in those networks. Then, we take into consideration the temporal information of tagging behaviors, in order to enhance popularity and freshness of the top-*k* results. Last, we provide a variation of the classic top-*k* algorithm which works efficiently for our user-dependent temporal scoring functions. Moreover, experimental evaluations on real social tagging datasets show that our framework works well in practice.

The rest of this chapter is organized as follows: in section 3.2 we review previous work on social tagging and web search. Section 3.3 describes the data model, user social networks and problem statement while section 3.4 demonstrates our user-based temporal scoring functions. The temporal top-k algorithm appears in section 3.5. Section 3.6 provides experimental results on real social tagging datasets while conclusion appears in section 3.7.

3.2 Related Work

Social tagging has become a hot research topic recently. Much work has investigated in related areas such as recommendation systems and web search.

Recommendation systems use information filtering (IF) techniques to present information items (movies, music, books, news, images, web pages, etc.) which are likely of interest to the users [36]. A highly-automated novel framework for real-time tag recommendation is proposed in [51]. [57] uses explanation-based diversity to explore compromises between accuracy and diversity in recommender systems.

An empirical analysis of how social bookmarking can influence web search is provided in [27], with both positive and negative insights. Various ranking methods have been developed and many of them are inspired by the well-known PageRank [13] method for web link analysis. They model the entities in social networks as a "social-content graph" and use a "random surfer" traversing the graph to compute the ranking of nodes to a user's query. [29] proposes FolkRank to identify important users, data items, and tags. [6] introduces SocialSimRank which calculates the similarity between social annotations and web queries, and SocialSimRank which captures the popularity of web pages.

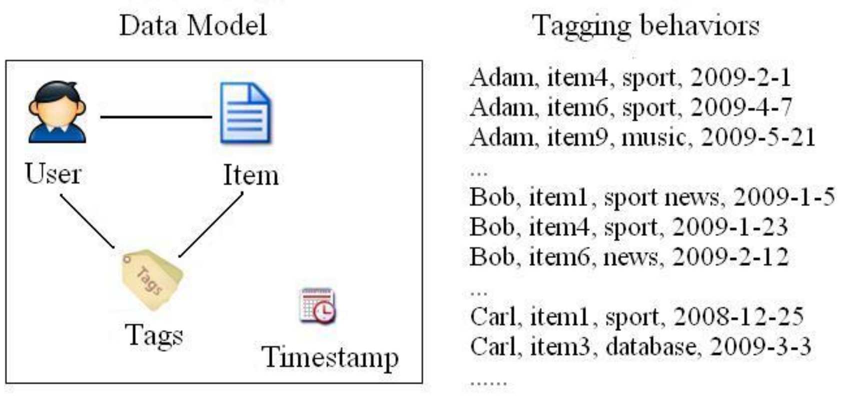
Recently, some studies expand traditional top-k algorithms [21] to do search in social tagging. An incremental top-k algorithm is developed in [50] with two expansions: the social expansion considers the strength of relations among users, and the semantic expansion considers the relations of different tags. A network-aware search is presented in [2] to incorporate social behavior into searching content in social tagging sites. It extends traditional top-k algorithms to bounds-based algorithms, and explores clustering users as a way to achieve a balance between processing time and space consumption. However, neither of them considers temporal information or the combining of multiple social networks.

3.3 Data Model

3.3.1 Tagging Behavior

Previous work in social tagging mostly ignores timestamps, and treats a tagging behavior as a three-factor tuple: <User, Item, Tags>, which indicates that a user u annotated one item i with arbitrary tags.

To take into account the temporal information, we extend the tagging behavior tuple by adding timestamps (Figure 9). In the following, we first demonstrate the model of social networks and static scoring functions without timestamps, and then explore a method to incorporate temporal information into ranking.



Adam, item4, sport, 2009-2-1
Adam, item6, sport, 2009-4-7
Adam, item9, music, 2009-5-21
...
Bob, item1, sport news, 2009-1-5
Bob, item4, sport, 2009-1-23
Bob, item6, news, 2009-2-12
...
Carl, item1, sport, 2008-12-25
Carl, item3, database, 2009-3-3
......

Figure 9: Four-factor data model for temporal tagging behaviors

3.3.2 Social Networks

In social tagging sites, users are generally participating in multiple social networks. Aside from the global connection, meaning that everyone can connect with anyone else on the whole web, here we consider two other kinds of social networks, namely, *friendship* and *common interest networks*.

Friendship is a kind of *explicit* social network. One user can choose to add any other users as friends. Most of them could be acquaintances in real life—friends, schoolmates, business contacts, etc; some may be known through the internet. We use *Friends*(*u*) to represent all the users in a friendship with user *u*.

Most social tagging sites have a service enabling users to create and join special groups. Users can post messages and share content to the group. This social network is also an explicit one, since members in the same group have direct connections with each other. Thus, for our purpose, we categorize group members into *Friends* as well.

We also consider another kind of social network called common interest network [2]. It is different from the traditional explicit social networks which are built up by adding friends or joining groups. The common interest network is *implicit* in nature, and is formed based on similar tagging behaviors. The items posted by a person and the tags used can be considered indicators of that person's interests. Linking people together whose tagging behaviors overlap significantly can implicitly form common interest networks. Users do not necessarily add each other as *Friends* when they have common interests. However, this social network may bring more relevant and interesting search results to the user.

The common interest network can be computed by considering the overlap in tagged items between users. Let *Items*(*u*) be the set of items tagged by the user *u* with any tag. Using *Links*(*u*) to represent the common interest network for the user *u*, we could define that another user v is in *Links*(u) iff a large fraction of the items tagged by u are also tagged by v, as follows: $|Items(u) \cap Items(v)| / |Items(u)| > \theta$, where θ is a given threshold.

The social networks we used in this chapter, namely friendship (*Friends*) and common interest networks (*Links*), are both very important social networks among social tagging sites. Different websites may have different names or forms; however, most of their social networks typically fall into these two main categories.

Naturally, different users have different social networks. As a result, when searching within a user's particular social networks, the top-ranked answers will be user-dependent.

3.3.3 Problem Statement

Given a query $Q = t_1,\ldots,t_n$ with n terms, issued by user u, and a number k, we want to efficiently return the top-k items with the highest overall scores. Our search strategy is user-focused, giving different results to different users, even when the query is the same. Our search strategy considers the user's multiple social networks. Moreover, the top-k results returned take into account the tagging behaviors' temporal information. For simplicity, tags and keywords are treated the same, and our framework deals with exact string matching.

3.4 Scoring Function

We first demonstrate how to score the items for a user's specific query. The static scoring functions for each social network component are initially discussed without timestamps, and are combined together to form an overall scoring function. A method for combining weight assignments based on user classification is then discussed. Finally, temporal information of tagging behaviors is added and temporal scoring functions are examined.

3.4.1 Multiple social network components

The overall static scoring function needs to aggregate three social network components: friendship (*Friends*), common interest network (*Links*), and global connection (*Global*).

Given a user *u*, the friendship component score of an item *i* for a tag t is defined as the number of users in *u*'s *Friends* who tagged *i* with tag *t*:

$$score_{Friends}(i,u,t) = |Friends(u) \cap \{v \mid Tagging(v,i,t)\}| \tag{1}$$

Similarly, the score from common interest network is defined as the number of users in *u*'s *Links* who tagged *i* with tag *t*:

$$score_{Links}(i,u,t) = |Links(u) \cap \{v \mid Tagging(v,i,t)\}| \tag{2}$$

Besides the above two score component from a user's social networks, we also consider the global effect on scoring. Not everyone is an active participant and/or has large personal social networks; if we only use the local social network scoring, the search effectiveness may decrease. The *Global* score is defined as the total number of users in the whole website tagged item *i* with tag *t*:

$$score_{Global}(i,t) = |\{v \mid Tagging(v,i,t)\}| \tag{3}$$

The global score is thus user-independent; it is only related to the corresponding item and tag, so for the same item and tag, the *Global* component score is the same for all users.

As a result, the static overall score of item *i* for user *u* with one tag *t* is an aggregate function of the weighted scores from the three components:

$$score_{Overall}(i,u,t) = w_1 * score_{Global}(i,t) + w_2 * score_{Friends}(i,u,t) + w_3 * score_{Links}(i,u,t) \quad \textbf{(4)}$$

where w_i is the weight of each component and $\sum_{i=1}^{3} w_i = 1$

Since a query contains multiple tags, we also define the static overall *SCORE* of item *i* for user *u* with the whole query $Q = t_1,\ldots,t_n$ as the sum of the scores from individual tags, which is a monotone aggregation function:

$$SCORE(i,u) = \sum_{j=1}^{n} score_{Overall}(i,u,t_j) \quad \textbf{(5)}$$

3.4.2 User Classification

Different weight assignments of components can generate different overall scores. Meanwhile, users may have different trusts on each component of the scoring function. So finding an efficient approach to set the weights is far from trivial.

There are several ways to assign component weights. Machine learning methods can be used to get "optimal" solutions. However, these need the definition of "optimal" and a large amount of user feedback data for training. Also, statistics algorithms need user log records and exploring data in the website, which are not easy to access either. For simplicity, here we use a user classification method based on the social networks size and recommend weight assignments for each class.

Users in social tagging sites have different usage patterns and degrees of participation in their social networks. Some users have many friends, while some may only have few. Also, for tagging, some users do frequent tagging and thus have a lot of tagged items; while others may not tag as much. As an example, we randomly collected 100 users in del.icio.us as shown in Table 9. A user can bookmark a URL with several tags in del.icio.us, and the friendship social network is called "Network". One can observe huge differences of usage pattern among the del.icio.us users.

Table 9: 100 randomly collected users in del.icio.us

	Maximum	Minimum	Average	Standard deviation
Bookmarks	29942	52	1769.38	4272.27
Network	100	0	15.82	21.19

In our general framework, we use three categories for each social network component, described as: many, some and few; nine classes are shown in Table 3. The users in the same class have similar usage patterns and degrees of participation, as their social networks have similar sizes.

Table 10: User classification

User		*Friends*		
		many	*some*	*few*
Links	*many*	Class 1	Class 2	Class 3
	some	Class 4	Class 5	Class 6
	few	Class 7	Class 8	Class 9

Within this classification, we assume that users in the same class have similar degree of trust on each social network scoring component. Then we can give a recommendation of weight assignments for users in each class. For example, a user in Class 1 has a large common interest network and has a lot of directly added friends. In Class 1, then, w_1 and w_2 could be set higher than w_3, because this user may wish to get more relevant and attractive information from his/her *Friends* and *Links*. In another example, a user in Class 7 only has a small common interest network, but he/she has a lot of friends. So such user may trust more on his/her *Friends* and have a high weight value w_2.

3.4.3 Temporal Scoring Functions

We believe that ranking results will be more attractive to users not only based on their relevance, but also on popularity and *freshness*; hence the temporal information of tagging behaviors is important. For example, one item may be more interesting if it is recently added. In this case, a simple interpretation of freshness is the first date the item was posted. However, not all new posts are popular, and not all popular posts are new. A more subtle way may consider how many recent tagging behaviors have targeted an item.

Our basic approach is to divide the tagging behaviors into multiple time slices, based on their time stamps for our scoring functions. We use *m* to denote the number of time slices and adjust the weights of different time slices based on their recency (or freshness). A higher weight is set to tagging behaviors occurring in the current time slice, and a lower weight to tagging behaviors in earlier time slices.

We use decay factor *a* ($0 < a < 1$) to penalize the count score from old time slices. Thus, the temporal score of *Global* component of item *i* with tag *t* can be defined as:

$$T\text{-}score_{Global}(i,t) = \sum\nolimits_{s=1}^{m} score_{Global}(i,t,s) * a^{m-s} \tag{6}$$

where $score_{Global}(i,t,s)$ is the global score of item i with tag t at time slice s, with $s = m$ being the current time slice.

We demonstrate the importance of temporal information for search with a real example. We used tag "kdd" as keyword in a search on del.icio.us on June 1st, 2009, using the search function provided, and got the top-5 results including "KDD Cup 2007", "KDD 2008" and "KDD 2009"[1]. The search revealed the three results in that order because the static total number of tags as "kdd" added to each item, which were *24*, *16* and *15* respectively. When we looked at the timestamps of these tagging behaviors, we found that "KDD Cup 2007" has many "old" tags, although it has the biggest total number of tags as "kdd". The details are shown in Figure 10.

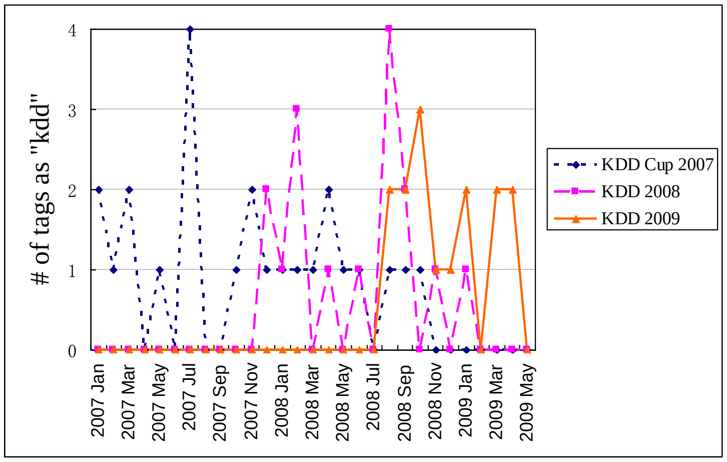


Figure 10: Monthly number of tags as "kdd" of three top results

Using temporal ranking in this example, we can set $a = 0.5$, $m = 5$, and separate the time slices with a length of 6-month for each. Then the temporal global component scores ($T\text{-}score_{Global}$) for "KDD Cup 2007" "KDD 2008" and "KDD 2009" are 5, 6.75 and 11

[1] The other two results in top-5 are "UCI KDD Archive" and "KDnuggest"

respectively. The ranking order will be changed as “KDD 2009”, “KDD 2008” and “KDD Cup 2007” which reflects the tag freshness.

