RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases

Shi Gao Jiaqi Gu Carlo Zaniolo
University of California, Los Angeles
{gaoshi, gujiaqi, zaniolo}@cs.ucla.edu

ABSTRACT

Knowledge bases that summarize web information in RDF triples deliver many benefits, including providing access to encyclopedic knowledge via SPARQL queries and end-user interfaces. As the real world evolves, the knowledge base is updated and the evolution history of entities and their properties becomes of great interest to users. Thus, users need query tools of comparable power and usability to explore such evolution histories or flash-back to the past. An integrated system that supports user-friendly queries and efficient query evaluation on the history of knowledge bases is required. In this paper, we introduce (i) SPARQL^T, a temporal extension of SPARQL that expresses powerful structured queries on temporal RDF graphs, (ii) an efficient in-memory query engine that takes advantage of compressed multiversion B+ trees to achieve fast evaluation of SPARQL^T queries, and (iii) a query optimizer that improves selectivity estimation of temporal queries and generates efficient join orders using the statistics of temporal RDF graphs. The performance and scalability of our system are validated by extensive experiments on real world datasets, which shows significant performance improvement comparing with other approaches.

1. INTRODUCTION

Knowledge bases that summarize valuable information in the RDF format are rapidly growing in terms of scale and significance and playing a crucial role in many applications such as semantic search and question answering. The extraordinary success of crowdsourcing and text mining for knowledge discovery makes it easy to generate and update the information in the knowledge bases. In fact, large knowledge bases undergo frequent changes. Table 1 lists the statistics of Wikipedia Infobox edit history, which shows that updates are quite common in many properties: e.g., on average each value in the population property of the city pages is updated more than 7 times. This is not specific to Wikipedia, but also happens in other knowledge repositories.

The management of historical information has emerged as a critical issue for knowledge bases. In fact, timestamping is an important part of the provenance information that is associated with each RDF triple in the knowledge base. The evolution history of knowledge bases captures and describes the change of real world entities and properties, and thus is of great interest to users. However, the size of the history is very large and the schema of knowledge base is also under evolution, which presents challenges in query language, query processing and indexing.

Category	Property	Average Number of Updates
Software	Release	7.27
Player	Club	5.85
Country	GDP(PPP)	11.78
City	Population	7.16

Table 1: Statistics of Wikipedia Infobox Edit History

As the RDF model for representing knowledge bases is gaining great popularity, the importance of managing and querying the evolution history of knowledge bases is also recognized. Gutierrez et al. [17] extended the RDF model with time elements and several approaches [16, 29, 30, 32] have been proposed to support the queries on temporal RDF datasets. Most previous works employ relational databases and RDF engines to store temporal RDF triples and rewrite temporal queries into SQL/SPARQL for evaluation. The languages proposed in these works use an interval-based temporal model which leads to complex expressions for temporal queries, e.g., those requiring joins and coalescing [12, 33]. At the physical level, previous approaches exploit indexes such as tGrin [30] to accelerate the processing of simple temporal queries, but they do not explore the use of general temporal indices and query optimization techniques. This limits their scalability and performance on large knowledge bases and for complex queries.

In this paper, we describe a vertically integrated system RDF-TX (RDF Temporal eXpress) that efficiently supports the data management and query evaluation of large temporal RDF datasets while simplifying the temporal queries for SPARQL programmers and consequently, for end-user interfaces facilitating the expression of the same queries. To support the queries over the evolution history of knowledge bases, we propose efficient storage and index schemes for temporal RDF triples using multiversion B+ tree [7] and implement a query engine which achieves fast query evaluation by taking advantage of comprehensive indices. We also build a query optimizer that generates efficient join orders using a cost-based model and the statistics of temporal RDF graphs.

We propose a general and scalable solution for the problem of managing and querying massive temporal RDF data based on three main contributions:

- We propose SPARQL^T, a temporal extension of the structured query language SPARQL based on a point-based temporal model which simplifies the expression of temporal joins and eliminates the need for temporal coalescing. This approach makes possible end-user interfaces, such as those in [6,

15], where queries are entered via simple by-example conditions in the infoboxes of Wikipedia pages.

- We present an efficient main memory system RDF-TX for managing temporal RDF data and evaluating SPARQL^T queries. Our system uses multiversion B+ tree (MVBT) to store and index temporal RDF triples. An effective delta encoding scheme is introduced to reduce the storage overhead of indices. The algorithms on MVBT are extended and optimized to exploit the characteristics of the compression scheme and query patterns. Experimental evaluation demonstrates superior performance and scalability of RDF-TX compared with other approaches.
- We implement a query optimizer that generates the efficient join orders of SPARQL^T query patterns using the statistics of temporal RDF graphs. To manage temporal statistics, we introduce compressed Multi-Version SB Trees (MVSBT) that provide highly accurate estimation of statistics with a small storage overhead.

The rest of this paper is organized as follows. Section 2 provides an overview of RDF-TX system and temporal RDF model. Then we present the SPARQL^T query language in Section 3. Section 4 describes our storage model and index compression techniques. The query evaluation techniques are discussed in Section 5. Section 6 introduces a query optimizer for join order optimization. We evaluate our system on real world datasets in Section 7, and discuss related work in Section 8. Finally, we conclude in Section 9.

2. OVERVIEW AND DATA MODEL

In this section, we discuss the challenges of supporting temporal queries against the history of knowledge bases and provide a general overview of the RDF-TX system. Then we review the temporal RDF model introduced in [17].

2.1 Overview

The addition of temporal information to the basic RDF model poses difficult challenges that parallel those encountered by researchers working on extending the relational model with temporal information. A first lesson learned from that experience is that supporting temporal event information is simple, but state-based temporal information presents many challenges. In fact, timestamps can be associated with temporal events via standard RDF predicates (e.g. *birthDate* and *establishedYear*). Then they can be queried as any ordered domain.

Supporting state information is much more complex, as demonstrated by the many temporal representations and constructs proposed [12, 13, 25, 33] and the rich set of interval-based operators required [5]. For instance, answering a simple query such as "Who was the president of University of California on 9/9/2009?" requires determining the time interval that contains 9/9/2009. In a valid time temporal database, the curators are responsible for supplying these timestamps, thus creating a valid-time history. However, in DBpedia and other web repositories, the curators do not update timestamps directly: instead they update web pages and associated Infoboxes to reflect the changes that occurred in the domain they describe. Thus, readers of the web page will notice a change of the president name from *Mark Youdof* to *Janet Napolitano*. The date and time of this change, i.e. the timestamp in which the update was executed by the system, is known as transaction time (or system time) and, as such, it is constructed from the system log.

Whereas tardy curators will eventually enter the correct valid time, any tardiness of their actions adds permanently to the imprecision of a transaction time databases. Nevertheless, when as in the case of Wikipedia, valid time histories are not available, transaction time history will provide a reasonable substitute. Indeed, since the varying tardiness with which Wikipedia Infoboxes are refreshed has not damaged their popularity, it is reasonable to expect that databases describing their system time history will be equally popular, particularly when users are given tools to compensate for temporal imprecision[1].

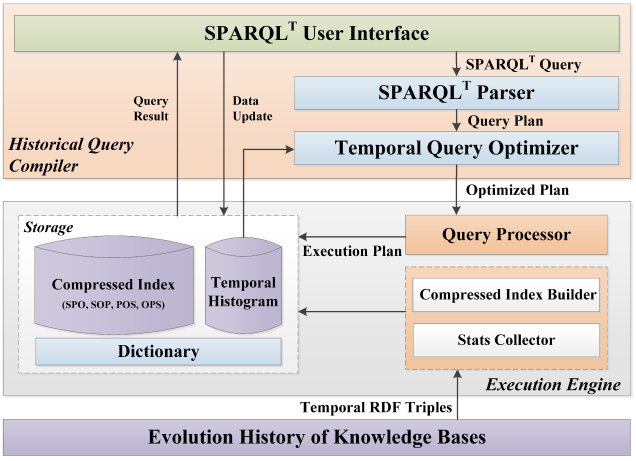


Figure 1: RDF-TX Architecture

Moreover, there are other scenarios in which transaction time histories are needed:

- *History Browsing and Analyzing* [14, 15]. The history of knowledge base captures the revisions of knowledge, which is of great interest for editors and users to understand how the knowledge is evolved and updated.
- *Knowledge Auditing and Verification* [35]. Timestamping, as an important part of provenance, is important for ensuring the quality of facts in many applications. A system with temporal query support also provides administrators with previous states of knowledge bases for auditing purposes.
- *Backup and Recovery*. The history of knowledge bases can be used to backup and recover missing information.

In this paper we present query extensions and an efficient system for managing and querying the transaction time history of knowledge bases. However, the user model and query constructs apply to valid-time histories as well. At the physical level, although the storage structure is designed for the transaction time model, our implementation remains effective for most valid-time histories as discussed in our technical report [2].

Figure 1 shows the high level architecture of our system, which can be divided into two main components as follows.

Historical Query Compiler. To express queries against the history of knowledge bases, we introduce a temporal extension of SPARQL called SPARQL^T. Users can write and submit SPARQL^T queries through our interface. Then the SPARQL^T queries are compiled to query plans represented as graphs of query patterns and passed to the temporal query optimizer. The optimized query plans are submitted to the *Execution Engine* for evaluation.

