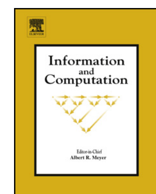
Information and Computation 259 (2018) 444–459

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco



User-friendly temporal queries on historical knowledge bases

Carlo Zaniolo [a,*], Shi Gao [a], Maurizio Atzori [b], Muhao Chen [a], Jiaqi Gu [a]

[a] University of California, Los Angeles, United States
[b] University of Cagliari, Italy

ARTICLE INFO

Article history:
Received 14 March 2016
Available online 7 September 2017

ABSTRACT

DBpedia and other RFD-encoded Knowledge Bases (KB)s give users access to encyclopedic knowledge via SPARQL queries. As the world evolves, the KBs are updated, and the history of entities and their properties becomes of great interest. Thus, we need powerful tools and friendly interfaces to query histories and flash-back to the past. Here, we propose (i) a point-based temporal extension of SPARQL, called SPARQL^T, which enables simple and concise expression of temporal queries, and (ii) an extension of Wikipedia Infoboxes to support user-friendly by-example temporal queries implemented by mapping them into SPARQL^T. Our main-memory RDF-TX system supports such queries efficiently using Multi-Version B+ trees, compressed indexes, and query optimization techniques, which achieve performance and scalability, as demonstrated by experiments on historical datasets including *Cliopedia* derived from Wikipedia dumps. We finally discuss how provenance information can be used to add valid-time features to these transaction-time KBs.

© 2017 Published by Elsevier Inc.

1. Introduction

By providing powerful search engines that give easy access to myriads of documents of potential interest, the web had an extraordinary impact on business, science, and society. With this triumph of computer technology now part of history, a consensus in slowly emerging that the next transformative breakthrough will occur when, in addition to the URL of documents of interest, the web will give people access to the knowledge contained in those documents needed to support *Question Answering* (QA) and *structured queries on text documents*. A significant first step toward realizing this vision of semantic computing is demonstrated by the DBpedia project that harvested the information contained in the InfoBoxes of Wikipedia pages and organized it into a Knowledge Base (KB) of RDF triples [17]. Indeed while InfoBoxes were originally added to Wikipedia pages to provide a structured summary for the important information contained in the page, they led to the creation of a knowledge base that can thus be searched by powerful SPARQL queries [44,47] employing systems such as Virtuoso [58] or Apache Jena [3]. KBs that organize information and knowledge using RDF, are rapidly growing in terms of scale and significance. In recent years, several projects have been devoted to create such KBs [23,39,42,10,12,14,30,50,63]. Thus we are witnessing a fast growth of a new technology which combines web document searching with the management and query-based search of traditional databases. Therefore, it is hardly surprising that problems and issues of traditional databases are also present here, starting with those relating to historical information management and temporal queries that have posed difficult challenges to database research and SQL standards over many years.

* Corresponding author.
E-mail addresses: zaniolo@cs.ucla.edu (C. Zaniolo), gaoshi@cs.ucla.edu (S. Gao), atzori@unica.it (M. Atzori), muhaochen@cs.ucla.edu (M. Chen), gujiaqi@cs.ucla.edu (J. Gu).

http://dx.doi.org/10.1016/j.ic.2017.08.012
0890-5401/© 2017 Published by Elsevier Inc.

Table 1
Statistics of Wikipedia Infobox edit history.

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

The management of historical information is now emerging as a critical issue for knowledge bases as well due to the fact that large knowledge bases undergo frequent changes. Table 1 lists the statistics of Wikipedia Infobox edit history, which shows that updates are quite common for many properties: e.g., on average each value in the population property of the city pages has been updated more than 7 times. This is not unique to Wikipedia, but it also happens in other knowledge repositories as well.

In fact there is a growing recognition of the importance of managing and querying the evolution history of knowledge bases and this is reflected in the technical literature. For instance, Gutierrez et al. [28] have proposed an extension of the RDF model with time elements, and several approaches have been introduced to support queries on temporal RDF datasets [26,45,48,51]. Most previous works employ relational databases and RDF engines to store temporal RDF triples and rewrite temporal queries into SQL/SPARQL for evaluation. The query languages in these works use an interval-based temporal model which leads to complex expressions for temporal queries, e.g., those requiring joins and coalescing [16,53]. At the physical level, previous approaches exploit indexes such as tGrin [48] 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 making possible very user-friendly interfaces that facilitate the expression of temporal queries by people who are not programmers. To support the queries over the evolution history of knowledge bases, we provide 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. Finally, our system includes a query optimizer that generates efficient join orders using a cost-based model and statistics on the temporal RDF graphs in the current database. Therefore, in this paper we describe the scalable solution for the problem of managing and querying massive temporal RDF KBs we have developed, and discuss the following research contributions:

- We extend Wikipedia Infoboxes into temporal infoboxes that operate as active forms where the history of values for a given property is displayed via pull-down menus.
- We provide a user-friendly system that supports a By-Example Structured Temporal Query (BESTQ) interface where users can specify queries by entering simple conditions in the active temporal infobox. This interface seeks to ensure usability by combining the user friendliness of Query-by-Example of relational databases [49] with the simple coalesce-free query formulation of the point-based temporal model [54,11]. For efficient execution, our system then translates these BESTQ queries into equivalent SPARQL^T queries as described next.
- We define 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 [4,21], and makes possible a simple translation of BESTQ queries into equivalent SPARQL and SPARQL^T queries.
- 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. We also feature a query optimizer that generates an efficient join order for SPARQL^T queries using the statistics of temporal RDF graphs. Experimental evaluation demonstrates the superior performance and scalability of RDF-TX compared with the state-of-the-art alternatives.
- We propose a provenance-based approach to manage and reconcile differences between transaction time and valid time in the real world.

The rest of this paper is organized as follows. Section 2 provides an overview of By-Example Structured Queries, and Section 3 describes the user-friendly BESTQ interface for temporal queries. The SPARQL^T query language is presented in Section 4. Section 5 describes our storage model and index compression techniques. The query evaluation techniques are discussed in Section 6. Section 6.2 introduces a query optimizer for join order optimization. We evaluate our system on real world datasets in Section 7, and discuss provenance-based bitemporalism in Section 8 and related work in Section 9. Finally, we conclude in Section 10.

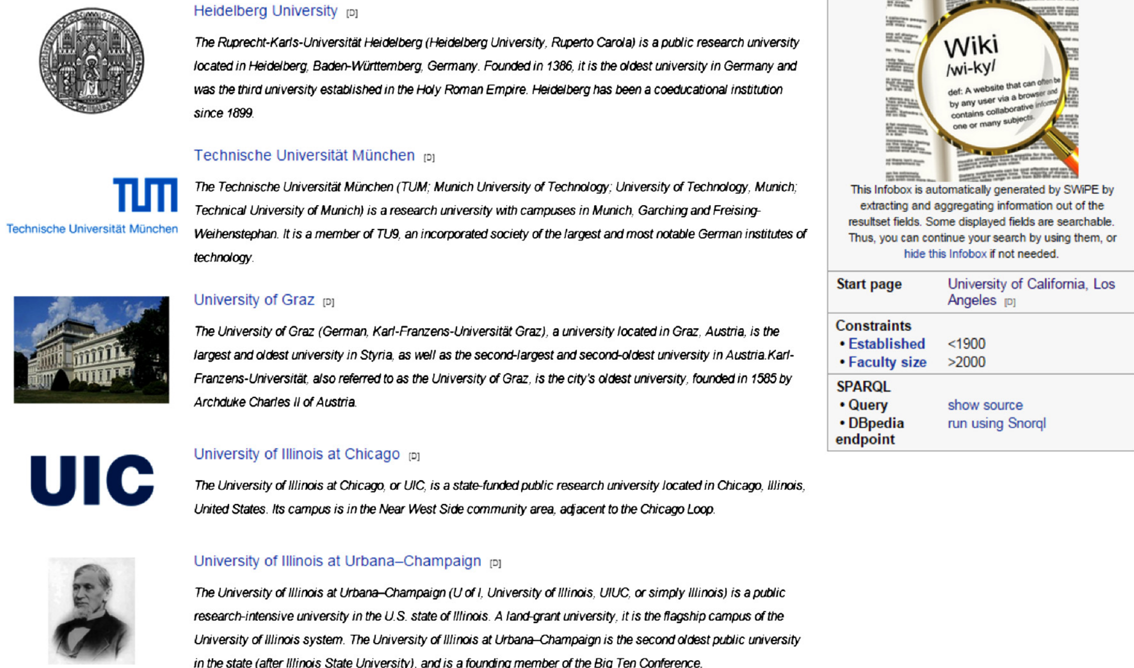


Fig. 1. First few snippets in the answer returned by the query in Fig. 2.