The temporal scoring functions for *Friends* and *Links* components are defined similarly with the temporal factors in *Global*:

$$T\text{-}score_{Friends}(i,u,t) = \sum_{s=1}^{m} score_{Friends}(i,u,t,s) * a^{m-s} \tag{7}$$

$$T\text{-}score_{Links}(i,u,t) = \sum_{s=1}^{m} score_{Links}(i,u,t,s) * a^{m-s} \tag{8}$$

The temporal overall scoring function of item *i* for user *u* with tag *t* is:

$$T\text{-}score_{Overall}(i,u,t) = w_1 * T\text{-}score_{Global}(i,t) + w_2 * T\text{-}score_{Friends}(i,u,t) + w_3 * T\text{-}score_{Links}(i,t) \tag{9}$$

Therefore, the temporal scoring for whole query is:

$$T\text{-}SCORE(i,u) = \sum_{j=1}^{n} T\text{-}score_{Overall}(i,u,t_j) \tag{10}$$

3.5 Temporal Ranking Algorithm

To compute the top-*k* items with query tags for a particular user, items are organized in inverted lists with some information pre-computed, so that the well-known top-*k* algorithm can be adapted.

Typically, one inverted list is created for each keyword and each entry contains the identifier of a document along with its score for that keyword [5]. For our framework, when the query is composed of multiple tags, we need to access multiple lists and apply the top-*k* processing algorithms.

One straightforward method is to have one inverted list for each (tag, user) pair and sort items in each list according to the temporal overall score (*T-score*$_{Overall}$) for the tag *t* and user *u*. However, there are a lot of users in social tagging sites (del.icio.us has over 5 million users). If we create inverted lists per keyword for each user, there will be too many inverted lists and thus large space is required.

Another solution is to factor out the user from each inverted list by using upper-bound scores [2]. Since we use the number of users as the static score without normalization and set all three social network component with the same temporal factors for a query, for the same item i with the same tag t, no matter which user, we have $T\text{-}score_{Friends}<=T\text{-}score_{Global}$ and $T\text{-}score_{Links}<=T\text{-}score_{Global}$.

As a result, temporal global score is an upper-bound of temporal overall score for all the users, because $T\text{-}score_{Overall}=w_1*T\text{-}score_{Global}+w_2*T\text{-}score_{Friends}+w_3*T\text{-}score_{Links} <=w_1*T\text{-}score_{Global}+w_2*T\text{-}score_{Global}+w_3*T\text{-}score_{Global}=T\text{-}score_{Global}$. Since the global component scoring is user-independent, we can create only one list for each keyword along with the temporal global scores ($T\text{-}score_{Global}$) as an upper-bound of the user-based temporal overall scores ($T\text{-}score_{Overall}$).

In our framework, the temporal factor is designed as adjustable for users. It is impossible to know a user's choice in advance, so the temporal factors may also need to be factored out from the inverted lists. The static global scores ($score_{Global}$) is an upper-bound for the temporal global scores ($T\text{-}score_{Global}$), since the static scores correspond to the temporal ones with $a = 1$. Therefore, the final upper-bound scores used in our inverted lists are the static global scores. The entries of lists have a form:

$$<\text{item}, \{(\text{user1}, \text{time1}), (\text{user2}, \text{time2}),\ldots\}, score_{Global}>$$

which includes item ID, all users who tagged the item with that tag along with timestamps, and the static global score.

We can thus extend Fagin's classic top-*k* TA algorithm [21] to rank the items listed in the order of static global scores $score_{Global}$ as the upper-bound for temporal overall scores in Algorithm 3.1. Given a user *u*, query *Q* and *k*, *Friends*(*u*), *Network*(*u*), weights w_1, w_2, and w_3 for that user's class are identified. We access the relevant inverted lists sequentially in parallel. When an item o is seen for the first time, we compute its exact temporal overall score ($T\text{-}score_{Overall}$) with a "local" aggregation function of three component temporal scores. For every item entry, we have all the IDs of tagging users and timestamps, so we can compute $T\text{-}score_{Global}$ directly, and $T\text{-}score_{Friends}$, $T\text{-}score_{links}$ by checking with user's *Friends* and *Links*. Then, we do the random access to other lists and perform computation of $T\text{-}score_{Overall}$. When at least one of $T\text{-}score_{Overall} = 0$, we set $T\text{-}SCORE = 0$ for that item. This means an item must include all query tags; otherwise it will be scored 0 for *T-SCORE*. After that, we can have the exact temporal overall score *T-SCORE* of this item for the whole query *Q* and check whether it can be swapped into top-*k* sorted heap. Meanwhile, a *Thres*, the sum of bottom bounds of all lists is recorded and updated. The algorithm stops whenever the score of the *k*th item in the heap is no less than the *Thres*, and outputs the top-*k* results.

Other top-*k* algorithms like NRA can easily be extended in a similar way. We also notice that the upper-bound scores can be coarse at times. [2] explores the use of clustering to save processing time. This methodology could be adapted into our framework as well; however it is left as future work.

Algorithm 3.1 Temporal Top-*k* Ranking Algorithm

```
Require: User $u$, Query $Q$, and $k$
1:  Get $Friends(u)$, $Links(u)$, weights $w_1$, $w_2$, $w_3$, and temporal factors;
      Open inverted lists for each keyword $t \in Q$;
2:  while score of $k$th heap item <= $Thres$ do
3:     Do sequential access in parallel to each of the list $L_i$;
4:     Once a new object $o$ is seen, get the exact $T\text{-}Score_{Overall}$ of $o$ in that list;
5:     Do random access to the other lists, and get the $T\text{-}Score_{Overall}$ of $o$;
6:     if at least one the $T\text{-}Score_{Overall} = 0$ then
7:        Set the exact T-SCORE of $o$ to be 0;
8:     end if
9:     else
10:       Compute T-SCORE by sum up $T\text{-}Score_{Overall}$;
11:       if $o$'s T-SCORE > $k$th score in top-$k$ heap then
12:          Replace $k$th item with $o$; keep heap sorted;
13:       end if
14:    end else
15:    Update $Thres$ as sum of bottom bounds of all lists;
16: end while
17: Output the heap as top-$k$ results
```

3.6 Experimental Evaluation

Below we evaluate our proposed framework and method for the temporal top-*k* search problem using various real datasets. An extended collection of our experiments and evaluations appears in [30].

3.6.1 Data Collections

To evaluate the effectiveness of our scoring functions and query process methods, we collected datasets from CiteULike (http://www.citeulike.org), an academic article social tagging site.

In CiteULike, articles are stored with their metadata, abstracts, and links to the papers at the publishers' websites. Users can add their academic papers to their online library with tags and personal comments. Friendship and social networks can be created between users through “Connections”. In addition, CiteULike also allows a user to set up and join groups that share academic or topical interests. Many other services, such as “Watchlist” and “Neighbours”, are also offered.

CiteULike provides some datasets from their core database.[2] However, to get more recent data, we further crawled datasets before 2009.7.1. After filtering, our datasets comprised approximately 104,000 unique articles posted by approximately 4,600 unique users using approximately 35,000 unique tags.

3.6.2 Top-k result lists

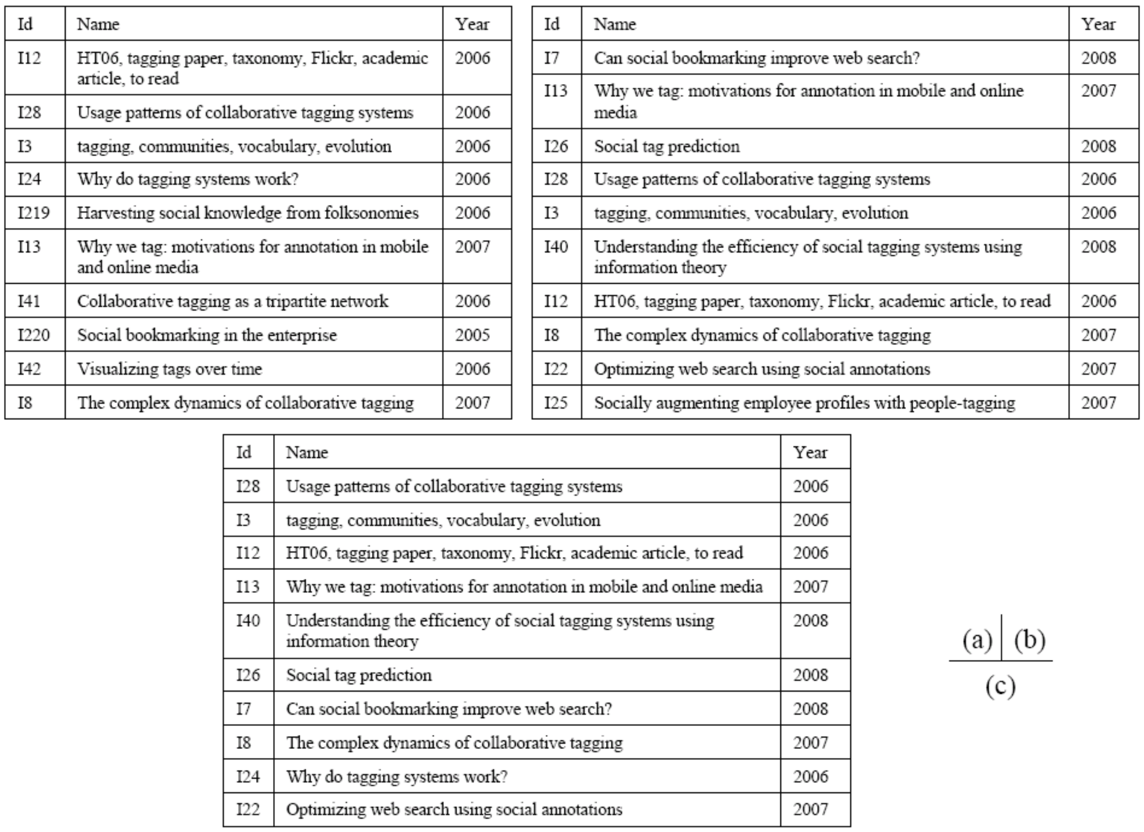
We proceed with experimental top-*k* results based on different temporal factors and weight assignments.

We first evaluate the effects of temporal information; we search for top-*k* results, and only consider the global scoring functions. Figure 11 depicts the top-10 results for the search query “social tagging” of two tags, “social” and “tagging”. We divide the time range of our datasets into six-month periods, starting from the most recent 2009.1.1 – 2009.6.30 to earlier time slices, which will remain the same throughout this work.

Changes of temporal factors differentiate the top-k lists. If the decay factor $a = 1$, then the scoring function is the same as the static scoring function, and the results are ranked by the total number of users who tagged the item with the query keywords, without considering any temporal information. On the other hand, with decay factor $a = 0$, the query considers recency and shows the ranking of the number of tagging behaviors only

[2] http://www.citeulike.org/faq/data.adp

in the most recent time slice. A balanced decay factor *a* = 0.5, is neither too high to miss the temporal information and freshness, or too low to lose classic popular items. The top-10 results are shown in Figure 11 in ranking order, along with the items' Ids, Names and Years they published.



Id	Name	Year
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read	2006
I28	Usage patterns of collaborative tagging systems	2006
I3	tagging, communities, vocabulary, evolution	2006
I24	Why do tagging systems work?	2006
I219	Harvesting social knowledge from folksonomies	2006
I13	Why we tag: motivations for annotation in mobile and online media	2007
I41	Collaborative tagging as a tripartite network	2006
I220	Social bookmarking in the enterprise	2005
I42	Visualizing tags over time	2006
I8	The complex dynamics of collaborative tagging	2007

Id	Name	Year
I7	Can social bookmarking improve web search?	2008
I13	Why we tag: motivations for annotation in mobile and online media	2007
I26	Social tag prediction	2008
I28	Usage patterns of collaborative tagging systems	2006
I3	tagging, communities, vocabulary, evolution	2006
I40	Understanding the efficiency of social tagging systems using information theory	2008
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read	2006
I8	The complex dynamics of collaborative tagging	2007
I22	Optimizing web search using social annotations	2007
I25	Socially augmenting employee profiles with people-tagging	2007

Id	Name	Year
I28	Usage patterns of collaborative tagging systems	2006
I3	tagging, communities, vocabulary, evolution	2006
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read	2006
I13	Why we tag: motivations for annotation in mobile and online media	2007
I40	Understanding the efficiency of social tagging systems using information theory	2008
I26	Social tag prediction	2008
I7	Can social bookmarking improve web search?	2008
I8	The complex dynamics of collaborative tagging	2007
I24	Why do tagging systems work?	2006
I22	Optimizing web search using social annotations	2007

Figure 11: Top-10 results for different decay factor *a*. (a) *a* = 1; (b) *a* = 0; (c) *a* = 0.5

These three lists reveal some interesting observations. First, when the decay factor *a* decreases from 1 to 0, the average "age" of the top-10 items becomes younger. Second, some recently popular articles improve their rating when *a* decreases, such as I13 and I7. Meanwhile, some classic items, due to their "old age", rank down dramatically. However, when *a* = 0.5, the top-10 lists include both classic and fresh popular items. Last, some articles (such as I12, I28, I3, I13, and I8) show up in all three lists. These articles are very important and stay in top-10 lists despite the decay factor changes.