Execution Engine. In our engine, the historical information is represented as temporal RDF triples and stored using compressed MVBT indices that support fast query processing. The query processor transforms the query plans from compiler to execution plans expressed in query operators (e.g. temporal selection and join) and executes them on the compressed MVBT indices.

[1]Provenance annotations that record the tardiness of refreshes can go a long way to cure this problem.

Predicate	Object	Timestamp
president	Mark Yudof	06/16/2008 ... 09/29/2013
	Janet Napolitano	09/30/2013 ... *now*
endowment (billions)	10.3	07/01/2013 ... 06/30/2014
	13.1	07/01/2014 ... *now*
undergraduate	184,562	05/14/2013 ... 01/29/2015
	188,300	01/30/2015 ... *now*
staff	18,896	08/29/2013 ... 01/29/2015
	19,700	01/30/2015 ... *now*
budget (billions)	22.7	01/30/2013 ... 01/29/2015
	25.46	01/30/2015 ... *now*

Table 2: Temporal RDF Triples for *University of California*

2.2 Data Model

Knowledge bases such as DBpedia [10] and Yago2 [18] can be represented as RDF graphs which consist of RDF triples in the format (*subject*, *predicate*, *object*). The subject and predicate of an RDF triple are elements from the set of Uniform Resource Identifiers $\mathcal{U}$, while the object is a URI from $\mathcal{U}$ or a value from the set of literals $\mathcal{L}$. For example, the RDF triple for "The president of University of California is Mark Yudof." is:

- *subject*: http://www.w3.org/edu/University_of_California
- *predicate*: http://www.w3.org/elements/president
- *object*: http://www.w3.org/people/Mark_Yudof

For the sake of simplicity, we do not discuss the concept of blank nodes and assume the prefix parts of URI (e.g. http://www.w3.org/edu/) are given. Above RDF triple is represented as (*University of California*, *president*, *Mark Yudof*).

Since the basic RDF model is designed for static information, we represent the evolution history of knowledge bases using the temporal RDF model proposed in [17] that extends the RDF model with temporal elements. Each RDF triple is annotated with a temporal element to represent the time when this triple is valid. Formally, given a point-based temporal domain $\mathcal{T}$, a *Temporal RDF Graph* consists of a set of temporal RDF triples where each temporal RDF triple is a RDF triple (s, p, o) annotated with a temporal element $t \in \mathcal{T}$. A set of temporal RDF triples with consecutive time points $\{(s, p, o)\ [t] \mid t_s \leq t \leq t_e\}$ are encoded using the interval-based expression as: $(s, p, o)\ [t_s \ldots t_e]$.

The evolution history of subject *University of California* is represented as a set of temporal RDF triples, as shown in Table 2. All the triples share the same subject *University of California*. We use DAY as the granularity of time and *now* as the current time. Typically, one triple is valid over several days, which we represent with ... between the start day and the end day (start and end included): e.g. [07/01/2013 ... 06/30/2014] represents all the days between 07/01/2013 and 06/30/2014.

3. SPARQL$^{\mathrm{T}}$ QUERY LANGUAGE

To support temporal queries over the history of knowledge bases, we propose a temporal extension of SPARQL called SPARQL$^{\mathrm{T}}$. One main difference between SPARQL$^{\mathrm{T}}$ and previous works is the choice of the temporal model. Many existing works [29, 30, 32] use the interval-based model because of efficiency considerations. However, to express temporal queries, the interval-based representation requires additional operators such as temporal interval overlap, intersect and coalesce, which introduce complications and difficulties [12, 42], particularly for casual users working with friendly wysiwyg interfaces. Therefore, we use a point-based temporal model that resolves these problems at the logical level; however at the physical level we retain the interval representation for efficiency reasons. Queries on the point-based model can be easily mapped into equivalent ones on the interval-based model for execution.

3.1 SPARQL$^{\mathrm{T}}$ Syntax

SPARQL$^{\mathrm{T}}$ extends SPARQL with temporal patterns and constructs to query temporal RDF data. To simplify our presentation, we first review the standard syntax of SPARQL [28] and then explain our temporal extension.

SPARQL graph patterns are defined as follows:

- A SPARQL graph pattern is a triple $\{s\ p\ o\}$ from $(\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})$ where $\mathcal{V}$ is a set of variables.
- If S and S' are SPARQL graph patterns and F is a filter clause, (S AND S'), (S OPT S'), (S UNION S') and (S FILTER F) are also graph patterns.

where (S AND S'), (S OPT S') and (S UNION S') denote the conjunction, optional and union graph patterns.

Temporal queries against the history of knowledge bases are expressed as SPARQL$^{\mathrm{T}}$ graph patterns.

SPARQL$^{\mathrm{T}}$ graph patterns are defined as follows:

- A SPARQL$^{\mathrm{T}}$ graph pattern is a tuple $\{s\ p\ o\ t\}$ from $(\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$.
- If P and P' are two SPARQL$^{\mathrm{T}}$ graph patterns and F' is a filter clause, (P AND P') and (P FILTER F') are also SPARQL$^{\mathrm{T}}$ graph patterns.

Where F' is constructed using the elements from $\mathcal{U} \cup \mathcal{L} \cup \mathcal{T} \cup \mathcal{V}$, comparison symbols, logical connectors and the temporal built-in functions discussed next. In SPARQL, there are 8 types of graph patterns as: S, P, O, SP, SO, PO, SPO, and full scan. For example, SP refers to a query pattern in which subject and predicate are constants and object is a variable. SPARQL$^{\mathrm{T}}$ supports 16 types of graph patterns, which enable the expression of many interesting queries over temporal RDF graphs.

(P UNION P') and (P OPT P') are not supported in current SPARQL$^{\mathrm{T}}$, and their implementation is planned for the future. In current state, SPARQL$^{\mathrm{T}}$ supports efficiently all the queries described in this paper, including the applications discussed in Section 2.1. In passing we observe that they follow the patterns of (i) retrieving information from a previous version of knowledge bases, and (ii) joining the information with similar keys and timestamps. These two scenarios correspond to two operators: single graph pattern matching and temporal join that are supported very efficiently in our system.

Time Representation and Functions. In SPARQL$^{\mathrm{T}}$, timestamp is from a discrete time domain with a minimum unit as chronon [13]. We define two temporal types: *dateTime* and *period*. *dateTime* corresponds to a single timestamp. *period* corresponds to a set of consecutive timestamps, represented as a pair of two datetime points. For timestamps, SPARQL$^{\mathrm{T}}$ is equipped with YEAR/MONTH/DAY functions to enable flexible temporal conditions. For periods, we define two built-in functions *TSTART* and *TEND* to return the first and last element in a set of consecutive timestamps.

Many temporal queries involve the reasoning of duration. Thus we define a built-in function *LENGTH* that counts the number of time units (we use DAY as the minimum unit in this paper) within the same consecutive period of time. If one fact is associated with multiple intervals, we return the length of max duration. Another similar function *TOTAL_LENGTH* is defined to compute the total length of all intervals.

3.2 Semantics and Examples

SPARQL$^{\mathrm{T}}$ is a graph matching language for querying temporal RDF data. The input of a SPARQL$^{\mathrm{T}}$ query is a temporal RDF

graph and the output is a set of mappings that replace the variables with values from the input temporal RDF graph. The operators of SPARQL are extended to manipulate the temporal element t of SPARQL^T graph patterns. For single pattern matching, the query result is the set of temporal RDF triples that match the graph pattern and the filter clause. If there are two or more patterns in the query, the results of single pattern matching are joined. Due to space limitations, we omit the discussion of formal semantics which can be instead found in our technical report [2]. All the syntax discussed in previous section are implemented in RDF-TX . Next we illustrate the usage of SPARQL^T via several examples.

Temporal Selection. We first discuss *temporal selection* queries that have one query pattern (a four-element tuple {*s, p, o, t*}). An example of temporal selection query is the "when" query that retrieves the valid timestamps of given facts. Users only need to specify the values for (*s, p, o*) and a variable for the temporal element.

EXAMPLE 1. When did Janet Napolitano serve as the president of University of California.

SELECT ?t
{University_of_California president Janet_Napolitano ?t}

In the query result, the timestamps will be displayed in the compact format $[t_s \ldots t_e]$. Running Example 1 against the temporal RDF graph in Table 2 returns [09/30/2013 ... *now*]. Another common type of temporal selection queries retrieves information from a previous version of the knowledge base. The temporal constraints (e.g. within a period) can be specified in the FILTER clause.

EXAMPLE 2. Find the budget of University of California in 2013.

SELECT ?budget
{University_of_California budget ?budget ?t .
FILTER(YEAR(?t) = 2013) }

EXAMPLE 3. Find each person who served as the president of University of California for more than one year before 2010.

SELECT ?person ?t
{ University_of_California president ?person ?t .
FILTER(YEAR(?t) <= 2010 && LENGTH(?t) > 365 DAY)}

Temporal Join. More complex queries often use temporal joins which, in SPARQL^T, are expressed by multiple query patterns that share the same temporal element. General temporal join may involve both key and temporal dimensions.

EXAMPLE 4. Find the name of the university in which Mark Yudof served as the president and the number of undergraduate students when he was in office.