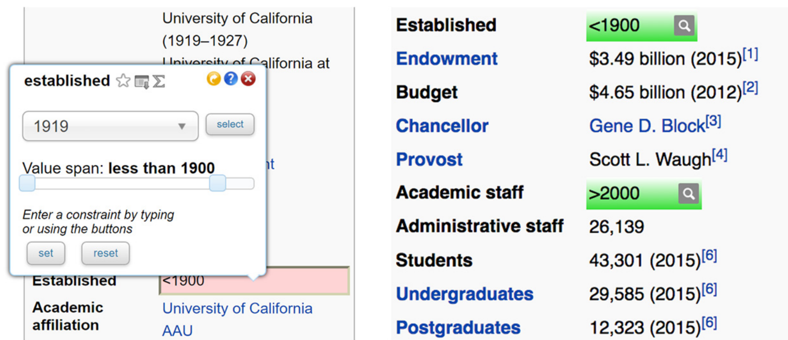


Fig. 2. A QBE query on a university Infobox. The system recognizes the first field as the DBpedia dbp:established property, making it editable. Having a numeric/temporal value, the interface also shows a range slider to set value constraints.

2. By-Example Structured Queries

Usability represents the first major problem, since the ability of using the powerful knowledge base via SPARQL engines is only available to those who can write SPARQL queries—thus casual users are excluded and similar issues apply to SPARQL^T. Even expert users who are familiar with query languages will need to spend some time and effort to become familiar with the internal terminology of the knowledge base for entities, values, and the thousands of attributes names (e.g., names such as: `foaf:givenName` and `dbprop:populationTotal`).

However, we have recently shown [4] that the usability problem for queries can be solved by a query-by-example approach that exploits as input query form the actual Infobox of some representative page of the entity type of interest.

A user who would like to find universities established before 1900 and with a faculty size above 2000 can (i) starts from the page of any university, (ii) click on a Swipe bookmark to refresh the same page with the Infobox turned into an active form (i.e. Fig. 2) that (iii) accepts conditions typed by the user, whereby upon another click the Swipe system takes those conditions and generates an equivalent SPARQL query producing results such as those shown in Fig. 1.

The first page of the returned results also shows an Infobox that summarizes the query, and provides links to meta-information, such as the SPARQL query used, which for the example at hand is shown in Fig. 3.

Another example that we have frequently used [4] focuses on finding cities with certain properties. For that, our user only needs to find in Wikipedia the page of a familiar city, and in its Infobox replace the existing values of the desired properties with conditions that specify the query. This is the well-known QBE approach that is credited for bringing ease-

```
PREFIX dbpprop: <http://dbpedia.org/property/>

SELECT ?university WHERE {
  ?university dbpprop:faculty ?faculty .
  ?university dbpprop:established ?date .
  FILTER (?faculty > 2000 && ?date < 1900)
}
```

Fig. 3. The SPARQL equivalent for the query in Fig. 2.

Fig. 4. The history of San Diego names is shown in the leader-name pull-down menu.

of-use to relational databases and is even more desirable here, since it shields the users from having to discover the internal names and organization of DBpedia. It is also quite powerful, since it supports the specification of queries involving joins and aggregates. Moreover, the user can still enter text conditions in the standard search box of Wikipedia to find the pages that satisfy both the structured query and the standard keyword based retrieval made popular by web search engines. Our experiments show that this combination yields very powerful queries producing selective high-precision answers [38].

Experiments comparing Swipe with Question Answering systems tested in the context of the QALD (Query Answering on Link Data) challenge also produced very encouraging results [5,57]. While most QALD systems are based on the use of a natural language interface, Swipe uses a WYSIWYG one. In the future, we expect systems that offer integrated support for questions and queries by combining the two kinds of interfaces. We are also devising a more advanced interface based on timelines for handling complex time constraints, inspired by the work on event visualization (see [46]). It will allow to drag multiple fields (such as the *Established* field in the previous example) into a canvas, where the user can express joint time constraints in a user-friendly way by moving cursors in each timeline.

3. Temporal infoboxes and by-example temporal queries

The infobox of the city of San Diego, California, clearly shows the current mayor of the city, but not the past mayors, nor it allows us to find who was the mayor at certain date, nor to find the total population of the city when Bob Filner was mayor. To provide answers to these and similar questions, we have developed a system that extends Wikipedia Infobox with historical information and extends SPARQL and Swipe with the ability of asking such temporal queries. We will now describe the new functionality supported by our system, starting from its user-friendly interface for temporal queries, and then progressing to the efficient implementation that supports those queries.

The first extension supported by our system is that the fields in the infobox are active, insofar as by selecting them a pull down menu opens that shows the history of that field. Thus, by selecting the mayor field in the infobox the user actually sees its history with the last four mayors of the city—Fig. 4.