We further show the top-10 results as related to different weight assignments of the three social network components, *Global*, *Friends* and *Links*. Here the temporal factors remain unchanged using the same query: “social tagging”.

CiteULike has an explicit friendship social network called “Connections”, however, due to privacy reasons such data is not available. It also has the “Groups” friendship, so we use *Friends*(*u*) to include all users who are members in the user *u*’s Groups.

Id	Name
I28	Usage patterns of collaborative tagging systems
I3	tagging, communities, vocabulary, evolution
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read
I13	Why we tag: motivations for annotation in mobile and online media
I40	Understanding the efficiency of social tagging systems using information theory
I26	Social tag prediction
I7	Can social bookmarking improve web search?
I8	The complex dynamics of collaborative tagging
I24	Why do tagging systems work?
I22	Optimizing web search using social annotations

(a)

Id	Name
I55	Tree, funny, to_read, google: what are tags supposed to achieve?
I26	Social tag prediction
I8	The complex dynamics of collaborative tagging
I7	Can social bookmarking improve web search?
I1	Efficient top-k querying over social-tagging networks
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read
I28	Usage patterns of collaborative tagging systems
I61	The Language of Folksonomies: What Tags Reveal About User Classification
I51	Tagging: People-powered Metadata for the Social Web (Voices That Matter)
I3	tagging, communities, vocabulary, evolution

(b)

Id	Name
I3	tagging, communities, vocabulary, evolution
I8	The complex dynamics of collaborative tagging
I7	Can social bookmarking improve web search?
I1	Efficient top-k querying over social-tagging networks
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read
I69	What drives content tagging: the case of photos on Flickr
I42	Visualizing tags over time
I40	Understanding the efficiency of social tagging systems using information theory
I22	Optimizing web search using social annotations
I19	Real-time automatic tag recommendation

(c)

Id	Name
I3	tagging, communities, vocabulary, evolution
I12	HT06, tagging paper, taxonomy, Flickr, academic article, to read
I7	Can social bookmarking improve web search?
I8	The complex dynamics of collaborative tagging
I1	Efficient top-k querying over social-tagging networks
I28	Usage patterns of collaborative tagging systems
I40	Understanding the efficiency of social tagging systems using information theory
I26	Social tag prediction
I13	Why we tag: motivations for annotation in mobile and online media
I69	What drives content tagging: the case of photos on Flickr

(d)

Figure 12: Top-10 results of different weights (a) $w_1 = 1$; (b) $w_2 = 1$; (c) $w_3 = 1$; (d) recommended

Moreover, CiteULike has a service named “Neighbours” which is quite similar to our common interest network in social tagging. Neighbours of *u* are the users who have bookmarked the same articles as *u*. To remove the “long tail”, it only shows neighbors who share at least the median number of articles. As a result, we used the “Neighbours” as our *Links* social network component for our experiments.

We picked up a specific user from our user dataset who uses the tags “social” and “tagging” very frequently. This user has 13 *Friends* and 23 *Links*, and is then categorized into Class 2 in the users classification. We gave a recommended weight assignment for this class as $w_1 = 0.1$, $w_2 = 0.3$, and $w_3 = 0.6$. The temporal scoring functions use the decay factor $a = 0.5$, while the time slices remain as six months. The top-10 results for different weight assignments are shown in Figure 12.

The first three lists are generated using only one social network component each time. As seen, some articles, I3, I12, I7, and I8, stay in top-10 among all three lists. Note that the top-10 list with our recommended weights for the three social networks also includes these important articles with relatively high ranking positions. Therefore, our top-k search framework using multiple social networks can embrace manifold opinions without losing any important items.

3.6.3 NDCG Measurements

We now proceed with an evaluation of our framework using the NDCG standard for measuring the search quality. Classical IR metrics, namely NDCG, MRR, and MAP [38] are widely used for measuring search quality. Here we use the NDCG (normalized discounted cumulated gain) measurement [33] to evaluate the performance of our experiments.

Every item in top-*k* lists is given a corresponding human judgment scoring from 0 to 3 (0=Bad, 1=Fair, 2=Good, 3=Excellent). The cumulated gain is computed by summing up and the discounted cumulated gain vector is defined recursively as:

$$DCG = \begin{cases} G[1], & \text{if } i = 1 \\ DCG[i-1] + G[i] / \log_2 i, & \text{otherwise} \end{cases}$$

The DCG vectors can be normalized by dividing them by the corresponding ideal DCG vectors (all score 3), so the normalized value ranges in [0, 1]: $NDCG[i] = DCG[i] / DCG_{Ideal}[i]$

Our human judgments of top-10 results are based on relevance and attractiveness (popularity and freshness) for particular query tags. Since it is difficult to ask real users, we elicit the help of graduate student volunteers to provide us with their educated judgments. Different queries may prefer different temporal factor settings. We thus used two different sets of popular query tags, and ask three volunteers to judge each query. For set-1, the queries are “social-network” and “tagging”. These are popular and very *hot* recently. For set-2, we use “algorithm” and “database” as queries, because they are very popular and *classic*.

We change the decay factor *a* from 1 to 0 with the same time slice division as six months. Meanwhile, we only evaluate the global temporal scoring ($T\text{-}score_{Global}$) to factor out user diversity. The average NDCG results are shown in Figure 13. From this figure, we observe that different kinds of queries have different preferences. *Hot* queries may prefer recent tagging behaviors much more than *classic* queries. And for *classic* queries, the total number is a very important factor for search. But for both sets, the average NDCG peaks when *a* is neither too high nor too low.

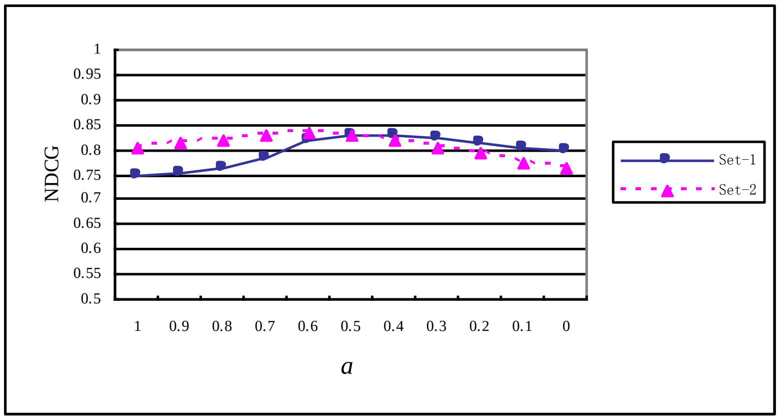


Figure 13: Average NDCG results for different decay factor *a*

Here we evaluate the NDCG of different user classes with different weight assignments for each social network. The users are classified based on the size of their social networks. The distributions of our user dataset are shown in Figure 14. We set *few* as *0~5*, *some* as *6~15*, and *many* as *15+* for both *Friends* and *Links*.

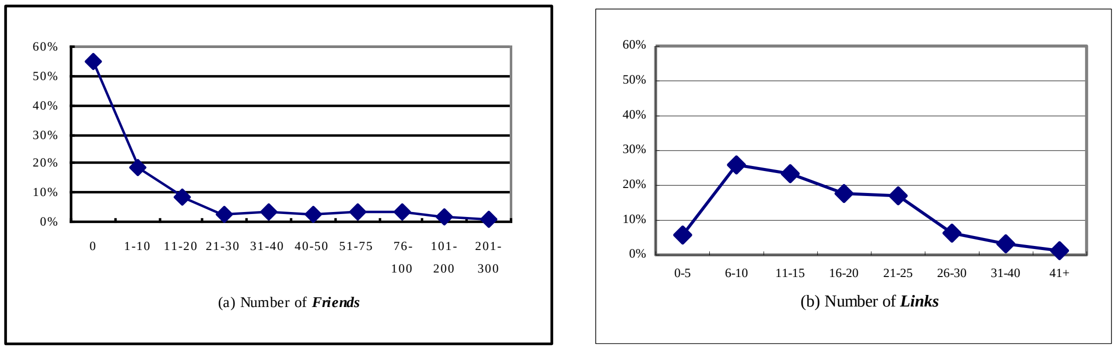


Figure 14: Distributions of the size of users' social networks

Based on the user classification in Table 10, we provided an example recommendation of weight assignments for six representative classes as listed in Table 11. The decay factor was set as a = 0.5 and the time slices were six months. We tested two queries—"tagging" and "algorithm", picked up two users from our dataset for each class, and used two volunteers to evaluate. Then we extracted the average NDCG.

Table 11: Recommendation of weight assignments

Class	Recommendation	Class	Recommendation
1	r1: $w_1 = 0.1$, $w_2 = 0.45$, $w_3 = 0.45$;	5	r5: $w_1 = 0.2$, $w_2 = 0.4$, $w_3 = 0.4$;
2	r2: $w_1 = 0.1$, $w_2 = 0.3$, $w_3 = 0.6$;	6	r6: $w_1 = 0.2$, $w_2 = 0.3$, $w_3 = 0.5$;
3	r3: $w_1 = 0.1$, $w_2 = 0.1$, $w_3 = 0.8$;	9	r9: $w_1 = 0.4$, $w_2 = 0.3$, $w_3 = 0.3$;

First we examined whether our multiple social network components method works better than using only one component. As shown in Figure 15, in all six representative classes, our multiple-component method produced better NDCG than any other one-component method.

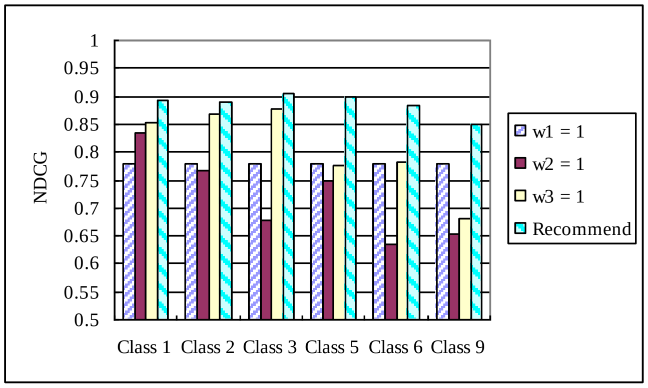


Figure 15: Average NDCG for different weight assignments across six classes

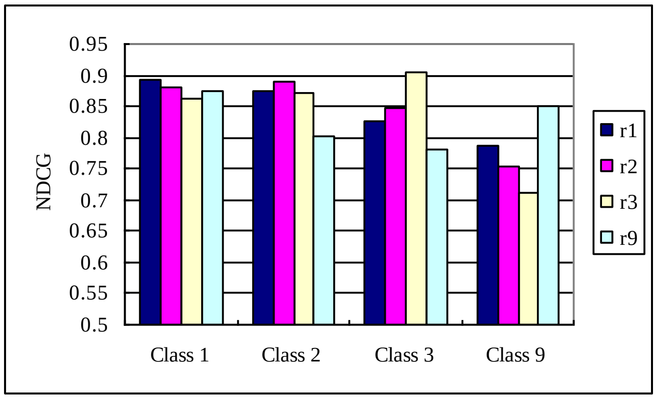


Figure 16: Average NDCG for different recommendations across four classes

Then we tested if our respective weight assignments recommended for each class were performing better than other multiple component assignments. In this experiment, our recommended weight assignments were r1 for Class 1, r2 for Class 2, r3 for Class 3, and r9 for Class 9. We tested all these four assignments for each class. As seen in Figure 16, our recommended weight assignments performed better than the other assignments for each specific class. For example, when running assignments r1, r2, r3, and r9 for Class 1, r1 performed better. Similarly, r2 performed better for Class 2, r3 for Class 3, and r9 for Class 9.

3.7 Conclusions

In this chapter, we presented an experimental study of temporal top-*k* search in social tagging sites using two main types of social networks, friendship and common interest networks, to model the scoring functions along with the global component. To set the weights of each scoring component for different users, a classification of users is proposed based on the size of users' social networks. To improve the popularity and freshness of ranking results, the timestamps of tagging behaviors are recorded and separated into multiple time slices. Then temporal scoring functions are formed by giving higher weights to more recent time slices. In addition, an efficient temporal top-*k* algorithm for ranking is proposed which stores inverted lists for each tag with static global scores as upper-bound of each item. Experimental evaluation on real social tagging website datasets shows that our framework and methodology work well in practice.

Chapter 4

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

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 chapter, 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 multi-version 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 a large real world dataset demonstrate the advantages of the newly proposed data organization and indexing approaches.

4.1 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 (http://en.wikipedia.org/), where textual content is explicitly version-controlled. Similarly, web archiving applications such as the Internet Archive (http://www.archive.org) and the European Archive (http://europarchive.org/) 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 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.

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.

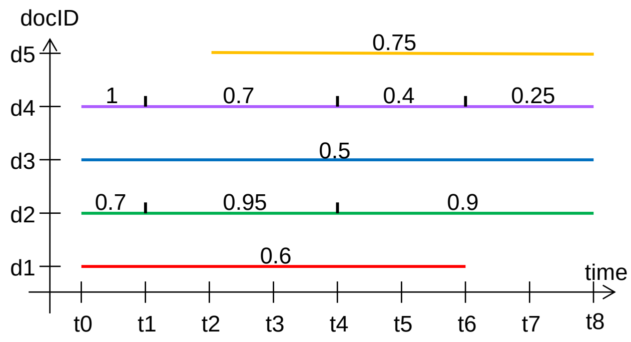
The rest of this chapter is organized as follows. Preliminaries and related work are introduced in section 4.2. Our novel approaches appear in section 4.3. Different query definitions of time-interval top-k queries are presented in section 4.4. All techniques are comprehensively evaluated and compared in a series of experiments in section 4.5 while the conclusions appear in section 4.6.