SELECT ?university ?number ?t
{?university undergraduate ?number ?t .
?university president Mark_Yudof ?t . }

Queries using multiple temporal joins are rather simple to express in SPARQL^T, whereas in languages based on an interval-based temporal model, such queries tend to be much more complex. For example, if users want to search the number of undergraduate and graduate students when Mark Yuodf was in office, we only need to add one more query pattern: {*?university graduate ?number2 ?t*} to Example 4. If we switch to interval-based model, the query will consist of three query patterns and three temporal conditions: $?I_1$ *overlap* $?I_2$, $?I_1$ *overlap* $?I_3$, $?I_2$ *overlap* $?I_3$ where $?I_1$, $?I_2$, $?I_3$ are three variables for intervals.

Besides temporal join, the point-based query patterns also support the expression of other temporal operations such as MEET and CONTAIN using the built-in functions TSTART and TEND.

EXAMPLE 5. Find who succeeded Mark Yudof as the president of University of California.

SELECT ?successor
{ University_of_California president Mark_Yudof ?t_1 .
University_of_California president ?successor ?t_2 .
FILTER(TEND(?t_1) = TSTART(?t_2)) . }

4. STORAGE AND INDEXING

Since the performance of query engines is heavily influenced by its use of indices, it is important to choose an appropriate index structure as well as storage schema for the temporal RDF data.

A natural approach followed by previous works [29] consists in managing temporal RDF triples using existing RDBMS. However, for searching both RDF information and temporal information, two sets of indices are required, and this results in significant costs in storage and retrieval time, which are shown in Section 7. A second natural approach will be using RDF engines, such as Jena and Virtuoso, which have seen recent improvements in performance and functionality. However, this requires the standard RDF reification approach in which a temporal RDF triple is represented as an entity instance with five properties: subject, predicate, object, start time and end time. Thus we need to use five triples for each temporal fact, whereby the space cost increases along with complexity of the queries and time required to optimize and execute them.

Therefore, rather than modifying and extending existing systems we design and build a new system that integrates advanced indexing and data compression techniques into an architecture conceived for efficient support of SPARQL^T queries

4.1 Index Scheme

Many index structures [7, 19, 22, 24] have been proposed for temporal data. Each index has its own strength and they have shared issues such as space overhead and limited support for general temporal queries.[2] In this project, we employ Multiversion B+ Tree (MVBT) to index temporal RDF data for the following reasons. First, MVBT is a bi-dimensional index with asymptotic worst-case guarantee and delivers good performance in real world datasets. Second, we propose an effective approach to compress MVBT which greatly reduces the space cost. The algorithms [8, 41] are extended and optimized on compressed MVBT to enable fast index scan and join. Next we briefly review the structure of MVBT and discuss the index scheme in RDF-TX system.

4.1.1 MVBT

Multiversion B+ Tree [7] is a temporal index structure with optimal worst case guarantees for data insert, update, and delete. The complexity of a temporal query in version i is asymptotically equal to the complexity of the query on a B+ tree that maintains all the data valid in version i. Rather than a single tree, an MVBT is actually a forest of trees. It has multiple root nodes and each of them corresponds to a temporal partition of data, as shown in Figure 2(a). An entry in the MVBT node can be represented as (*key*, *start version*, *end version*, *data value/pointer*) where *key* is unique for a given version and *start version* and *end version* together denote the live period of data. An entry that stores data inserted in version i carries a period of (i, *now*). We denote an entry with end version *now* as a live entry. The delete operation modifies the end version of a live entry.

A simple example of MVBT insertion/deletion is shown in Figure 2(b). We first insert five values into an empty MVBT in Version 1, which results in an MVBT tree (i). Then we insert key 14 in Version 2 and delete key 46 in Version 3. The MVBT index becomes

[2]A detailed discussion of temporal index can be found in Section 8.

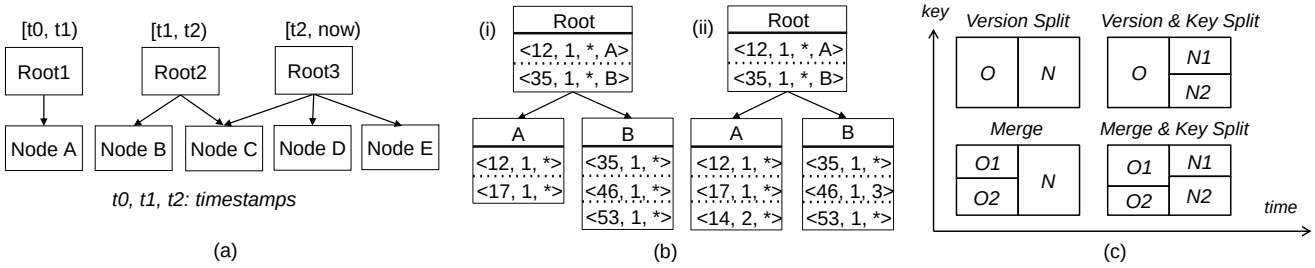


Figure 2: MVBT Example (a) Index Structure; (b) Data Insertion and Deletion; (c) Node Structure Changes.

(ii) with <14, 2, *> added to Node A and <46, 1, *> changed to <46, 1, 3> in Node B.

To guarantee the performance, MVBT has *weak version condition* that there must be at least k live entries in a live node. When there are too many entries or not enough live entries (*weak version underflow*) in an MVBT node, node structure changes (version split or merge) are triggered. After structure changes, the number of live entries in a node should be in the range $[k_l, k_h]$ (*strong version condition*), which prevents long sequences of split and merge operations.

There are four types of node structure changes in MVBT, illustrated in Figure 2(c). We assume that an insertion results in too many entries in a node *O*. Then a *Version Split* is performed that copies all the live entries in *O* to a new node *N*. If node *N* has more than k_h entries, then an additional key split is performed that splits *N* into two nodes, as shown in *Version & Key Split*. If *N* has less than k_l entries, *Merge* is triggered.

Assume we need to perform *Merge* operation on the node *O1*. MVBT identifies a live sibling node *O2*, performs version split and copies the live entries into the new node *N*. If node *N* has more than k_h live entries, a key split is performed immediately, as shown in *Merge & Key Split*. Due to the space limitation, we address the readers to [7] for more details of MVBT node structure changes.

4.1.2 Indexing Temporal RDF

In RDF-TX all the data and indices are stored in the main memory. We implement in-memory MVBT to index the temporal RDF triples. The insertion of an interval-encoded RDF triple {(s, p, o) [t_s, t_e]} on MVBT index M is decomposed into two operations : (i) insert data item (s, p, o) into M at time t_s; (ii) delete data item (s, p, o) at time t_e.

Since the variable may be located in any position of (s, p, o), we create four MVBT indices (SPO, SOP, POS, OPS) for different orders of keys (s, p, o). These MVBT indices cover all 16 SPARQL$^{\mathrm{T}}$ graph patterns. For example, the MVBT index for temporal RDF triples in POS order can cover four patterns: P, PT, PO, POT. In query evaluation, the query engine parses the SPARQL$^{\mathrm{T}}$ prefix patterns to identify the corresponding MVBT index.

We employ dictionary encoding in the index construction, which reduces the index size and avoids the slow comparison between long string literals. Thus RDF-TX replaces the literals with dictionary IDs, and the triples that consist of IDs and timestamps are inserted into our indices. The mapping relations are maintained in our in-memory dictionary for index update and query evaluation. Since the main space cost in our indices is the large number of MVBT entries, dictionary encoding only reduces space cost by 10% - 20%. After dictionary encoding, we exploit delta compression which significantly reduces the space cost of MVBT indices.

4.2 Index Compression

For the Wikipedia Infobox History, the size of one standard MVBT index implemented in Java is 1.5–2.2 times of the raw data. Moreover, a temporal RDF graph requires four MVBT indices. If a naive approach is used, comprehensive indexing of temporal RDF data becomes prohibitively expensive. Thus effective compression techniques are needed for large scale datasets.

We observe two characteristics of MVBT. First, the entries in the MVBT node are sorted and neighboring entries often share the same prefix, which could be utilized to reduce space cost. Second, all the node structure operations start from *version split*. This guarantees the query performance but leads to a lot of long intervals. Given these characteristics, we introduce an effective delta encoding method to compress MVBT indices.

4.2.1 Compression Techniques

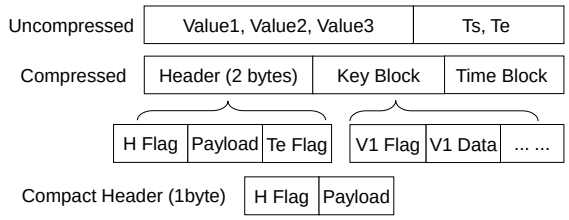
We design a compression scheme for variable delta encoding of MVBT entry. An MVBT entry for temporal RDF data consists of five values: (v_1, v_2, v_3, t_s, t_e) where v_1, v_2, v_3 are elements in RDF triples. We store the minimum values for keys and timestamps in each node as base values. Since the data entries are sorted by start version (t_s) and key, most entries have very close start versions. Therefore for t_s, we only keep the minimal value of each node, and compute and store the delta start versions. For t_e, the compression rules are as follows: (i) if the valid interval (t_s, t_e) is a short interval, t_e is stored as the length of intervals; (ii) if the valid interval is long, t_e is stored as the delta value between t_e and minimum t_e in the node; (iii) if the valid interval is a live interval (t_e is *now*), a special flag is set and t_e is stored as empty. Other values (v_1, v_2, v_3) are compressed as the delta values (i) between current value and the value in neighbor entry or (ii) between current value and minimum value in leaf node.