But in addition to finding former values in the history of the entity properties, our user might enter query conditions in the fields of the infobox that are identified as active fields by their colorful backgrounds. Then, say that our user who

Fig. 5. What was the population of San Diego when Bob Filner was the mayor?

saw the previous mayor in the pull-down menu of San Diego, now wants to find its population at the time when Bob Filner served as the mayor. For that, the user can use the page of any city, including San Diego whose infobox is shown in Fig. 5. Since the infobox has now become an active form, its fields are editable and the user can enter "Bob Filner" in the *Government Mayor* and variable "?pop" in *City Population* and specify a temporal join by entering "[?t]" in both fields. Indeed, the brackets next to the values in the fields describe the periods of validity for those values, where periods are sets of days, since days is the granularity used in our temporal point-based representation. This simple way to express temporal joins, represent one of the most desirable properties of the point-based temporal model [54,11] that are further discussed in the next section. Our system generates and executes SPARQL^T query, as discussed later. Again, it is important to remember that this query could have been entered in the infobox of any city, and indeed since in our query conditions "San Diego" is not specified, our query will return the population of every city where Bob Filner served as the mayor.

4. Data model and query language

As the information in the real world evolves, the triples stored in the RDF knowledge bases are updated to reflect those changes. These updates capture the history of real world entities, which is of great interest to users. The addition of timestamps to RDF model was studied by Buraga et al. [13] who introduced the XML-based *Temporal Relation Specification Language* (TRSL) to express temporal relations in RDF. Later, Gutierrez et al. [27,28] introduced the point-based temporal RDF model that extends RDF model with time elements. In this paper, we employ this model [28] and temporally annotated RDF triples.

4.1. Temporal RDF

The basic RDF model consists of *(subject, predicate, object)* triples. In the temporal RDF model, each *(s, p, o)* triple is annotated with a time element, producing the quadruplet *(s, p, o, t)*. Here we use the edit history of Wikipedia entity *California* to illustrate how we model the evolution history using the temporal RDF model, as shown in Table 2. Consider the fact that Arnold Schwarzenegger served as the governor of California from November 17, 2003 to January 02, 2011. This fact can be represented as: <*California*, *governor*, *Arnold Schwarzenegger*>:[*11/17/2003 ...01/02/2011*], while [*11/17/2003 ...01/02/2011*] presents all the days between 11/17/2003 and 01/02/2011. DAY provides the basic granularity in our temporal model.

Note that we adopt the point-based temporal model that dovetails with our BESTQ interface and simplifies the expression of temporal queries at the logical level. As discussed in [54], while interval-based representations and point-based representations lead to query languages of equivalent expressive power, the latter leads to simpler queries particularly since coalescing is avoided. Thus in our system we support a point-based temporal model at the BESTQ level, but we use an interval representation at the physical level as the basis for efficient implementation. In practice, this approach is facilitated by the fact that queries on point-based representations can be easily mapped into equivalent ones on interval-based representations.

4.2. SPARQL^T query language

Once the history of a knowledge base, such as that shown in Table 2 has been stored using the temporal RDF model, the next question is how to express and support efficiently temporal queries. For that, we propose an extension of SPARQL called SPARQL^T that inherits the standard syntax from SPARQL and enriches it with temporal constructs.

Table 2
Temporal RDF triples for *California*.

Subject	Predicate	Object	Timestamp
California	Governor	Gray Davis	01/04/1999 ...11/16/2003
		Arnold Schwarzenegger	11/17/2003 ...01/02/2011
		Jerry Brown	01/03/2011 ...*now*
	Median Household Income	49,894	22/06/2007 ...08/04/2008
		54,385	08/05/2008 ...07/01/2010
		61,021	07/02/2010 ...*now*
	Total Population	38,332,521	12/30/2013 ...04/29/2014
		38,340,000	04/30/2014 ...12/23/2014
		38,802,500	12/24/2014 ...12/23/2015
		39,144,818	12/24/2015 ...*now*

SPARQL$^{\mathrm{T}}$ query pattern. To express selection where each element is a constant or variable, SPARQL uses triple query patterns (subject, predicate, object). For example, the SPARQL query pattern that finds the governor of California is: {*California governor ?p*}. While SPARQL is powerful, it is not designed for temporal reasoning. To specify the triple that is valid at a specified time or in a given periods, SPARQL$^{\mathrm{T}}$ extends the SPARQL query pattern with one temporal element. Formally:

- A SPARQL$^{\mathrm{T}}$ query pattern is a tuple $\{s \;\; p \;\; o \;\; t\}$ from $(\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{L}) \times (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V}) \times (\mathcal{T} \cup \mathcal{V})$

where $\mathcal{U}$, $\mathcal{L}$, $\mathcal{V}$, $\mathcal{T}$ respectively denote the set of Uniform Resource Identifiers (URIs), the set of literals, the set of variables, and the set of timestamps.