4.2 Preliminaries and Related Work

4.2.1 Definitions

The data model for versioned document collections was formally introduced in [11], and used by later works [10, 3, 4]. 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: $d_i = \{d_i^1, d_i^2, \ldots, d_i^{m_i}\}$. Each version has a semi-closed validity time-interval (or lifespan) $life(d_i^j) = [t_s, t_e)$. Moreover, it is assumed that different versions of the same document have disjoint life spans. An example of five documents and their versions appears in Figure 17; 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 [5], language models [43] and Okapi BM25 [47]. 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*.



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

Figure 17: Example of versioned documents with scores for one term

In order to support temporal queries, the inverted file index must also contain temporal information. Thus [11] 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, [11] 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: $D^{[lb,rb]} = \{d_i^j \in D \mid [lb, rb] \cap life(d_i^j) \neq \varnothing\}$. 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.

4.2.2 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.

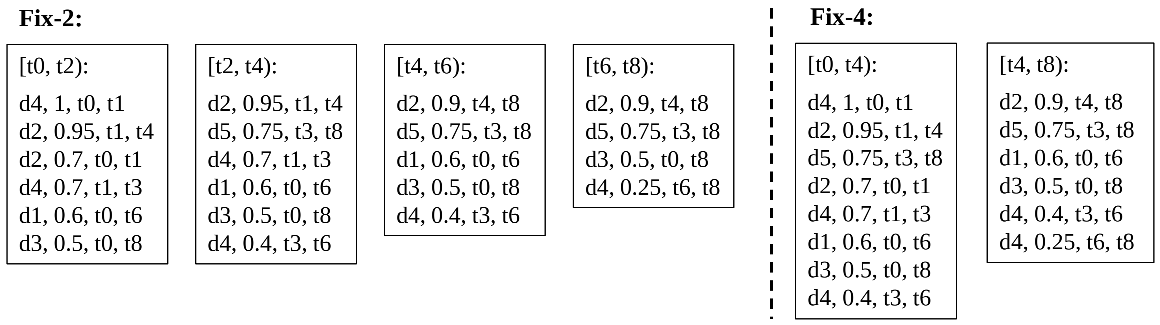
The sort-order of the index lists is also important. One natural choice is to sort each list in score order. This method (score-order) enables the classical top-k algorithms [21] to stop early after having identified the *k* highest scores with qualified lifespan. Another suitable sorting choice is to 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 [10, 3]. 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, [3] 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. Figure 18 shows the Fix-2 and Fix-4 sub-lists of our running example from Figure 17, 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.



Fix-2:

[t0, t2):	[t2, t4):	[t4, t6):	[t6, t8):
d4, 1, t0, t1	d2, 0.95, t1, t4	d2, 0.9, t4, t8	d2, 0.9, t4, t8
d2, 0.95, t1, t4	d5, 0.75, t3, t8	d5, 0.75, t3, 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, 0.7, t1, t3	d1, 0.6, t0, t6	d3, 0.5, t0, t8	d4, 0.25, t6, t8
d1, 0.6, t0, t6	d3, 0.5, t0, t8	d4, 0.4, t3, t6	
d3, 0.5, t0, t8	d4, 0.4, t3, t6		

Fix-4:

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

Figure 18: Time Interval Based Slicing sub-list examples

Stencil Based Partitioning. Another index partitioning method along the time-axis was proposed in [26]. 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 Figure 17) is shown in Figure 19.

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.

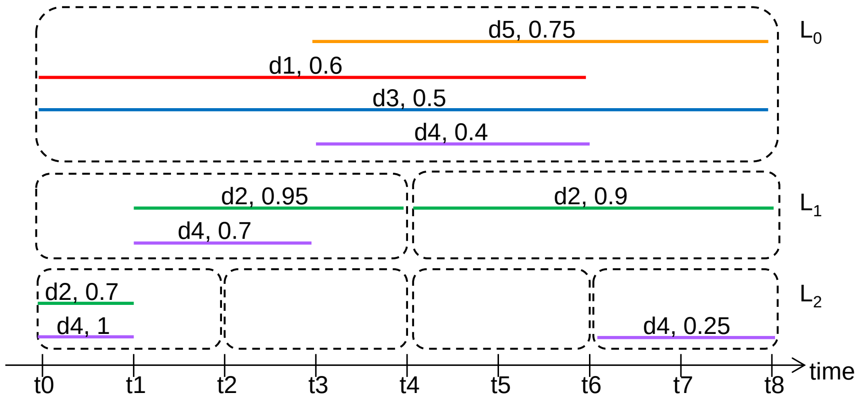
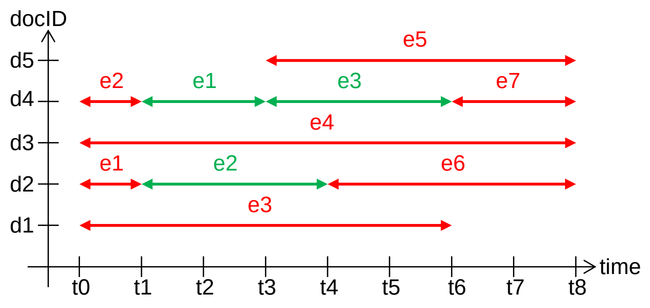


Figure 19: Stencil-based partitioning with 3 levels and b = 2

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 [4] 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: $\forall p, q \in g_i, ts(p) \le ts(q) \Rightarrow te(p) \le te(q)$. An optimal greedy algorithm for creating this partitioning is given in [4]; an example of temporal sharding for the term list from Figure 17 is shown in Figure 20.



Shard_1:	Shard_2:
e1: d2, 0.7, t0, t1	e1: d4, 0.7, t1, t3
e2: d4, 1, t0, t1	e2: d2, 0.95, t1, t4
e3: d1, 0.6, t0, t6	e3: d4, 0.4, t3, 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	

Figure 20: 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.

4.3 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 Figure 17 is shown in Figure 21.

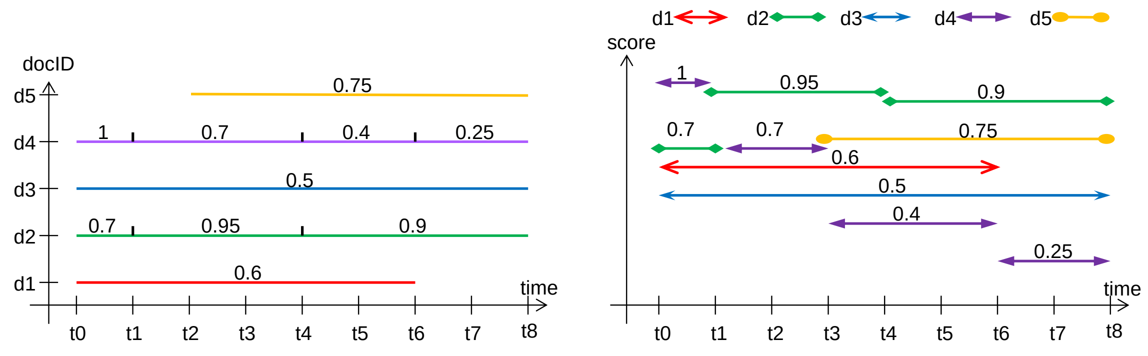


Figure 21: The Score-Time view of the versioned documents

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 4.3.1), 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 4.3.2). 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.

4.3.1 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 ($i = 1,2,\ldots$), 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 Figure 17) is shown in Figure 22(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.

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).

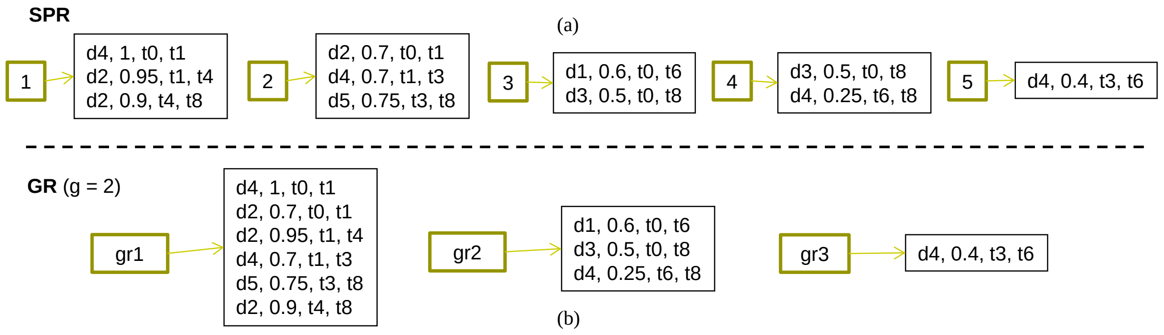


Figure 22: Ranking position based partitioning

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 Figure 17) with $g = 2$ is shown in Figure 22(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 [52] 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.

4.3.2 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 [9], 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 [9].

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) [9].

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 look-ups 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*.

4.4 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.

4.4.1 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 $D^{[lb,rb]} = \{d_i^j \in D \mid [lb,rb] \cap life(d_i^j) \neq \varnothing\}$. 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.

4.4.2 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 [11]:

MIN. This model judges the relevance of a document based on the minimum score. It is formally defined as: $drs(d_i, lb, rb) = \min\{s(d_i^j) \mid [lb,rb] \cap life(d_i^j) \neq \varnothing\}$. 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: $drs(d_i, lb, rb) = \max\{s(d_i^j) \mid [lb, rb] \cap life(d_i^j) \neq \varnothing\}$. 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 **TKTI$^{\mathbf{A}}$** = (*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.

4.4.3 Consistent Top-k Time-Interval 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 $\mathbf{TKTI}^{\mathbf{C}} = (q, lb, rb, k)$ 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 [54] 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 [54].

4.5 Experimental Evaluations

4.5.1 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 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 (http://google.com/zeitgeist/) 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. More details can be found in [31].

4.5.2 Comparison Results

First, the space usage for all implemented methods on both the US-news and World-news datasets is presented in Table 12. 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 12: 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 13 (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 14 for both time-point query and 30-day time-interval queries.

Table 13: 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 14: 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.

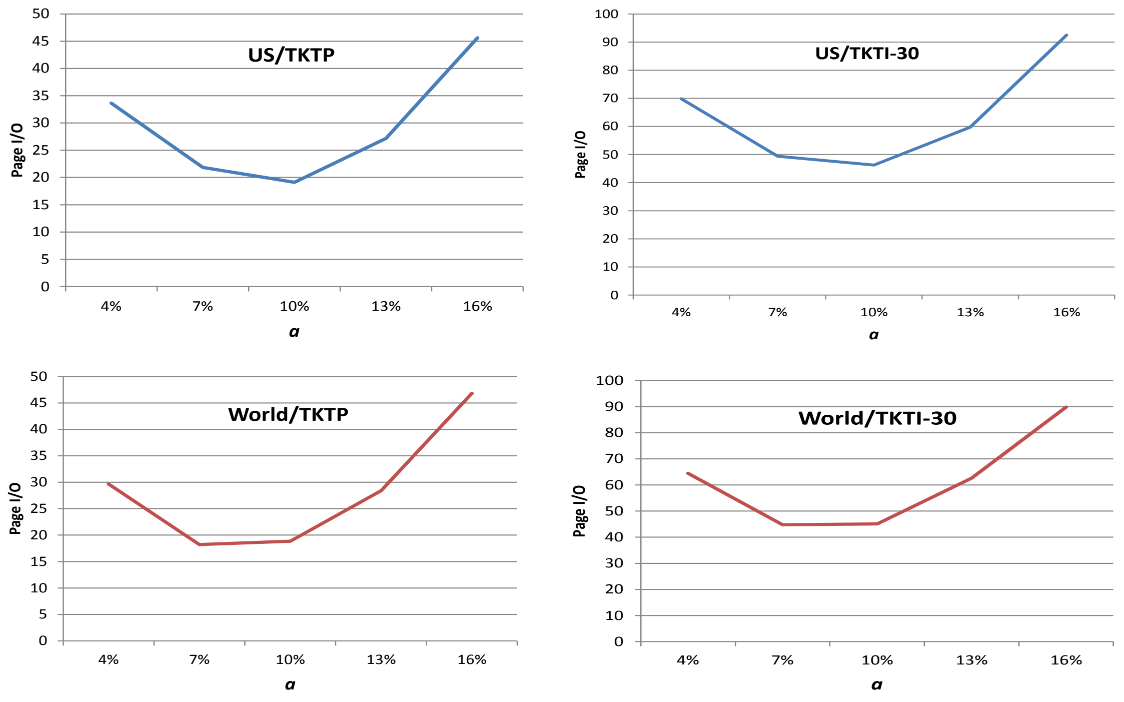


Figure 23: The Multiversion list method for different ratio a using the US and World news datasets

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 $a = 7\%$.

4.6 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.

Chapter 5

Querying Transaction-time Databases under Branched Schema Evolution

Transaction-time databases have been proposed for storing and querying the history of a database. While past work concentrated on managing the data evolution assuming a static schema, recent research has considered data changes under a linearly evolving schema. An ordered sequence of schema versions is maintained and the database can restore/query its data under the appropriate past schema. There are however many applications leading to a *branched* schema evolution where data can evolve in parallel, under different concurrent schemas. In this work, we consider the issues involved in managing the history of a database that follows a branched schema evolution. To maintain easy access to any past schema, we use an XML-based approach with an optimized sharing strategy. As for accessing the data, we explore branched temporal indexing techniques and present efficient algorithms for evaluating two important queries made possible by our novel branching environment: the vertical historical query and the horizontal historical query. Moreover, we show that our methods can support branched schema evolution which allows version *merging*. Experimental evaluations show the efficiency of our storing, indexing, and query processing methodologies.