The compressed values are stored in a compact byte array. Figure 3(a) illustrates the format of compressed MVBT entry. Every entry consists of three parts: header, key block (v_1, v_2, and v_3), and time block (t_s and t_e). A normal header (2 bytes) contains a flag (H Flag, 1 bit) for header type (normal/compact), a payload (13 bits in total, 7 bits for key block and 6 bits for time block) that stores the number of bytes for each delta value, and the t_e flag (2 bits) that records the compression rule for t_e. For the delta values in key block, we use 1 bit to record how the delta is computed (with neighbor or with node minimum value).

We observe that in large datasets, it is very common that two neighboring MVBT entries (i) share at least one element in key block; (ii) have very close t_s (delta size ≤ 4 bytes) (iii) both t_e are *now*. Thus for these entries, we propose a compact header which consists of 1 bit header type and 7 bits payload (for two delta values in key block and t_s delta value).

There is a trade-off between the compression ratio and query performance. Since the number of index nodes is much smaller than the number of leaf nodes and the index nodes are accessed more frequently than leaf nodes, we only compress the leaf nodes of MVBT indices. As shown in evaluation (Section 7), the size of compressed MVBT is about 24% of standard MVBT.

We build MVBT indices for different subsets of Wikipedia dataset



(a)

# of triples (million)	5	10	15	20	25	30
Time (seconds)	1.36	2.65	3.87	4.81	6.18	7.25

(b)

Figure 3: (a) Compressed MVBT Entry (b) Compression Time

and then test the time for compressing MVBT entries, as shown in Figure 3(b). The result shows that it takes very little time to apply the delta encoding technique. Given an MVBT index built from 30 million temporal RDF triples, the time for compressing all the leaf nodes is only 7.25 seconds.

4.2.2 Index Maintenance

An important principle of index compression is to reduce the storage overhead while maintaining the performance of index update and search. For data insertion, we first look up index nodes and identify the leaf node to be updated. In the leaf node, we decompress the start versions (t_s) to find the position of input start version i and compute the delta values of input data. Then we modify the $(i + 1)$th entry if its delta values are changed. One issue is that we need to scan from the beginning of all entries. To address this issue, we add a checkpoint in each node that stores the position of MVBT entry with largest t_s. Then in data insertion, since the t_s of input data must be larger than existing t_s, we only decompress the entries after checkpoint. Deletion in MVBT only updates the end version of a live entry. Thus we simply scan all the entries and modify the t_e of the matched entry. As shown in Section 7, insertion/deletion on compressed MVBT only takes 5% more time than standard MVBT.

5. QUERY PROCESSING

In this section, we present the design and implementation of RDF-TX query engine, which makes use of MVBT to process the temporal operations of the language.

5.1 Compiling SPARQL$^{\mathrm{T}}$ Query

The overall evaluation of SPARQL$^{\mathrm{T}}$ queries consists of four steps:

- Parse the input query and translate point-based query patterns to interval-based query patterns.
- Construct a query plan. The plan is represented as a graph in which each node is an interval-based query pattern.
- When the query contains multiple temporal joins, optimize the query plan to improve the join order.
- Translate the query plan to an execution plan that is evaluated on compressed MVBT indices.

Next we elaborate each step with more details.

Translating Query Patterns. Since the temporal RDF graph is stored as interval-based temporal RDF triples, we translate the point-based SPARQL$^{\mathrm{T}}$ query patterns to the interval-based patterns that can be converted to range queries and executed on MVBT. For key elements, we take the literals as prefix and convert the unknown parts to key ranges. For temporal element (t), if there exist temporal constraints in the FILTER clause, we generate time ranges based on the constraints; otherwise, the default range is [*0*, *now*] where *0* refers to the minimum time point. Consider the query pattern {*University of California budget ?budget ?t*} (*YEAR(?t) = 2013*) in Example 2. The interval-based query pattern can be described as a query region with key range and time range as follows:

- key range: (*University of California*, *budget*, _) – (*University of California*, *budget*, ∞)
- time range: *01/01/2013 – 12/31/2013*

Here _ and ∞ denote the extrema of the string domain.

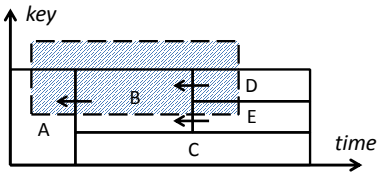


Figure 4: An Example of MVBT Backward Link

Constructing and Optimizing Query Plan. The query engine generates a query plan that consists of interval-based query patterns from the first step. This query plan can be represented as a graph in which the edges between the nodes are added when two query patterns share the same variable. If there are multiple join operations, the query optimizer (discussed in Section 6) is called to find efficient query plans using the statistics of temporal RDF graphs.

Executing the query plan on MVBT. Lastly, the optimized plan is translated to an execution plan which is similar to the query plan in relational databases. Every query pattern is converted to an index scan operator on MVBT indices. Then the join operators are added based on the optimized join order. Finally, appropriate filter operators are added using the FILTER clause of SPARQL.

5.2 Executing Query Plan

Next we describe the implementation of index scan and temporal join in RDF-TX . Other operators (e.g. filter) are implemented similar to their counterparts of existing engines thus omitted.

5.2.1 Index Scan

We perform an index scan for each interval-based query pattern. For the index scan on MVBT, we employ the link-based range-interval algorithm [8] which introduces *Backward Link* in MVBT to process the range queries. The MVBT leaf nodes are equipped with backward links that point to the temporal predecessors. The index scan is performed as: (i) search all the nodes that intersect the right border of query region; (ii) follow the backward links of the nodes to find all the nodes that intersect query region; (iii) scan the leaf nodes found in the first two steps to retrieve the entries. An example of linked index scan is shown in Figure 4. The shadowed rectangle represents a query. MVBT nodes D and E are first visited. Then as the predecessor of D and E, node B is checked. Lastly, node A is visited.

5.2.2 Temporal Join

Temporal join represents one of the most expensive operations in the temporal query language, especially when the size of knowledge base is very large. Therefore we explore three types of joins: *Merge Join*, *Hash Join*, and *Synchronized Join*.

Merge join is very popular and widely used in existing SPARQL engines [27, 38]. These systems build indices for all permutations so that the optimizer leverages the indices to perform order-preserving merge joins. However, this does not work for MVBT index since the entries are sorted by time. We mainly use *Hash Join* in our query engine.

When the size of result is large, the cost of building a hash table may be very expensive. Thus we extend the synchronized join [41]. The basic idea of synchronized join is as follows: (i) synchronously find the set of all MVBT node pairs (e_1, e_2) that intersect each other and the right border of query region; (ii) join e_1 and e_2; (iii) join the predecessors of e_1 and e_2 by following the backward links. This algorithm avoids materializing the intermediate result, but it is much slower than hash-based join since one page and its predecessors are visited many times. So we optimize this algorithm by caching recently visited records; that is, given a page e from step (i), we cache the records in e and its predecessors, and perform joins between e and other pages. This optimized synchronized join is used when the query pattern in the join accesses a large portion of index (e.g. find all the triples valid in a certain period).

6. OPTIMIZATION

In RDF-TX , improper join orders may generate large intermediate results and slow down execution. Therefore, a natural step is to optimize complex SPARQL^T queries by finding efficient join orders. The key of join optimization is to efficiently estimate the costs of different join orders, which is not a trivial task for temporal queries. In this section, we present a query optimizer that uses estimated statistics of temporal RDF graph to optimize the orders of temporal joins in SPARQL^T queries.

6.1 RDF-TX Query Optimizer

For queries that involve multiple temporal joins, we implement a query optimizer that uses the bottom-up dynamic programming strategy [23] to find the cost-optimal query plans. Our optimizer generates multiple query plans and finds the plan with lowest estimated cost. A large query plan is generated by joining two small optimal query plans. The cost is computed based on the cardinalities of query patterns and intermediate results.

The cardinality estimation is a well-studied problem in relational databases and SPARQL engines [26, 27, 31]. To estimate the cardinality of join result, an effective approach is characteristic set [26]. In a RDF graph R, the characteristic set $S_C(s)$ of a subject s is the set of related predicates: $S_C(s) = \{p|\exists o, (s, p, o) \in R\}$.

The idea of characteristic set is that semantically similar subjects (e.g. University of California and University of Michigan) usually have the same characteristic set. For every characteristic set, the number of distinct subjects that belong to the characteristic set, and the number of occurrences of the predicates in these subjects are recorded and used to estimate the cardinality. For example, given a characteristic set {*president, undergraduate*}, there are 100 distinct subjects belong to this characteristic set. The numbers of occurrences for *president* and *undergraduate* are 150 and 110 respectively. Consider a SPARQL query with two query patterns:

SELECT ?s ?o1 ?o2 .
{?s president ?o1 .
?s undergraduate ?o2 . }

Suppose that only this characteristic set contains both predicates in the query. Then the result cardinality is estimated as: $100 \times \frac{150}{100} \times \frac{110}{100}$ = 165.