5.1 Introduction

Due to the collaborative nature of web applications, information systems experience evolution not only on their data content but also under different schema versions. For example, Wikipedia has experienced more than 170 schema changes in its 4.5 years of lifetime [16]. Schema evolution has been addressed for traditional (single-state) database systems and issues on how data is efficiently transferred to the latest schema have been examined [15]. Consider however the case where the application maintains its past data (typically for archiving, auditing reasons etc.) which may have followed different schemas. A temporal database can be facilitated to manage the historical data, but issues related to how data can be queried under different schemas arise. The pioneering work in PRIMA system [40] addresses the issues of maintaining a transaction-time database under schema evolution by introducing: (i) an XML-based model for archiving historical data with evolving schemas, (ii) a language of atomic schema modification operators (SMOs), and (iii) query answering and rewriting algorithms for complex temporal queries spanning over multiple schema versions. Nevertheless, PRIMA considers only a linear evolution: a new schema is derived from the latest schema and at each time there is only one current schema.

In many applications, however, the schema may change in a more complex way. For instance, in a collaborative design environment, an initial schema may be branched into a number of parallel schemas whose data can evolve concurrently. Another common case of non-linear evolution is in software development management. Revision control enables the modifications and developments happening in parallel along multiple branches. The release history of Mozilla Firefox shows that 10 branches of versions have been developed and 4 more branches are on the way.

In this chapter we address the issues involved in archiving, managing and querying a branched schema evolution. In particular, we maintain the branched schema versions in an XML-based document (*BMV-document*) using schema sharing. This choice was made because the number of schema changes is relatively smaller than data changes and the hierarchal structure of XML allows for easy schema querying. The data level changes are stored in column-like tables (*BC-Tables*), one table for each temporal attribute, with the support of applicable temporal indexing. To the best of our knowledge, this is the first work to examine both data and schema evolution in a branched environment ([32]). Our contributions can be summarized as:

1. We utilize a *sharing* strategy with *lazy-mark* updating, to save space and update time when maintaining the schema branching.
2. We employ branched temporal indexing structures and link-based algorithms to improve temporal query processing over the data. Moreover, we propose various *optimizations* for two novel temporal queries involving multiple branches, the *vertical* and *horizontal* queries.
3. We further examine how to support *version merging* within the branched schema evolution environment.

4. Our experiments show the space effectiveness of our sharing strategy while the optimized query processing algorithms achieve great data access efficiency.

The rest of this chapter is organized as follows. Section 5.2 summarizes work on linear schema evolution (PRIMA). Section 5.3 introduces branched schema evolution while section 5.4 presents the BMV-Document for storing schema versions and the BC-Tables for storing the underlying data changes (with the support of branched temporal indexing). Section 5.5 provides algorithms and optimizations for efficient processing of temporal queries. The merging challenges are discussed in section 5.6 and the experimental evaluations are presented in section 5.7. Finally, conclusions appear in section 5.8.

5.2 Preliminaries

5.2.1 A linear Evolution Example

Consider the linear schema evolution shown in Table 15 and Figure 24(a), of an employee database, which is used as the basic running example in this chapter. When the database was first created at T_1, using schema version $V_{1.1}$, it contains three tables: **engineerpersonnel**, **otherpersonnel** and **job**. As the company seeks to uniformly manage the personnel information, the DBA applies first schema modification at T_2, which merges two tables **engineerpersonnel** and **otherpersonnel**, producing schema $V_{1.2}$. Each schema version is valid for all times between its start-time T_s and its end-time T_e (the time it was updated to a new schema). The rest schema versions and their respective time intervals appear as well until the latest schema $V_{1.5}$. A special value "now" is used to represent the always increasing current time.

Schema changes are represented by Schema Modification Operators (SMOs) [15]; each operator performs an atomic action on both the schema and the underlying data, like CREATE/MERGE/PARTITION TABLE, ADD/DROP/RENAME COLUMN. For example, two tables in $V_{1.1}$ were merged to one table by a MERGE TABLE operation in $V_{1.2}$. In the following discussion we will use the term SMO to denote a change operator applied to one schema without detailing which SMO was actually used.

Table 15: A linearly evolving employee database

VID	Schema Versions	T_s	T_e
$V_{1.1}$	engineerpersonnel (id, name, title, deptname) otherpersonnel (id, name, title, deptname) job (title, salary)	T_1	T_2
$V_{1.2}$	employee (id, name, title, deptname) job (title, salary)	T_2	T_3
$V_{1.3}$	employee (id, name, title, deptno) job (title, salary) dept (deptno, deptname, managerid)	T_3	T_4
$V_{1.4}$	employee (id, title, deptno) job (title, salary) dept (deptno, deptname, managerid) empbio (id, name, sex)	T_4	T_5
$V_{1.5}$	employee (id, title, deptno, salary) dept (deptno, deptname, managerid) empbio (id, name, sex)	T_5	*now*

5.2.2 XML Representation of a Linear Schema Evolution

The history of the relational database content and its schema evolution can be published in the form of XML, and viewed under a temporally grouped representation whereby complex temporal queries can be easily expressed in standard XQuery [40, 41]. The MV-Document [40] intuitively represents both schema versions and data tuples using XPath notation, as: **/db/table-name/row/column-name**. Each of the nodes, representing respectively: databases, tables, tuples, and attributes, has two more attributes, start-time (ts) and end-time (te), respectively representing the (transaction-) time in which the element was added to and removed from the database.

Consider our running example: when the three-table schema in version $V_{1.1}$ was created, three table nodes with names **engineerpersonnel**, **otherpersonnel** and **job** were created in the MV-Document, each with interval [T_1, "now"). Similarly, the nodes for their attributes etc., were added in the XML document. In $V_{1.2}$ the schema evolved into the two tables **employee** and **job**; these changes were updated in the MV-Document by changing the end-time of **engineerpersonnel** and **otherpersonnel** to T_2 (as well as the intervals of their attribute and tuple nodes). Meanwhile, a new table node for **employee** is added with interval [T_2, "now"). Since the **job** relation continues in the new version, there is no update on that table node.

To make the storage and querying of MV-Documents more scalable, [41] uses relational databases and mappings between the XML views and the underlying database system. This is facilitated by the use of H-Tables, firstly introduced in [56]. Consider the **employee (<u>id</u>, title, deptno, salary)** relation of schema $V_{1.5}$ in Table 1. Its history is stored in four H-Tables, namely: (i) a key table, **employee_key (<u>id</u>, ts, te),** that stores the interval (ts, te) during which tuple with key <u>id</u> was stored in the corresponding relation. (ii) three attribute history tables: **employee_title (<u>id</u>, title, <u>ts</u>, te), employee_deptno (<u>id</u>, deptno, <u>ts</u>, te)** and **employee_salary (<u>id</u>, salary, <u>ts</u>, te)** that maintain how the individual attributes of a tuple (identified by <u>id</u>) changed over time, and (iii) an entry in the global relation table **relations (<u>relationname</u>, ts, te)** which records the time spans covered by the various relations in the database.

5.3 Branched Schema Evolution

Many modern complex applications need to support schema branching; examples include scientific databases, collaborative design environment, web-based information systems, etc. With branched schema evolution enabled, a new branch can be created by updating the schema of a *parent* version V_p. If version V_p is a current schema version and the data populating the first schema of the new branch is adapted from the currently alive data of V_p, we have a current branching (*c-branching*). An example of c-branching appears in Figure 24(b) where the most current version of branch B_1 is $V_{1.5}$. At the current time T_6 branch B_2 is created out of $V_{1.5}$ (i.e., the B_2 creation time is T_6) by applying SMOs on the relations that $V_{1.5}$ has at T_6. For example, under branch B_2 a new attribute *status* was added in **empbio** to describe the marital status of employees. As a result, data can start evolving concurrently under two parallel schemas, $V_{1.5}$ and $V_{2.1}$. A real life scenario leading to c-branching is the case when a company establishes a subsidiary. These two companies share the same historical database (branch B_1 from T_1 to T_6) but in the future their schema and data evolve independently. Note that a version can start from any past version (*h-branching*). Here we concentrate on c-branching due to the challenges of the parallel evolving it imposes.

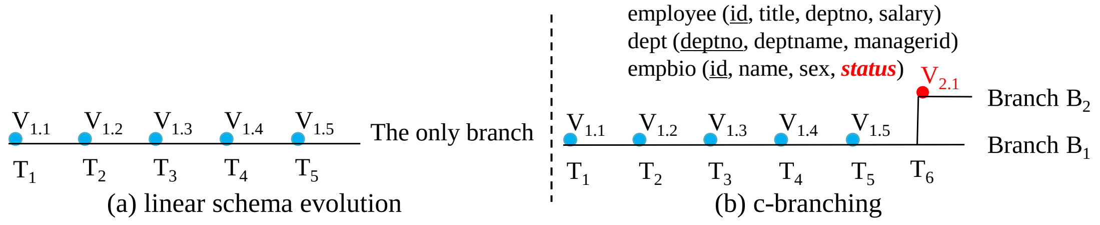

employee (id, title, deptno, salary)
dept (deptno, deptname, managerid)
empbio (id, name, sex, status)
V1.1 V1.2 V1.3 V1.4 V1.5
T1 T2 T3 T4 T5
The only branch
(a) linear schema evolution
V1.1 V1.2 V1.3 V1.4 V1.5 V2.1
T1 T2 T3 T4 T5 T6
Branch B2
Branch B1
(b) c-branching

Figure 24: Linear evolution and branching

As more branches occur, effectively the different schema versions create a *Version Tree*; an example (assuming c-branching) with six branches is shown in Figure 25, which is an extension of the branched employee DB example from Figure 24(b). Such version tree can easily display the parent-child relationship among versions and branches; this relationship information is very useful for further optimizations.

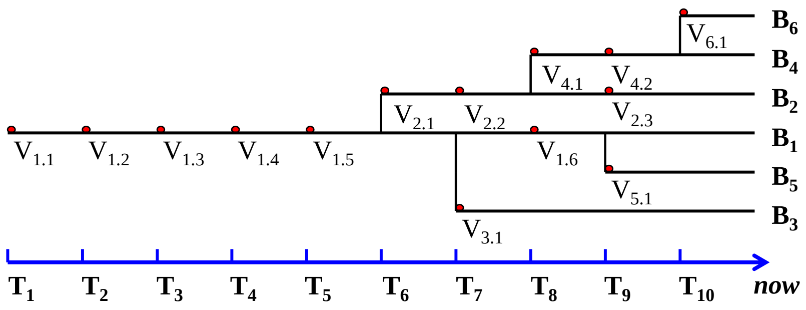


Figure 25: Example of Version Tree

The novel problems in supporting c-branching are emanated from its sharing of data: the same original data can evolve in parallel under different branches. To provide efficient access and storage in a branched environment, we use different structures to maintain the evolution of schema versions and their underlying data. Since schema changes are much less frequent, we adopt an XML-based model that enables complex querying (BMV-Document). In contrast, the data evolution over time creates large amounts of historical, disk-resident data, so our focus is on branched column tables (BC-Tables) and efficient index methods.

5.4 BMV-Document and BC-Tables

5.4.1 BMV-Document

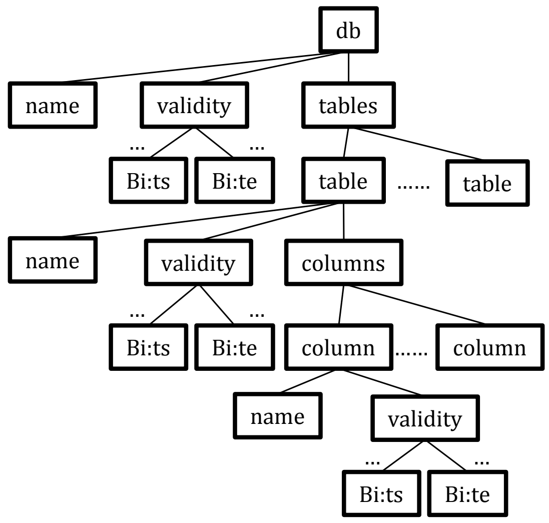


Figure 26: Illustration of BMV-Document

The BMV-Document is an extension of the MV-Document for storing the branched evolving schema versions in an XML-based representation. The main upgrades are: (i) branch identifier *bid* is needed, because a single timestamp cannot uniquely identify the appropriate schema version. (ii) The BMV-Document refers only to the schema-level storage, and does not detail the data level. (iii) The BMV-Document uses a sharing strategy between versions with various update options and a validity interval (**bid:ts**, **bid:te**) is thus required, as shown in Figure 26. When a c-branching is created, the child branch may only modify a relatively small part of its parent schema. Simply copying the schemas of all live tables and their columns from the parent version would incur storage overhead.

Schema Sharing. Consider the c-branching on B_1 that creates a new branch B_2 in Fig. 1(b). B_2's creation time is the start time of its first version, namely $V_{2.1}$, which emanated from $V_{1.5}$ by applying some SMOs.

One approach for schema sharing is *full-mark* which adds new (B_2:ts, B_2:te) interval to all corresponding tables and their columns explicitly for the new branch. While this is better than copying all tables and columns, it still requires update work, especially when there are many current tables and columns. To archive better efficiency, we develop a *lazy-mark* approach, which adds a new (B_2:ts, B_2:te) interval to the db node only, and leaves all shared tables and columns unchanged. If the c-branching partially updated the parent schema, besides adding a validity interval on the db node, the lazy-mark approach updates only the modified tables and columns (based on the corresponding table-level and column-level SMOs).

Therefore, the lazy-mark approach can be summarized as: For each update the path to the corresponding level (db, table or column) is visited and the related nodes are updated. Later on, SMOs can update the BMV-Document within a branch as well, and we re-mark those lazy-marked nodes. As a result, the complexity of each schema update for the lazy-mark sharing strategy remains constant per SMO.

Schema Querying. While using schema sharing and lazy-mark to save updating time and storage space, the BMV-Document can still provide efficient access to all branched schema versions. A typical schema query is: "show the schema version at time t for branch B_i". This implies finding the valid tables, as well as their columns, at time t for branch B_i. The procedure of checking whether a table is valid at a given time is shown in

Algorithm 1. The interesting case is if table node T does not have a validity interval for B_i; the algorithm should then check whether this table is shared from one of B_i's ancestor branches through lazy marking (line 7-16). For example, consider the case when branch B_2 is created at time T_6 by adding a status attribute in **empbio** table (Figure 24(b)). Due to lazy-marking, the table **empbio** has only the B_1 branch id in its interval. However, when we check it for branch B_2, following Algorithm 5.1, we determine that it has been inherited from B_1 and shared by B_2 at time T_6.

Algorithm 5.1: CheckTable (T, t, B_i)

Check whether table node T is valid at time t for branch B_i, where t is later than B_i's start time.

```
1   if T has a validity interval for $B_i$ then
2       if $B_i$:ts = null then return false;
3       else
4           if $B_i$:ts <= t < $B_i$:te then return true;
5           else return false;
6   else
7       $B_h$ = $B_i$'s parent; $B_g$ = $B_i$;
8       while ($B_h$ != null)
9           if T has a validity interval for $B_h$ then
10              if $B_h$:ts = null then return false;
11              else
12                  tt = $B_g$'s start time;
13                  if $B_h$:ts<tt<$B_h$:te then return true;
14                  else return false;
15          $B_g$ = $B_h$; $B_h$ = $B_g$'s parent;
16      end while
```

5.4.2 BC-Tables

While the BMV-Document maintains the branched schema versions, the BC-Tables are used to store the underlying evolving data changes. Like H-Table [56], each BC-Table stores the (history of) values for a certain attribute of a base relation. A BC-Table starts from a particular time and may span over multiple schema versions. However, there are considerable improvements: (i) a BC-Table can be shared by multiple branches; (ii) each data record carries only the start time of its time interval; (iii) suitable branched temporal indexing methods are built on top of BC-Tables.

For indexing a BC-Table we facilitate the branched temporal index ([34, 49]) which is a directed acyclic graph over data and index pages. Data pages (which are at the leaf level) contain temporal data, while index pages contain the searching information to lower level pages. In data pages, due to data sharing, a compact data representation <key, data, ts> is used, where ts corresponds to the record's start time (which will be a bid:time in our BC-Tables) of the original record. In an index page, an entry referencing a child page C is of the form <KR(C), TI(C), address(C)>, where KR is the key-range of the child page, and TI is a list of temporal interval(s) for the shared multiple branches of C.

Splitting occurs when a page becomes full. However, unlike in B+-tree page splitting, when a temporal split happens, the data records currently valid are copied to a new page. Thus data records are in both the old page and the new page. The motivation for copying valid data from the full page is to make the temporal query efficient. Splits (temporal-split, key-split, and consolidation) cluster data in pages so that when a data page is accessed, a large fraction of its data records will satisfy the query.

Index page splits and consolidations are similar to those of data pages. Since in index page temporal splits, children entries can be copied, this creates multiple parents for these children. As a result, the branched-temporal index is a DAG, not a tree [34].

When the search for a given key *k*, branch B_i and time *t*, is directed to a particular data page P through the index page(s), the algorithm checks all the records in P with key *k*, and finds the record with the largest start time *ts*, such that *ts* <= B_i:*t*.

Nevertheless, page P may have been shared by branch B_i, in which case some of its B_i related entries may not contain the B_i interval. Those entries are inherited from B_i's ancestor branches. Therefore, we need to extend the search algorithm of the branched-temporal index [34, 49]. In particular, we extend the meaning of the "<" comparison when comparing bid:time tokens. Given two tokens $B_i:T_i$ and $B_j:T_j$ the comparison $B_i:T_i < B_j:T_j$ is satisfied whether ($B_i=B_j \wedge T_i<T_j$) or ($B_i:T_i$ < Par(B_j):Ts(B_j)"), where Par(B_j) is the parent branch of B_j in the version tree, and Ts(B_j) is the start time of B_j.

For example, assume that a data page is shared by branch B_1 and B_2, having entries: <a, v_1, $B_1:t_1$>, <b, v_2, $B_1:t_2$>, <c, v_3, $B_1:t_3$>, <b, v_4, $B_2:t_{14}$>, <c, v_5, $B_1:t_{15}$>, and let branch B_2 be created from B_1 at time t_{10}. So the valid data entries for B_1 at time t_{15} are <a, v_1, $B_1:t_1$>, <b, v_2, $B_1:t_2$>, <c, v_5, $B_1:t_5$>; while the valid data entries for B_2 at time t_{15} are <a, v_1, $B_1:t_1$>, <b, v_4, $B_2:t_{14}$>, <c, v_3, $B_1:t_3$>.

5.5 Query Processing

Data queries are temporal queries on the data records (stored in the BC-Tables and indexed by the branched temporal index). As with traditional temporal queries [53], a user may ask for: (i) a *snapshot* query, or (ii) a time *interval* query. In a linear schema evolution, snapshot or interval queries deal with a single branch. In a branched schema evolution, the following *multiple*-branch queries (first introduced in [37]) are also of interest: (i) *vertical* query and (ii) *horizontal* query. We first discuss how to process temporal snapshot and interval queries within one branch, and then proceed to vertical and horizontal queries over multiple branches.

5.5.1 Queries within a Single Branch

In this case, the temporal constraint (time snapshot or interval) falls within the lifetime of branch Bi. For a snapshot query, the target schema version that stores the queried data is unique and can be identified easily (from the BMV-Document). The corresponding BC-Tables are then accessed through their branched temporal indices.

Processing a time interval query is more complicated because of two challenges: (i) the time interval may have multiple target schema versions (thus even for a single attribute, multiple BC-Tables may be accessed); (ii) in one BC-Table, many data pages may intersect with the time interval, so the search algorithm needs to avoid duplications. The first challenge also appeared in PRIMA [40]: the original temporal query should be reformulated by query rewriting into different sub temporal interval queries for each related BC-Table and the final results are merged from those BC-Tables.

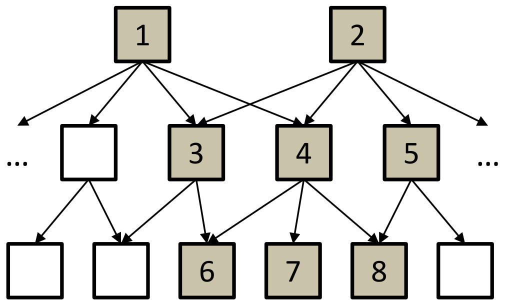


Figure 27: Visited pages

For the second challenge, even in one BC-Table with branched temporal indexing, the naïve depth-first traversal strategy leads to two problems: first, the response set can contain duplicates (due to page splitting copies); second, the same directory entry can be accessed more than once while a query is evaluated. This effect is illustrated in Figure 27 where the gray-colored rectangles display the pages of the branched temporal index visited for a time-interval query. The naïve algorithm would visit pages 1, 2, 5 once, pages 3, 4, 7 twice, page 8 thrice and page 6 four times.

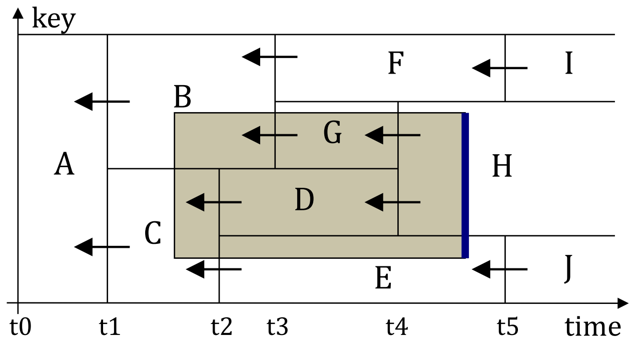


Figure 28: Data pages with links

Traditional duplicate elimination methods such as hashing or sorting may require storage/time overhead, and they are not easy to solve index entry duplication. Therefore, we adopt the $\text{Link}_{\text{based}}$ algorithm proposed in [12] for (linear) multi-version index structures. The BC-Tables' data pages are equipped with external links pointing to their temporal predecessors.

An example is presented in Figure 28 where each page is viewed as the time-key rectangle of the records it contains. A key-range time-interval query (the grey rectangle) intersects pages B, C, D, E, G and H. The $\text{Link}_{\text{based}}$ algorithm consists of two steps. First, the right border of the query rectangle is used to perform a key-range snapshot query. In Figure 28, this snapshot query will access data pages H and E. Second, for each qualifying page obtained in step 1, its temporal predecessor pages are checked to see whether they contain an answer. If they do, the corresponding pages are put into the buffer, answers are reported and the process is repeated. If the left border of the page is already earlier than the left border of the query rectangle, then we do not proceed further. The worst-case performance of $\text{Link}_{\text{Based}}$ is $O(log_B n + a/B + u/B)$ where B is the page capacity, n is the number of records at right-border time t, a is the number of answers, and u denotes the number of updates in the query time period.

5.5.2 Data Queries over Multiple Branches

Vertical Query. The vertical query is an extension of a single branch query, seeking information for a given branch and its ancestors. An example of a vertical query is: "find the data within a key range KR for a given branch B_i and its ancestor branches, at a time stamp t" (or "during a time interval I"). The time stamp t or interval I must be no later than the end time of branch B_i.

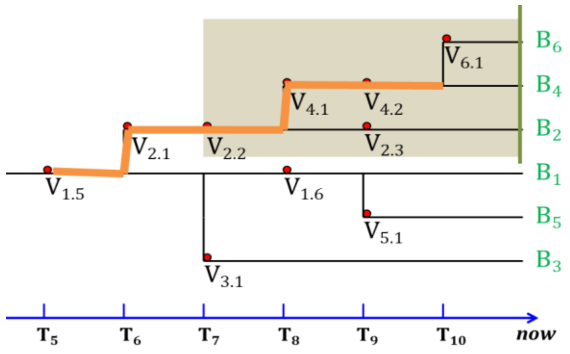


Figure 29: A part of version tree

For a *vertical snapshot query* of branch B_i and at time t, if t is earlier than the start time of B_i, then the result conceptually lies in one of B_i's ancestors B_j, whose lifetime covers time t. For a *vertical interval query*, the time interval may span multiple branches along a path in the version tree. For example, in Figure 29, to find titles of employees within a range KR for branch B_4 and its ancestors in a time interval $[T_5, T_{10})$, we need to access data from branches B_4, B_2 and B_1.

To process a vertical interval query, we first divide the whole query interval *I* for branch Bi into multiple smaller adjacent sub-intervals $\{I_1, I_2,\ldots, I_k\}$, one for each ancestor branch along the path $\{B_{i1}, B_{i2},\ldots, B_{ik}\}$ (where $B_{i1} = B_i$, $B_{i2} = B_i$'s parent and so on). In the above example, querying for B_4 with a time interval $I = [T_5, T_{10})$, *I* should be divided to $[T_8, T_{10})$ for B_4, $[T_6, T_8)$ for B_2 and $[T_5, T_6)$ for B_1 (depicted as the thick lines in Figure 29). Then we process the vertical interval query by answering multiple interval queries for each branch and merge the results together.

However, certain sub-intervals from different branches may be sharing the same BC-Tables, hence a BC-Table could be processed multiple times by different sub-queries. Notice that the sub-intervals are adjacent and the shared data pages are connected by backward links ($Link_{based}$ approach). Therefore, an optimized processing on vertical interval query is to unite the multiple adjacent sub-queries for the same BC-Table into one "super-query". This optimization, called *reunion*, can guarantee that each BC-Table is processed only once for any vertical interval query.

In the above query example, "find the title of employees within a KR for B_4 and its ancestors during $[T_5, T_{10})$", we assume that the **employee_title** table schema is never changed by any branches after it was created at T_5. With the naïve method, we need to process this table three times for three branches with three time intervals as [B_4:T_8, B_4:T_{10}), [B_2:T_6, B_2:T_8) and [B_1:T_5, B_1:T_6). When utilizing the optimized method, the three sub-queries are united into one super-query with an interval [B_1:T_5, B_4:T_{10}).

Horizontal Query. The Horizontal query accesses temporal information for a given branch and its descendants. An example is: "find data within a key range KR for a given branch B_i and its descendants, at time point t" (or during "a time period I"). The time stamp t or interval I must be no earlier than the start time of branch B_i.

A *horizontal snapshot query* can be visualized as a snapshot of multiple relevant branches from a sub-tree of the version tree. For example, the query: "find data for branch B_2 and its descendants at time *now*", corresponds to the vertical dash line in Figure 29, involving branches B_2, B_4 and B_6. To process a horizontal snapshot query on time t, we first determine which descendants of branch B_i (including itself) are valid at t, and then issue multiple vertical snapshot queries, one for each branch.

A *horizontal interval query* can be visualized as a branch-time rectangle on a sub-tree of the version tree. For example, the query: "find data for branch B_2 and its descendants during time interval [T_7, now)", corresponds to the grey rectangle in Figure 29, involving branches B_2, B_4 and B_6. To process a horizontal snapshot query on time t, we again first issue multiple vertical interval queries, one for each descendant branch.

However, this naïve processing method for the horizontal interval query will not be efficient if the multiple vertical interval queries have common parts. In the above example, the vertical interval queries for B_2, B_4 and B_6 during interval [T_7, now) have common parts: [B_2:T_7, B_2:T_8) and [B_4:T_8, B_4:T_{10}), as depicted in Figure 29 by the thick orange line inside the grey rectangle.