Characteristic sets provide highly accurate estimation of cardinality. But it can not be used to estimate the cardinality of SPARQL^T queries since the statistics of temporal RDF graph vary on different time points. Consider following SPARQL^T query:

SELECT ?s ?o1 ?o2 ?t
{?s president ?o1 ?t.
?s undergraduate ?o2 ?t.
FILTER(?t $\leq$ 01/01/2013) . }

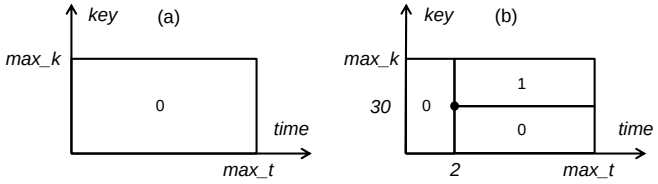


Figure 5: An Example of MVSBT Entry Split

To estimate the cardinality of this SPARQL^T query, we need to know the number of subjects that (i) belong to the set of temporal RDF triples valid in the period [0, 01/01/2013] and (ii) share the the characteristics set {*president, undergraduate*}. These statistics change with time and none of existing data structures can provide estimation of these statistics. Thus we introduce a temporal histogram to maintain the statistics of temporal RDF data in next Section. With temporal histogram, the characteristics sets can be easily integrated into our query optimizer.

6.2 Temporal Histogram

The problem of estimating the statistics of temporal RDF data is similar to the temporal aggregation that computes the aggregate value in a certain period. Thus we propose Compressed MVSBT that extends the temporal aggregate index Multiversion SB Tree to maintain the statistics of characteristic sets.

6.2.1 MVSBT

MVSBT [39, 40] is a temporal index that combines the features of MVBT and SB Tree [37] and supports the dominance-sum query. Given key k and time t, it returns the aggregation value of data records with keys less than k and timestamps smaller than t.

Similar to MVBT, MVSBT is a forest of trees with multiple root nodes and each of them points to an SB Tree for a temporal partition of data. Each entry in MVSBT corresponds to a rectangle in the key-time space. The structure of MVSBT entry is as follows:

- Leaf Entry: $< k_s, k_e, t_s, t_e, v >$
- Index Entry: $< k_s, k_e, t_s, t_e, v, ptr >$

A leaf entry has a key range (k_s, k_e), an interval (t_s, t_e), and a value v. The key range and the interval represent the rectangle covered by this entry in the key-time space. v maintains the aggregate value. The index entry has one additional pointer ptr that points to a child node. The rectangles of the entries are mutually disjoint and the union of all the rectangles is equal to the whole key-time space.

The node structure of MVSBT is similar to MVBT while the insertion algorithm is quite different. First we need to review two concepts introduced in [39]. Given a point (k, t), and max key value max_key, an entry is referred as *partly covered entry* if its key range intersects the range [k, max_key] but not contained in this range; an entry is referred as *fully covered entry* if its key range is contained by the range [k, max_k]. When a new point p (k, t) is inserted into a node N, the fully covered entries are vertically split at t. If there exists a partly covered entry, then p is inserted into the child page. If node N is in the leaf level, the partly covered entry is split into three entries based on point p. The node structure operations of MVSBT are similar to the ones of MVBT (Section 4.1.1) thus omitted.

An example of MVSBT for aggregate COUNT is shown in Figure 5. Figure 5(a) shows the initial entry of an empty MVSBT. The aggregate value is 0 since no point is inserted. Then one point with key 30 and timestamp 2 is inserted. The initial entry is split into three entries as shown in Figure 5(b). The entry on the top right corner has aggregate value 1 and other entries has aggregate value

Algorithm: leafEntrySplit(n_f, r, q)
Input: CMVSBT leaf node n_f, entry r, new point p (k, t)
1: $r.c = r.c + 1$
2: **if** $p.k > r.k_m$ **then**
3: $r.k_m = p.k$
4: $r.t_m = p.t$
5: **if** $r.c = c_m$ **then**
6: **if** $r.k_m \neq r.k_s$ *and* $r.t_m \neq r.t_s$ **then**
7: // Split r to three entries
8: $v' = r.c/2 + r.v$
9: r_1 = new entry($r.k_s$,$r.k_m$,$r.t_m$,$r.t_e$,k_s,t_m,v',0)
10: r_2 = new entry($r.k_m$,$r.k_e$,$r.t_m$,$r.t_e$,k_m,t_m,$r.c/2$,0)
11: n_f.add(r_1); n_f.add(r_2)
12: $r.t_e = r.t_m$
13: **else**
14: // Split r to two entries, similar to step 7 - 11, omit

Algorithm: indexEntrySplit(n_i, r, p)
Input: CMVSBT index node n_i, entry r, new point p (k, t)
1: $r.list$.add(p)
2: **if** length($r.list$) == l_m **then**
3: r_1 = new entry($r.k_s$,$r.k_e$,$p.t$,$r.t_e$,*new list*(),$r.ptr$,$r.c$)
4: n_i.add(r_1)
5: $r.t_e = p.t$

Figure 6: Algorithms for Entry Split in CMVSBT

0 since all the points in top right entry are larger than split point (30, 2). Suppose we have two queries (10, 1) and (40, 5). The first query point falls in the left entry and returns 0. The second query points falls in the top right entry and thus gets the result 1.

6.2.2 *Compressed MVSBT*

Although MVSBT has good performance on temporal aggregation, it takes too much space for storage. Since query optimization does not require very accurate estimates, we can trade accuracy with efficiency. Compressed MVSBT (CMVSBT) is based on the idea that instead of accurately recording the points and splitting entries for every new point, a CMVSBT entry contains m ($m \geq 1$) points. Then we can estimate the aggregate value using the ratio of covered space to full space. Instead of storing the exact values of points, we store the statistic values such as total number of points and max value. The structure of CMVSBT entry is as follows:

- Leaf Entry: $< k_s, k_e, t_s, t_e, k_m, t_m, v, c >$
- Index Entry: $< k_s, k_e, t_s, t_e, list, ptr, c >$

where k_s, k_e, t_s,t_e are for the key range and associated time interval. k_m and t_m store the max key value and time value of the points located in the rectangle of this entry; $list$ is a list of points; ptr is the pointer to a CMVSBT node; v and c are *fixed* and *current statistic values*. *fixed statistic value* refers to the aggregate value, while *current statistic value* refers to the aggregate value computed over the points contained in the current entry. The final statistic value is estimated by combining both values (discussed in Section 6.3).

Since the index nodes are visited more frequently than leaf nodes, we store the exact values of points in a list in the index nodes, while in leaf nodes we only maintain three statistics (k_m, t_m, c). The algorithm for data insertion in CMVSBT is similar to the one for MVSBT. Instead of splitting the entry for every input point, CMVSBT entry is split when the number of points in an entry is larger than the threshold. The split point is (k_m, t_m). The algorithm for entry splitting in CMVSBT for COUNT is shown in Figure 6. Let c_m and l_m denote the thresholds for the number of points in leaf nodes and index nodes. When a new point p (k, t) is inserted into compressed MVSBT, we look up the index nodes to find a set

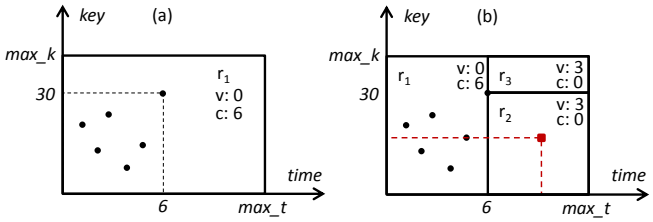


Figure 7: An Example of CMVSBT Entry Split

of nodes N whose rectangles cover this point. In a leaf node n_f, if p falls in the rectangle, c is increased by 1, and the max values (k_m, t_m) are updated if $p.k > k_m$ or $p.t > t_m$. Then if $c = c_m$, we split the entry based on the position (k_m, t_m). After split, the statistical values (v) of new entries will be equal to (i) $c/2 + v$ if the new entry has the same k_s with old one; (ii) $c/2$ otherwise (based on the logical splitting in MVSBT [40]). We use $c/2$ since we assume the points are uniformly distributed in the entry. The c of new entries are initialized to be 0. In an index node $n_i \in N$, if r_i is the lowest entry that fully covers p, p is appended to the end of $list$. Like MVSBT, compressed MVSBT also assumes that the data items come in nondecreasing time order. Thus $list$ is automatically sorted by time. If length($list$) $= l_m$, the entry is split on $p.t$ and c is copied to new entry. If $r.k_m = r.k$ or $r.t_m \neq r.t_s$, the split point ($r.k_m$, $r.t_m$) falls on the borders of rectangle. Then r is split into two entries. When we set $c_m = 1$ and $l_m = 1$, the algorithm in Figure 6 is the same with the split algorithm of MVSBT. More details about CMVSBT construction are available in our technical report [2].

Here we prepare a simple example to illustrate the process of entry split in CMVSBT. We assume that we have inserted six points into a compressed MVSBT, as shown in Figure 7(a). The threshold c_m is set to be 6. The max key $r_1.k_m$ is 30 and the max time $r_1.t_m$ is 6. Although the rectangle of r_1 is the whole space, all the points fall in the effective rectangle rec (key range:[0, k_m], time range:[0, t_m]). Since $r_1.c \geq 6$, we split it into three rectangles as shown in Figure 7(b). The values of r_1 are not changed since all the points still fall in r_1. $r_2.v$ is approximated by the number of points covered by the virtual center of r_2 (the red point). As we can see, the red rectangle covers half of rec, so $r_2.v = r_1.c/2 + r_1.v = 3$, $r_3.v = r_1.c/2 = 3$.

For each characteristic set, we need to maintain: (i) the number of distinct subjects and (ii) the number of predicate occurrences. As discussed in next section, each type of statistic values requires two CMVSBTs: one for start points and one for end points. Thus our temporal histogram consists of four CMVSBTs and the schema of characteristic sets. In RDF-TX , we set the max size of temporal histogram as 10% of raw data. If the size of temporal histogram is larger than the threshold, we increase c_m and l_m and merge the neighbor entries until the temporal histogram is small enough.

6.3 Statistics Estimation

Given a query q (k, t), compressed MVSBT estimates the statistics of data records with keys less than k and timestamps smaller than t. The query algorithm consists of two main steps: (i) starting from root node, we look up the CMVSBT nodes whose rectangle covers q; (ii) in each node, we find all the rectangles whose time range contains t and $k_s \leq k$, and accumulate the approximate statistic value v_a of these rectangles. In a CMVSBT entry, v_a equals to the sum of fixed statistics value v and current statistic value v_e. v_e is approximated by multiplying c by the proportion of query region in the rectangle $ratio$, as $c \times ratio$ where $ratio$ = $ratio_k \times ratio_t$. If $q.k \geq r.k$, $ratio_k$ = 1; otherwise, $ratio_k$

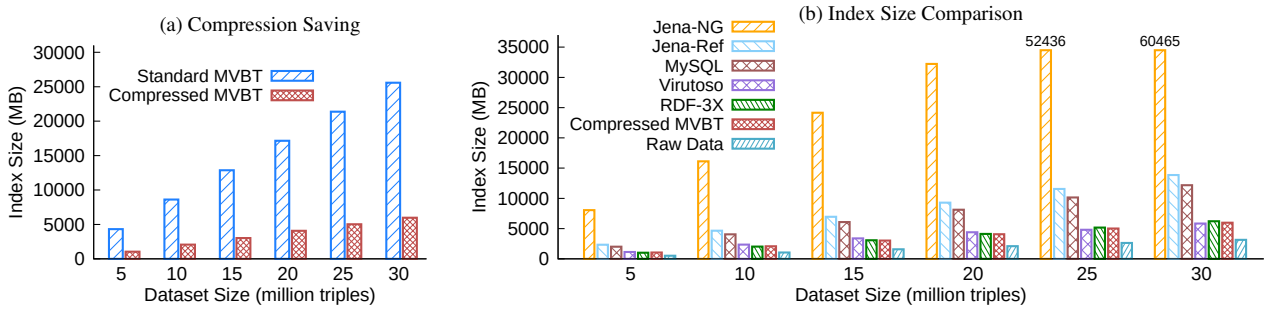


Figure 8: (a) Compression Saving for MVBT Index; (b) Index Size Comparison. The size of dictionary is included in the results.

$= (r.k_m - q.k)/(r.k_m - r.k_s)$. And $ratio_t$ can be computed in a similar way.

The query pattern in SPARQL^T is translated to a range query which is not supported by CMVSBT. Thus we use the query reduction approach [40] which reduces one range query into four point queries. In this approach, we need two CMVSBTs for the start points and end points of temporal RDF triples. Then the statistics in the query region (key: $[k_1, k_2]$, time: $[t_1, t_2]$) is calculated as:

$$Q_s(k_2, t_2) - Q_e(k_2, t_1) - Q_s(k_1, t_2) + Q_e(k_1, t_1)$$

where $Q_s(k, t)$ and $Q_e(k, t)$ refer to the point queries on the CMVSBT of start points and end points respectively.

During query optimization, we cache all the statistics to reduce the time on scanning CMVSBTs. When one statistic value is required, we first search the statistics cache. If it is not cached, then we use the CMVSBTs to estimate the statistic value.

7. EXPERIMENTAL EVALUATION

RDF-TX is implemented in Java as a sequential main memory query engine. To evaluate the performance of our system, we conduct experiments on several real world datasets and compare results with alternative approaches.

7.1 Experiment Setup

7.1.1 Dataset

Wikipedia. Wikipedia [4] is a real world dataset extracted from the edit history of English Wikipedia. We parse the raw file and generate 38 million temporal RDF triples as our test benchmark. This dataset contains the history of 1.8 million subjects and 3500 frequent predicates (used in more than 500 triples).

GovTrack. GovTrack [1] is a public dataset that contains the information about congressmen, votes, bills and committees. There are 20 million historical records for 0.4 million subjects and 60 related events (e.g. conressman election or bill voting). We parse the XML source files to temporal RDF triples as our test dataset.

Yago2. Yago2 [18] is a knowledge base derived from Wikipedia, WordNet and GeoNames with more than 30 million temporal RDF triples. Due to space limit and the fact that the evaluation results on Yago2 are very similar to Wikipedia and Govtrack, we leave the results on Yago2 in our technical report.

7.1.2 Implementation and Configuration

RDF Reification. RDF reification provides a way to store RDF triple and its meta knowledge in standard RDF model by representing annotated RDF triple as an entity with following properties: *subject, predicate, object, meta knowledge*. Similarly, we represent a temporal RDF triple as an entity with five properties: *subject, predicate, object, start time, end time*. Then SPARQL^T queries are easily rewritten to SPARQL queries. We evaluate the reification approach in three well known RDF engines: Jena v2.13 [36], Virtuoso v7.20 [3] and RDF-3X v0.3.8 [27].

RDBMS-based Approach. Temporal RDF triples can be stored in a relational table with five columns *subject, predicate, object, start time, end time*. We choose MySQL memory engine (v5.5) in our evaluation since it supports in-memory B+ tree index. We build four B+ tree indices on SPO, SOP, PSO, OPS and two additional indices on start/end time for evaluation of temporal constraints.

Named Graphs. Named graph [11] is an extension of RDF model that identifies graphs with URLs and allows graph metadata such as provenance and trust. We implement the approach described in [32] that stores temporal information as graph metadata using Jena Named Graph implementation. We also test Ng4j v0.9.3 implementation [9], but it is much slower than Jena and other approaches, so we leave the results on Ng4j in our technical report [2]. In the rest of this paper, we use "**Jena Ref**" and "**Jena NG**" to denote Jena Reification and Jena Named Graph respectively.

RDF-TX . Our query engine is a single-thread implementation using compressed MVBT as indices. Only the construction of compressed MVBT is paralleled (using at most four threads).

All the experiments are performed on a machine with 4 AMD Opteron 6376 CPUs (64 cores) and 256GB RAM running Ubuntu 12.04 LTS 64-bit. The index decompression time is included in the query execution time. The execution time reported is calculated by taking the average of 5 runs. The datasets and queries are available in our website [2].

7.2 Index Space

We first investigate the effectiveness of our delta encoding techniques (Section 4.2). We implement the standard MVBT indices (4 indices: SPO, SOP, POS, OPS) with numeric keys as baseline. Figure 8 (a) shows the space costs of standard MVBT and compressed MVBT in Wikipedia dataset. On average, our delta encoding technique reduces the space cost of MVBT by 76%.

Then we compare the space overhead of compressed MVBT and other types of index in Wikipedia, as shown in Figure 8 (b). Since Wikipedia has a large number of unique timestamps, most named graphs are very small (≤ 5 triples). Thus indexing named graph incurs a lot of overhead and Jena Named Graph takes far more space than other approaches. The space cost of MySQL memory engine and Jena Reification are similar, which are 3-4 times of raw data. The index space of our implementation is almost the same with Virtuoso and RDF-3X, while the query performance is much better as shown in Section 7.3. On average, the space of our comprehensive indices (4 compressed MVBT + dictionary) is about 1.8 times of raw data. The results for GovTrack dataset are similar and thus omitted.