As a result, for the multiple vertical interval queries, instead of using the same original query time interval I, we should use different intervals for those descendant branches. For each descendant branch B_j, the new query time interval I_j is the intersection of [ST_j, SE_j) with I, where ST_j and SE_j is the start time and end time of branch B_i. For the above example, the optimized vertical interval queries are: [B_6:T_{10}, B_6:now), [B_4:T_8, B_4:now), and [B_2:T_7, B_2:now). This *rearrange* optimization can improve horizontal interval querying by preventing multiple visits of common parts.

5.6 Merging of Branches

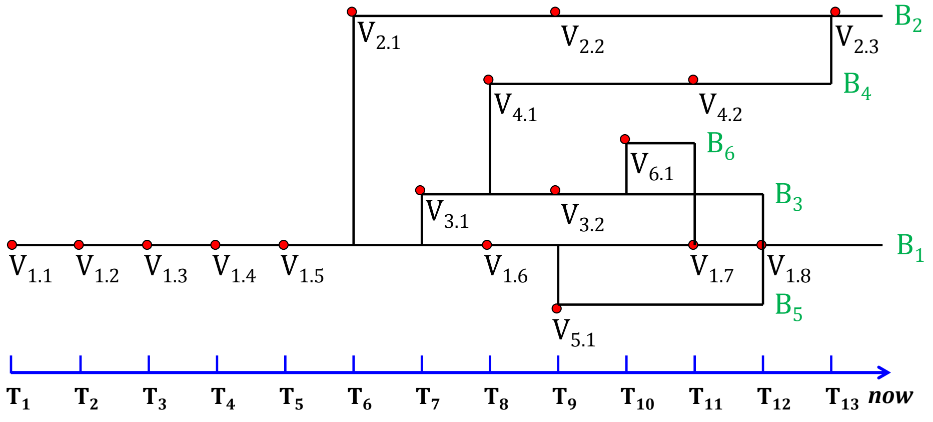


Figure 30: Schema evolution with branching and merging

Since branching is allowed for schema evolution, it is quite natural for us to consider the possibility of merging multiple branches. Branching and merging are two key aspects in many modern environments, such as web-based information systems, collaborative framework, and software development managing tools. Branching provides isolation and parallelism, while merging provides subsequent integration. In this section, we consider how to support current version merging (c-merging).

With c-branching, any currently alive version can create a branch; for a c-merging, the currently alive version of branch B_i can merge to another currently alive version from a different branch B_j by creating a new common schema version. In the example shown in Figure 30, both branching and merging are applied. Such schema evolution will form a *Version Graph* instead of a version tree.

5.6.1 Merging in BMV-Documents

When branch B_i's latest version $B_{i.x}$ merges to branch B_j's latest version $B_{j.y}$ at time t, the branch B_i and version $B_{i.x}$ should be ended and a new version $B_{j.y+1}$ should be created for branch B_i. The branch and version termination can be achieved by updating the end time for corresponding nodes and the lazy-mark process can be utilized for only updating the db and table nodes without reaching to column nodes. After figuring out which elements are discarded from $B_{j.y}$ to $B_{j.y+1}$, and which are added from $B_{i.x}$ to $B_{j.y+1}$, we apply the updates for the corresponding tables and columns. Suitable schema duplication elimination and conflict resolution are applied.

5.6.2 Merging in BC-Tables

When merging is applied in BC-Tables at the data level records, we still can use the same sharing strategy with the branched temporal index but with special extensions. Assume branch B_i merges to B_j at time t. For both branches, some data records have remained while others are removed (especially when there are conflicts). In BC-Tables, we only delete the removed records by adding null values and keep the remained records unchanged, which is consistent with our sharing method in section 5.5. Data duplication elimination and conflict resolution are applied as well.

For data accessing, certain extensions should be implemented for merging, since merging integrates data records from two branches into one. Exploring of a branch's ancestors due to lazy mark is extended from one single path to multiple paths with depth-first or breath-first search along the version graph. Meanwhile, the branched temporal indexing can be adapted for merging with certain modifications.

5.6.3 Query Processing

Here we concentrate on data querying within multiple branches. For vertical queries seeking temporal information for a given branch and its ancestor branches, the ancestors include not only the ones formed by branching but also those by merging. So even for a snapshot querying, the vertical query may need to traverse multiple paths along the version graph by DFS or BFS. For example, assume that in the example of Figure 30, we want to find some records for branch B_1 and its ancestors at time T_{10}. Traversing the version graph backward for B_1 from *now* to T_{10}, we meet two merging points at time T_{12} and T_{11}. Hence the final result unites the response records from not only branch B_1 but also B_5, B_6 and B_3 at time T_{10}.

To process a vertical interval query we access data from multiple parallel paths which may have common parts. The *rearrange* optimization proposed for horizontal querying under branching can be used here. For example, as shown in Figure 30, assume we want to find some data for branch B_1 and its ancestors during time interval $[T_9, T_{13})$. From the version graph, we know that B_3 and B_5 merged to B_1 at time T_{12} and B_6 merged to B_1 at time T_{11}. We can avoid visiting the common paths $[B_1{:}T_{12}, B_1{:}T_{13})$ four times and $[B_1{:}T_{11}, B_1{:}T_{12})$ twice by utilizing *rearrange* to make querying intervals as $[B_1{:}T_9, B_1{:}T_{13})$, $[B_3{:}T_9, B_3{:}T_{12})$, $[B5{:}T_9, B_5{:}T_{12})$, and $[B_6{:}T_9, B_6{:}T_{11})$.

5.7 Experimental Evaluation

To illustrate the efficiency of our framework we present several experiments based on the running example of the employee DB in Figure 25. First, we extend it with more schema versions and branches. The first ten schema changing points (from T_1 to T_{10}) are shown in Fig 2. After that, we make another ten schema changing points (from T_{11} to T_{20}) in two rounds. In each round, there are five schema changes: the first two are linear schema evolutions followed by one schema version branching and two linear schema evolutions. For each linear schema evolution, we choose 50% of the existing branches and make new schema versions for them updating 20% tables and 20% columns in those tables. For each schema branching, we chose all existing branches and make a new branch for each by updating 20% tables and columns. In the end, we have 20 schema changing points with 24 branches of 104 schema versions.

In addition to linear and branched schema evolution, we also create content-level data changes. From T_1 to T20, after each schema changing point, we update the record-level data value 500 times. For each time, we update all existing branches, and for each branch we update 0.2% of all employees for salary, title, and some other randomly chosen attributes. In the end, we have 10,000 time instants of content-level data updates. The Employee DB schema is initialized with 1,000 tables and average 5 columns in each table. We also produce 10,000 employees with 100 titles and other relevant information. For both schema changes and data changes, the tables, attributes and tuples are chosen randomly with a uniform distribution. The page size of our system is 4KB and we set the data page capacity as B = 100 records.

5.7.1 BMV-Documents

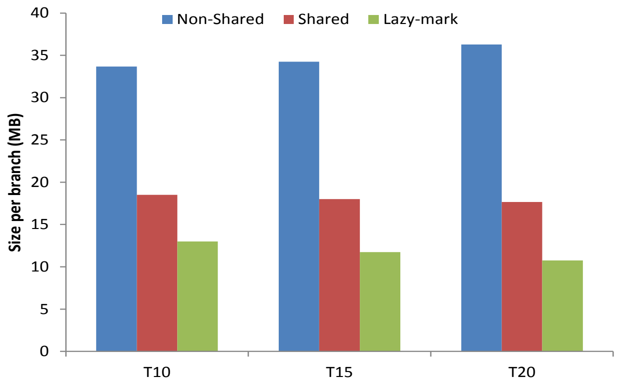


Figure 31: Space saving in BMV-Documents

The sharing strategies among multiple branches and the lazy-mark approach are advantageous in space saving for the BMV-Document without sacrificing querying efficiency. We store the branched schema versions, in XML-based BMV-Documents with three different options when branching occurs: (i) copy the schema without any sharing (Non-Shared); (ii) use the sharing strategy and full-mark approach (Shared); (iii) use the sharing strategy and lazy-mark approach (Lazy-mark). Figure 31 depicts the size per branch (total size / number of versions) of the documents under certain schema changing points: T_{10} (6 branches), T_{15} (12 branches), and T_{20} (24 branches). The options using sharing strategies use much less space than the non-shared option. Compared to the full-mark, the lazy-mark approach is more efficient.

5.7.2 BC-Tables

Space Saving. In addition to the shared BC-Tables (SBT), we use a non-shared method which simply copies alive records from the parent branch when a c-branching happens. The non-shared copying method (NSC) utilizes the MVBT ([9]) to store data in each branch separately, so that each single branch has its own data pages and index structure. The total sizes of data pages and index pages for all tables of all 24 branches are: NSC 71.4 GB and SBT 54.9 GB; clearly, the shared BC-Tables provide significant space saving. Nevertheless, the querying performance of the non-shared method will be better than the fully shared BC-Tables since data has been fully materialized at each branch. Therefore, we consider a trade-off between space and querying performance by applying an enforced copying method (EC), which only allows at most p branches that can be shared in one BC-Table. If a shared BC-Table already reaches this number p, then for a later c-branching, we enforce copying (make a new BC-Table for the newly branch) instead of sharing. The fully shared BC-Tables and non-shared method are two extreme situations for this enforced copying ($p = 1$ corresponds to the non-shared method). In our experiments we implemented an enforced copying method EC with $p = 12$ (EC-12) and $p = 6$ (EC-6).

In order to factor out the query reformulating, we choose one particular BC-Table **employee_title**, whose schema never changes from the beginning and is shared by all branches. To compare space usage of the **employee_title** table by the four methods (NSC, SBT, EC-12 and EC-6), we depict a normalized space usage. Since NSC has the largest storage usage (data pages + index pages), the normalized space is computed by (method$_i$'s space) / (NSC's space). As shown in the Figure 32, the shared **employee_title** BC-Table provides the best space savings followed by EC-12 and EC-6.

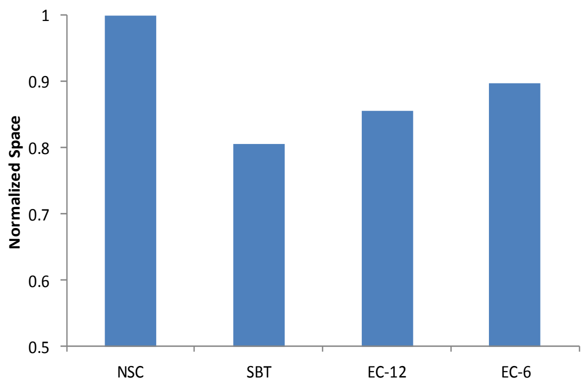


Figure 32: pace saving in employee_title table

Snapshot querying. We use the following query: "find titles of all employees whose ids are within a key range of size 100, for branch B$_i$ at time t" and test on all 24 branches. For each branch, we randomly pick 100 time instants which are in the lifespan of that branch and measure the average snapshot querying time. The average results of all 24 branches are calculated and depicted as normalized page I/O (Figure 33). The SBT method has the largest I/O usage, so the normalized page I/O is computed by (method$_i$'s I/O) / (SBT's I/O). The non-shared copying method has a better snapshot querying performance because data records are stored separately for each branch. However, considering the space saved, shared BC-Tables are performing relatively well on query time. The trade-off methods (EC) gain better querying performance while controlling the space overhead.

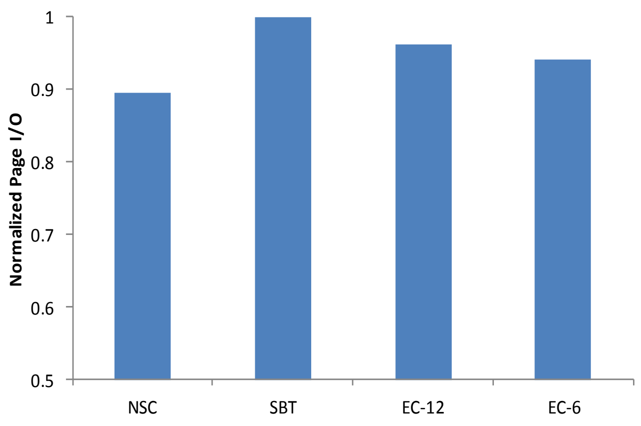


Figure 33: Snapshot Querying

Interval Query Processing. For interval query processing we implement the $Link_{Based}$ algorithm along with the *reunion* and *rearrange* optimizations in shared BC-Tables. First, we test vertical interval queries involving multiple branches: "find titles of employees whose ids are within a key range of size 100 for branch B_{24} and its ancestors in the time interval I". Five different time intervals are used and their coverage rates with respect to the whole temporal data lifetime are 5%, 10%, 20%, 50%, and 100% correspondingly. Two methods are implemented here: one is the basic solution (Basic) which divides the query interval into multiple sub-intervals for each branch. The other is the optimized *reunion* method (Reunion) that unites the sub-intervals into one super-interval if they are sharing the same BC-Table. The I/O ratio of these two methods (Reunion's I/O) / (Basic's I/O) is shown in Figure 34. Clearly the *reunion* optimization can improve the vertical interval querying, and the improvements are more significant when the query interval covers more ancestor branches.