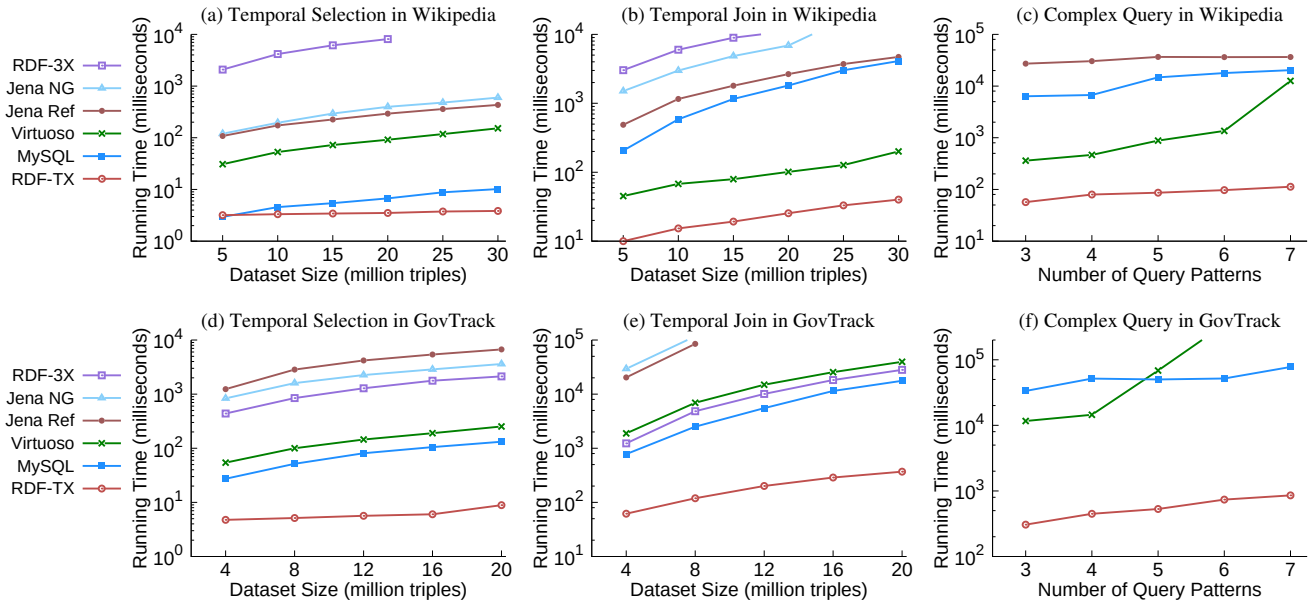


Figure 9: Query Running Time in (a - c) Wikipedia; (d - f) GovTrack.

7.3 Query Performance

To evaluate the query performance of our system, we created three sets of queries: (a) Temporal selection queries, as Example 2 in Section 3; (b) Temporal join queries, as Example 4; (c) Complex queries (2 or more temporal joins). We use the first two query sets to evaluate the performance as the dataset size increases, and the third query set to evaluate the performance as the query pattern size increases. For all the implementations, we report the average warm-cache query execution time.

Temporal Selection and Join. We create 10 temporal selection and 10 temporal join queries for each dataset and conduct the experiments as the size of dataset N increases (N: 5-30 million in Wikipedia and 4-20 million in GovTrack).

Figure 9(a) shows the query execution time for temporal selection in Wikipedia. RDF-TX and MySQL show similar performance in small datasets. As the size of dataset increases, RDF-TX shows better performance than MySQL. In the largest dataset (30 million), RDF-TX is about 3X faster than MySQL and 10X faster than Virtuoso. Jena Named Graph and Reification are 2 orders of magnitude slower than SPARQL^T engine due to the slow index scan.

RDF-3X is much slower than other systems due to its poor support of constraints. Most historical queries involve temporal constraints. For instance, consider Example 2 in Section 3 that searches the budget of University of California in 2013. This query has one temporal constraint that the valid period of temporal RDF triple should overlap (01/01/2013, 12/31/2013). This constraint can be expressed as: $?t_s \leq 12/31/2013$ && $?t_e \geq 01/01/2013$. In RDF-3X, the numbers are encoded as strings. So for temporal constraints, RDF-3X converts strings back to integers at running time to evaluate the constraints, which is inefficient.

The results of temporal join in Wikipedia are shown in Figure 9(b). RDF-TX is about 2 orders of magnitude faster than MySQL and Jena, and 6X faster than Virtuoso. RDF-3X is still slow since the condition of temporal join (e.g. OVERLAP and MEET) is expressed as constraints in FILTER clause.

Figure 9 (d) (e) show the query execution time for temporal selection and join in GovTrack. These approaches take more time to execute since the query patterns (e.g. P and PT) return much more results in GovTrack due to the reduction of predicates. The RDF-3x performs better than Jena on this dataset since it has a smaller number of distinct time periods ($\sim$ 10000) and predicates. MySQL and Virtuoso are about 1 order of magnitude slower than RDF-TX on selection and 2 orders of magnitude slower on Join.

RDF-TX performs 1-2 orders of magnitude faster than most competitors for selection and join. An important reason behind this is that MVBT can process two-dimensional (key and time) range query in one operation, while SPARQL and SQL engines need additional join and index scan.

Complex Queries. We generate 25 complex queries for each dataset with increasing query pattern size (3-7). The generation process is as follows: a set of 5 queries is created initially, and each query has 3 query patterns; then we incrementally add query patterns to existing queries until the size of query patterns reaches 7. The experiment is conducted on two datasets (each has 20 million triples) and the optimizers are enabled in all compared approaches.

The evaluation results in Wikipedia are shown in Figure 9 (c)[3]. Jena Named Graph and RDF-3X are not reported since they are much slower than other approaches so we omit them. For RDF engine and RDBMS, a query with more patterns is translated to more joins, which increases the complexity of parsing and optimization. On average, RDF-TX is 2 orders of magnitude faster than MySQL and Jena, and 1 order of magnitude faster than Virtuoso.

The evaluation results for GovTrack are shown in Figure 9 (f). A notable change in this graph is that Jena is not reported in this experiment. Jena is too slow compared to other approaches on GovTrack since the query patterns usually cover a large portion of dataset, which leads to slow execution time if an inefficient join order is generated; meanwhile, the column-store traits of Virtuoso excel in this small predicate cardinality case. On average, our system is still about 2 orders of magnitude faster.

7.4 Effectiveness of Query Optimizer

In this section, we explore the impact of query optimizer in the

[3]The query running time of Virtuoso on pattern size 6/7 is averaged over four queries since Virtuoso generates a very inefficient join order for one query which takes more than 1 hour to finish.

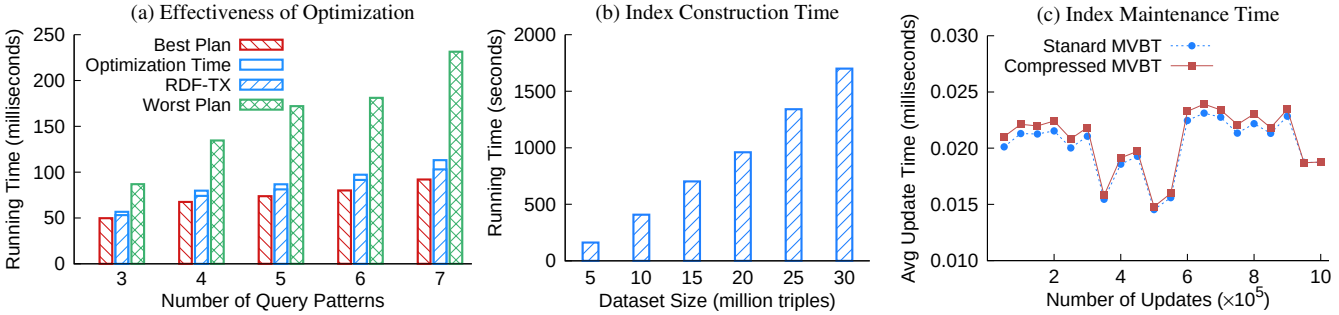


Figure 10: (a) Query Execution Time of the best/worst plans, and the plan generated by SPARQL^T optimizer for complex queries in Wikipedia (b) Index Construction Time (c) Index Maintenance Time

query evaluation. We enumerate all the possible query plans of the complex queries (Section 7.3) in Wikipedia and find the best and worst execution times. Figure 10 (a) shows the query execution times of best/worst plans and the plan generated by RDF-TX query optimizer (blue bar) in a Wikipedia set with 20 million triples. The result shows that the plan generated by our query optimizer is very close to the best performing execution plan. On average, the execution time of optimized query plan is about half of the time used by worst plan. In the relatively simple queries (3 query patterns), the difference between the best plan and the worst plan is small. As the number of query patterns increases, the difference becomes much larger. Thus, the optimizer is important for scaling up towards complex queries with a lot of query patterns. We also measure the time used for query optimization, which varies from 3.5 to 10 milliseconds as the size of query increases.

Then we measure the storage overhead of temporal histogram. The CMVSBTs for temporal statistics are built using the dictionary IDs. As discussed in Section 6.2, we merge CMVSBT entries and increase c_m and l_m until the size is small enough. In this experiment, the size of temporal histogram is 177.5 MB, which is about 8.5% of raw data size.

7.5 Index Construction & Maintenance

For large datasets, we first build standard MVBT and then compress the MVBT indices. In RDF-TX , the process of index construction is paralleled using at most 4 threads. We evaluate the index construction time for compressed MVBT time on different sizes of subsets of Wikipedia in Figure 10 (b) (compression time included). The time for index construction is approximately linear with the size of datasets, and it increases slightly faster in the datasets with 25 million and 30 million triples due to degraded performance caused by JVM garbage collection .

RDF-TX also supports the index update on compressed MVBT, which is important for real-time applications. Thus we further measure the average index maintenance time on a compressed MVBT index built from a 25 million subset of Wikipedia. We perform 1 million updates (68% insert, 32% delete) which simulates the changes in real Wikipedia edit history. Figure 10 (c) shows the results by comparing maintenance time of compressed MVBT with the time used on standard MVBT. Our compression technique shows a decent performance. Comparing with the update on MVBT, the update on compressed MVBT only takes 5% more time. This little overhead is negligible w.r.t. 76% space saved using compression.

8. RELATED WORK

Temporal Index. There has been a large body of research on temporal index in the literature [7, 19, 22, 24, 34]. MAP21 [24] is an index over B+Tree by mapping time ranges to one dimensional points, thus time intervals/points can be used as keys and queried in a B+Tree. OB+tree [34] organizes B+Trees in a versioned way with shared nodes whose contents do not change over versions. However, MAP21 and OB+tree only support single dimension query. BT-tree [19] enables branched versions along with the temporal index, while the time in our system is linear, i.e. no branching. MVBT [7] and TSB-Tree [22] are bi-dimensional indices, which satisfy our requirements exactly. TSB-Tree is a temporal index very similar to MVBT and implemented in Immortal DB [21] on Microsoft SQL Server, with better integration to SQL Server's existing index structures. The major difference between these two is that TSB-Tree migrates old data to a historical store during node splitting, while MVBT moves new data. Since MVBT is a general approach which is not targeted on specific platforms, we adopt and extend it in RDF-TX .

Query Languages and Systems for Temporal RDF. Several query languages [16, 29, 30, 32] have been proposed for temporal RDF triples. T-SPARQL [16] is a temporal extension of SPARQL based on a multi-temporal RDF model. The RDF triple is annotated with a temporal element that represents a set of temporal intervals. Thus a temporal join is expressed using additional functions (e.g. OVERLAP). At the best of our knowledge, no actual implementation of T-SPARQL is available. The τ-SPARQL system reported in [32] uses the temporal RDF model [17] and augments SPARQL query patterns with two variables ?s and ?e to bind the start time and end time of temporal RDF triples and express temporal queries. The evaluation is done by rewriting τ-SPARQL queries to standard SPARQL queries. Perry et al. [29] propose a framework to support temporal and spatial semantic queries. Simple selection and join queries are expressed using two temporal operators. These operators are implemented in Oracle by extending Oracle Semantic Data Sore and SQL functions. These works rely on relational databases/RDF engine to store and query temporal RDF triples, which results in complex SPARQL and SQL queries.

The tRDF system [30] extends the temporal RDF model [17] with indeterminate temporal annotations. The temporal queries are evaluated using tGrin index that clusters the temporal RDF triples based on graphical-temporal distance. However, tRDF only supports a subset of temporal queries discussed in this paper. Most significantly, temporal joins are not supported since tGrin index relies on the temporal distance to filter the triples, while the temporal distance between two temporally joined patterns can not be determined. STUN [20] system supports queries on annotated RDF, but it is not scalable for large temporal datasets.

9. CONCLUSION

In this paper, we present SPARQL^T and its system RDF-TX which supports powerful queries over the history of knowledge bases.

SPARQL^T enables the expression of a wide variety of temporal queries via simple extension of SPARQL graph patterns and built-in functions. SPARQL^T queries are efficiently evaluated in the backend query engine that achieves excellent performance by exploiting M-VBT as index and leveraging fast algorithms for range selection and temporal join. RDF-TX also features a query optimizer that uses the statistics of temporal RDF graphs to find the efficient join orders for complex SPARQL^T queries. Extensive experiments on real world datasets show that RDF-TX outperforms other approaches that use state-of-art RDF engines and relational databases in all kinds of queries and delivers 1 - 2 orders of magnitude performance improvement in complex queries. This confirms the effectiveness and superior performance of RDF-TX .

10. ACKNOWLEDGEMENTS

The authors would like to thank Maurizio Atzori, Massimo Mazzeo, Mohan Yang and Alexander Shkapsky for their insightful comments and suggestions on this research. This work was supported in part by NSF Grant IIS 1218471 and IIS 1118107.

11. REFERENCES

[1] GovTrack Dataset. https://www.govtrack.us/.

[2] RDF-TX Technical Report and Datasets. http://yellowstone.cs.ucla.edu/rdftx/.

[3] Virtuoso. https://github.com/openlink/virtuoso-opensource.

[4] E. Alfonseca, G. Garrido, et al. WHAD: Wikipedia Historical Attributes Data. *LRE*, pages 1163–1190, 2013.

[5] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Commun. ACM*, 26(11):832–843, 1983.

[6] M. Atzori and C. Zaniolo. SWiPE: Searching Wikipedia by Example. In *WWW*, pages 309–312, 2012.

[7] B. Becker, S. Gschwind, et al. An Asymptotically Optimal Multiversion B-tree. *VLDB*, 5(4):264–275, 1996.

[8] J. V. d. Bercken and B. Seeger. Query Processing Techniques for Multiversion Access Methods. In *VLDB*, pages 168–179, 1996.

[9] C. Bizer, R. Cyganiak, et al. Ng4j-named Graphs Api for Jena. In *ESWC*, 2005.

[10] C. Bizer, J. Lehmann, et al. DBpedia - A Crystallization Point for the Web of Data. *J. Web Sem.*, 7(3):154–165, 2009.

[11] J. J. Carroll, C. Bizer, et al. Named graphs, Provenance and Trust. In *WWW*, pages 613–622, 2005.

[12] C. X. Chen and C. Zaniolo. Universal Temporal Extensions for Database Languages. In *ICDE*, pages 428–437, 1999.

[13] J. Clifford and A. Rao. A Simple, General Structure for Temporal Domains. In *Temporal Aspects in Information Systems*, pages 17–28, 1987.

[14] J. D. Fernández, P. Schneider, et al. The DBpedia Wayback Machine. In *SEMANTiCS*, pages 192–195, 2015.

[15] S. Gao, M. Chen, et al. SPARQLT and its User-Friendly Interface for Managing and Querying the History of RDF Knowledge Bases. In *ISWC (Demo Track)*, 2015.

[16] F. Grandi. T-SPARQL: a TSQL2-like Temporal Query Language for RDF. *GraphQ*, pages 21–30, 2010.

[17] C. Gutierrez, C. A. Hurtado, et al. Introducing Time into RDF. *TKDE*, 19(2):207–218, 2007.

[18] J. Hoffart, F. M. Suchanek, et al. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[19] L. Jiang, B. Salzberg, et al. The BT-tree: A Branched and Temporal Access Method. In *VLDB*, pages 451–460, 2000.

[20] C. Kang, A. Pugliese, et al. STUN: Spatio-Temporal Uncertain (Social) Networks. In *ASONAM*, pages 543–550, 2012.

[21] D. B. Lomet, R. S. Barga, et al. Immortal DB: Transaction Time Support for SQL server. In *SIGMOD*, pages 939–941, 2005.

[22] D. B. Lomet, M. Hong, et al. Transaction Time Indexing with Version Compression. *PVLDB*, 1(1):870–881, 2008.

[23] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*, pages 930–941, 2006.

[24] M. A. Nascimento and M. H. Dunham. Indexing Valid Time Databases via B$^+$-Trees. *TKDE*, 11(6):929–947, 1999.

[25] S. B. Navathe and R. Ahmed. A Temporal Relational Model and a Query Language. *Inf. Sci.*, 49(1-3):147–175, 1989.

[26] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. ICDE, pages 984–994, 2011.

[27] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 19(1):91–113, 2010.

[28] J. Pérez, M. Arenas, et al. Semantics and complexity of SPARQL. *TODS*, 34(3), 2009.

[29] M. Perry, A. P. Sheth, et al. Supporting Complex Thematic, Spatial and Temporal Queries over Semantic Web Data. In *GeoS*, pages 228–246, 2007.

[30] A. Pugliese, O. Udrea, et al. Scaling RDF with Time. In *WWW*, pages 605–614, 2008.

[31] M. Stocker, A. Seaborne, et al. SPARQL Basic Graph Pattern Optimization using Selectivity Estimation. In *WWW*, pages 595–604, 2008.

[32] J. Tappolet and A. Bernstein. Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *ESWC*, pages 308–322, 2009.

[33] D. Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *PODS*, pages 58–67, 1996.

[34] T. Tzouramanis, Y. Manolopoulos, et al. Overlapping B$^+$-Trees: An Implementation of a Transaction Time Access Method. *DKE*, 29(3):381–404, 1999.

[35] O. Udrea, D. R. Recupero, et al. Annotated RDF. *TOCL*, 11(2):10:1–10:41, 2010.

[36] K. Wilkinson, C. Sayers, et al. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[37] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *ICDE*, pages 51–60, 2001.

[38] P. Yuan, P. Liu, et al. TripleBit: a Fast and Compact System for Large Scale RDF Data. *PVLDB*, 6(7):517–528, 2013.

[39] D. Zhang, A. Markowetz, et al. Efficient Computation of Temporal Aggregates with Range Predicates. In *PODS*, pages 237–245, 2001.

[40] D. Zhang, A. Markowetz, et al. On Computing Temporal Aggregates with Range Predicates. *TODS*, 33(2):12:1–12:39, 2008.

[41] D. Zhang, V. J. Tsotras, et al. Efficient Temporal Join Processing Using Indices. In *ICDE*, pages 103–113, 2002.

[42] X. Zhou, F. Wang, et al. Efficient Temporal Coalescing Query Support in Relational Database Systems. In *DEXA*, pages 676–686, 2006.