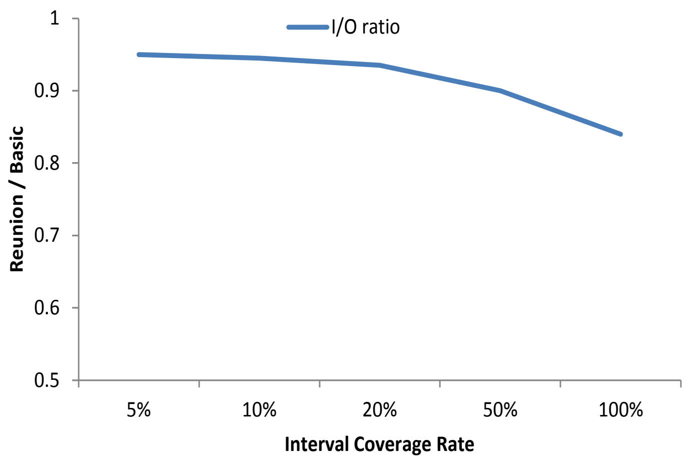


Figure 34: Vertical interval querying

Then we consider horizontal interval queries involving multiple branches: "find titles of employees whose ids are within a key range of size 100 for branch B_1 and its descendants in the time interval I". The different interval I coverage rates are used as same as above. We again implement two methods: one is the basic solution (Basic) that issues multiple vertical queries with the same query interval for each descendant branch, and the other is the optimized *rearrange* method (Rearrange) that arranges different query intervals for each descendant branch to achieve querying efficiency. The I/O ratio of these two methods (Reunion's I/O) / (Basic's I/O) is shown in Figure 35. As seen, the *rearrange* optimization can effectively improve the horizontal interval querying especially when the query interval covers more common parts.

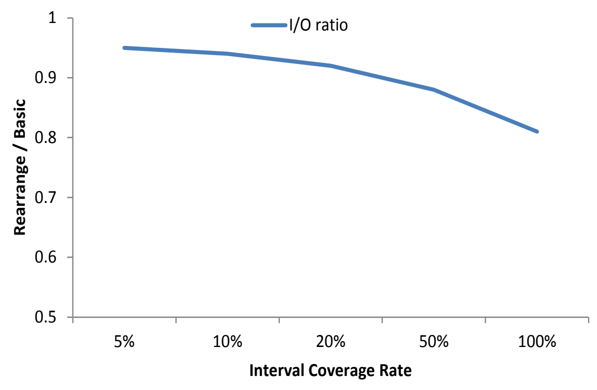


Figure 35: Vertical interval querying

5.7.3 Branched schema evolution with merging

Finally, we employ schema merging into the branched system as well. The branched schema versions and datasets are extended as follows: We randomly insert 5 schema merging points into the 20 schema changing points, and for each such schema merging point, we randomly pick some existed branches to do the merges. A parameter mr (0 ~ 1) is used to control the merging rate. For example, if mr = 50%, we randomly pick half of existed branches to do the merges. The content-level data changes are generated as before: the data is updated 500 times after each schema changing point (evolving, branching and merging). The total number of time instants with data updates is increased from 10,000 to 12,500.

Here we only show results for the horizontal interval querying for branch B_1. We set up five different querying interval coverage rates as same as above with two different merging rates as mr = 50% and mr = 100%. The methods we test include (i) the basic method (Basic) without avoiding the common sub-paths and (ii) the optimized method

(Optimized) with both *reunion* and *rearrange* implemented. The I/O ratio of these two methods (Optimized's I/O) / (Basic's I/O) is shown in Fig 36 for the two mr rates. The optimized method has an advantage in interval querying processing, and this becomes more apparent for larger merging rates and longer query intervals.

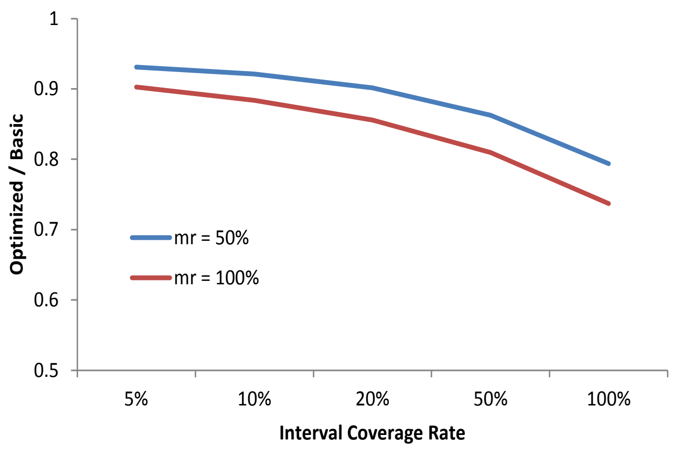


Figure 36: Querying with merging added

5.8 Conclusion

We addressed branched schema evolution for transaction-time databases. To the best of our knowledge, this is the first attempt to examine both data and schema evolution in a branched environment. Efficient schema sharing strategies with smart lazy-mark updates are used. Schema versions are stored in XML-based documents for ease of querying. Data records are stored in relational column tables with branched and temporal indexing. We also explored temporal querying optimizations, especially for vertical and horizontal interval queries. The feasibility of supporting schema merging was also examined. In future research, we will investigate temporal joins and aggregations under schema evolution with branching and merging.

Chapter 6

Conclusions

This dissertation discusses problems related to temporal query processing over social media data and related applications. For evolving graphs in social networks, we proposed efficient algorithms and index structures to process temporal shortest-path queries. For top-k search in social tagging websites, we presented an experimental study by utilizing multiple social networks and temporal information of tagging behaviors. For the temporal top-k query over versioned text collections, we compared previously proposed methods, as well as introduced novel approaches that facilitate multi-version indexing to improve query performance. Meanwhile, we also studied how to archive, manage, and query temporal data over a branched schema evolution.

Evaluating historical queries, such as shortest-path queries, over a temporally evolving graph is an important tool for further analyzing graph properties over time. Based on our newly proposed data model and query definitions, we extended the traditional Dijkstra's algorithm for both time-point and time-interval queries. We investigated how to incorporate index structures such as CH and ALT to speed-up shortest-path query processing. To analyze trade-off and explore further enhancement, we analyzed temporal partition ideas. Finally, the efficiency of our methods and optimizations was shown using real-world social network datasets.

Then we presented a study of top-k search in social tagging websites using three main types of social networks, friendship, common interest networks, and global connections. For weight assignment of each social network component, a user classification method is proposed. To improve the popularity and freshness of ranking results, the timestamps of tagging behaviors are recorded and temporal scoring functions are formed by giving higher weights to more recent time slices. Experimental evaluation on real datasets showed that our framework and methodology work well in practice.

We also 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 based on the score-time view. Experimental evaluation on real-world data showed that the multi-version list method provided the most robust performance considering space usage and query time efficiency for both time-point and time-interval queries.

Last, we addressed branched schema evolution for transaction-time databases. Efficient sharing strategies with lazy-mark updating were implemented. Data records were stored in relational column tables with branched and temporal indexing. Temporal query optimizations were explored for vertical and horizontal queries. The feasibility of supporting schema merging were also analyzed and examined.

Bibliography

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In SEA, pages 230-241, 2011.

[2] S. Amer-Yahia, M. Benedikt, L. Lakshmanan, and J. Stoyanovich. Efficient Network-Aware search in Collaborative Tagging Sites. In VLDB, 2008.

[3] A. Anand, S. Bedathur, K. Berberich, and R. Schenkel. Efficient Temporal Keyword Queries over Versioned Text. In CIKM, 2010.

[4] A. Anand, S. Bedathur, K. Berberich, and R. Schenkel. Temporal Index Sharding for Space-Time Efficiency in Archive Search. In SIGIR, 2011.

[5] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval. Addison-Wesley, 1999.

[6] S. Bao, X. Wu, B. Fei, G. Xue, Z. Su, and Y. Yu. Optimizing Web Search Using Social Annotations. In WWW, 2007.

[7] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In ALENEX, 2007.

[8] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm. In ACM Journal of Experimental Algorithmics, 2010.

[9] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. In VLDB Journal, 1996.

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

[11] K. Berberich, S. Bedathur, and G. Weikum. Efficient Time-Travel on Versioned Text Collections. In BTW, 2007.

[12] J. Bercken, B. Seeger. Query Processing Techniques for Multiversion Access Methods. VLDB 1996.

[13] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In WWW, 1998.

[14] E. Chan and Y. Yang. Shortest Path Tree Computation in Dynamic Graphs. IEEE Transactions on Computers, 58(4):541-557, 2009.

[15] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful Database Schema Evolution. In VLDB, 2008.

[16] C. A. Curino, H. J. Moon, and C. Zaniolo. Managing the history of metadata in support for db archiving and schema evolution. In ECDM, 2008.

[17] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In WEA, pages 52-65, 2007.

[18] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In Algorithmics of Large and Complex Networks, 2009.

[19] E. W. Dijskstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269-271, 1959.

[20] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In EDBT, pages 205-216, 2008.

[21] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In PODS, 2001

[22] R.Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. Master’s thesis, Institut fur Theoretische Informatik Universitat Karlsruhe, 2008.

[23] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchial Routing in Road Networks. In WEA, pages 319-333, 2008.

[24] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In Proc. 16th SCM-SIAM Symposium on Discrete Algorithms, pages 156-165, 2005.

[25] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In IEEE Transactions on System Science and Cybernetics, volumn 4, 1968.

[26] J. He and T. Suel. Faster Temporal Range Queries over Versioned Text. In SIGIR, 2011.

[27] P. Heymann, G. Koutrika, and H. Garcia-Molina. Can Social Bookmarking Improve Web Search? In WSDM, 2008.

[28] M. Hilger, E. Hohler, R. H. Mohring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In Proc. of the 9th DIMACS Implementation Challenge, pages 73-92, 2006.

[29] A. Hotho, R. Jaschke, C. Schmitz, and G. Stumme. Information Retrieval in Folksonomies: Search and Ranking. In the Semantic Web: Research and Applications, 411—426, 2006.

[30] W. Huo and V. J. Tsotras. Temporal Top-*k* Search in Social Tagging Sites Using Multiple Social Networks. In DASFAA, 2010.

[31] W. Huo and V. J. Tsotras. A Comparison of Top-k Temporal Keyword Querying over Versioned Text Collections. In DEXA, 2012.

[32] W. Huo and V. J. Tsotras. Querying Transaction-Time Databases under Branched Schema Evolution. In DEXA, 2012.

[33] K. Jarvelin and J. Kekalainen. IR evaluation methods for retrieving highly relevant documents. In SIGIR, 2000.

[34] L. Jiang, B. Salzberg, D. Lomet, and M. Barrena. The BT-Tree: A branched and temporal access method. In VLDB, 2000.

[35] G. Koloniari, D. Souravlias, and E. Pitoura. On Graph Deltas for Historical Queries. In Workshop on Online Social Systems (WOSS), 2012.

[36] J. A. Konstan. Tutorial: Introduction to Recommender Systems. In SIGMOD, 2008.

[37] G. M. Landau, J. P. Schmidt, and V. J. Tsotras. Historical Queries along Multiple Lines of Time Evolution. In VLDB Jounal, 1995.

[38] C. D. Manning, P. Raghavan, and H. Schutze. Introduction to Information Retrieval. Cambridge University Press, 2008.

[39] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the Flickr social network. In SIGCOMM Workshop on Social Networks (WOSN'08), pages 25-30, 2008.

[40] H. J. Moon, C. A. Curino, A. Deutsch, C.-Y. Hou, and C. Zaniolo. Managing and Querying Transaction-time Databases under Schema Evolution. In VLDB, 2008.

[41] H. J. Moon, C. A. Curino, and C. Zaniolo. Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas. In SIGMOD, 2010.

[42] I. Pohl. Bi-directional search. Machine Intelligence, 6:127-140, 1971.

[43] J. M. Ponte and W. B. Croft. A Language Modeling Approach to Information Retrieval. In SIGIR, 1998.

[44] M. Potamias, F. Bonchi, C. Castillo, and Ar. Gionis. Fast Shortest Path Distance Estimation in Large Networks. In CIKM, 2009.

[45] C. Ren, E. Lo, E. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. In VLDB, 2011.

[46] M. Rice and V. J. Tsotras. Graph Indexing of Road Network for Shortest Path Queries with Label Restrictions. In VLDB, 2011.

[47] S. E. Robertson and S. Walker. Okapi/keenbow at TREC-8. In TREC, 1999.

[48] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In ESA, pages 568-579, 2005.

[49] B. Salzberg, L. Jiang, D. Lomet, M. Barrena, J. Shan, and E. Kanoulas. A Framework for Access Methods for Versioned Data. In EDBT, 2004.

[50] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient Top-*k* Querying over Social-tagging Networks. In SIGIR, 2008.

[51] Y. Song, Z. Zhuang, H. Li, Q. Zhao, J. Li, W. C. Lee, and C. L. Giles. Real-time Automatic Tag Recommendation. In SIGIR, 2008.

[52] V. J. Tsotras and N. Kangelaris. The Snapshot Index: an I/O Optimal Access Method for Snapshot Queries. In Information System, vol.20, no. 3, pp237-260, 1995.

[53] V.J. Tsotras, C.S. Jensen, R.T. Snodgrass. An Extensible Notation for Spatiotemporal Index Queries. In SIGMOD Record 27(1), 1998.

[54] L. H. U, N. Mamoulis, K. Berberich, and S. Bedathur. Durable Top-k Search in Document Archives. In SIGMOD, 2010.

[55] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in Facebook. In SIGCOMM Workshop on Social Networks (WOSN'09), pages 37-42, 2009.

[56] F. Wang, C. Zaniolo, and X. Zhou. Archis: An xml-based approach to transaction-time temporal database systems. In VLDB Journal, 2008.

[57] C. Yu, L. Lakshmanan, and S. Amer-Yahia. It Takes Variety to Make a World: Diversification in Recommender Systems. In EDBT, 2009.