SPARQL$^{\mathrm{T}}$ syntax. The SPARQL$^{\mathrm{T}}$ was designed to (i) simplify the mapping of BESTQ queries into SPARQL$^{\mathrm{T}}$ queries, and (ii) to only use the basic **SELECT**, **WHERE**, **FILTER** constructs of SPARQL syntax, with which most users are already familiar with. We now provide some examples that illustrate the use of the SPARQL$^{\mathrm{T}}$ language and its ability to express powerful temporal queries.

Temporal selection. We first discuss temporal selection and how selection conditions are specified by SPARQL$^{\mathrm{T}}$ query patterns. An example of such query are "when" queries that retrieve the valid timestamps of given facts, such as the query in Example 1 that returns all the points in time when Gray Davis was the governor of California. The set of such points, form temporal periods that are displayed in the compact format $[t_s \dots t_e]$. Thus the execution of Example 1 against the temporal RDF graph in Table 2 returns [01/04/1999 ... 11/16/2003].

Example 1. When did Gray Davis serve as the governor of California?

```
SELECT ?t
WHERE {California governor Gray_Davis ?t}
```

Temporal constraints. Another very popular type of query is the time travel query that retrieves information from a past snapshot of the knowledge base. The temporal conditions (e.g. at a time point or within a period) are expressed in FILTER clause.

Example 2. Find the total population of California in August 2014.

```
SELECT ?pop
WHERE {California TotalPopulation ?pop ?t .
FILTER (?t = 2014-08-?) }
```

In this example, we express the temporal condition "during August 2014" as "**?t = 2014-08-?**" in the filter clause, where **2014-08-?** actually represents the 31 days of August 2014, since the temporal granularity of our stored data is days. Thus our query will retrieve all the triples where the subject is **California**, the predicate is **TotalPopulation**, and the timestamp falls in the set {2014-08-01, ..., 2014-08-31}.

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

Example 3. Find the median household income in California when Arnold Schwarzenegger served as its governor.

```
SELECT ?income ?t
WHERE {California governor Arnold_Schwarzenegger ?t .
          California medianHouseholdIncome ?income ?t}
```

A temporal join that in SPARQL^{T} is expressed by n query patterns sharing the same temporal variables, can be complex to express in an interval-based temporal model. However, to express the same query in the interval-based language we must specify overlap conditions between all pairs of intervals (i.e., $?I_1$ OVERLAP $?I_2$, $\ldots ?I_1$ OVERLAP $?I_n \ldots ?I_{n-1}$ OVERLAP $?I_n$ where $?I_1 \ldots ?I_n$ are temporal variables).

Aggregates and negation. Aggregate in SPARQL^{T} are expressed in using the syntax used in SPARQL and thus will not be discussed here. However, negation in SPARQL^{T} is expressed using a novel approach that takes advantage of the fact that the periods of validity of a triple implicitly define the periods in which the triple was not valid. Thus in SPARQL^{T} the pattern **California governor Arnold_Schwarzenegger ?t** denotes the periods of time in which Arnold Schwarzenegger was governor, whereas the pattern **California governor Arnold_Schwarzenegger notattime(?t)** denotes the periods of time when he was not. Thus to find the median incomes during the times in which Arnold_Schwarzenegger was not the governor of California, we can write the following query, which also demonstrates the use of **attime(?x)** as a synonym of **?x**:

Example 4. Find the median household income in California at the time in which Arnold Schwarzenegger was not the governor.

```
SELECT ?income  ?t
WHERE {California  medianHouseholdIncome  ?income  attime(?t) .
          California  governor  Arnold_Schwarzenegger  notattime(?t ) }
```

Negation increases the expressive power of SPARQL^{T}, and allows us to express query conditions that determine the start points of temporal intervals. For instance the start point for the interval(s) in office of Jerry Brown can be expressed by the following patterns where we find each day where Jerry Brown was the governor whereas he was not in the previous day. In these patterns we also use the notation **next(?tb)** as a synonym for **?tb+1**.

```
California governor Gray_Davis attime(?tstart).
California governor Gray Davis notattime(?tb).
FILTER(next(?tb)=?tstart).
```

The end(s) of time intervals can be identified by similar patterns, and then the end of each interval can be associated with its corresponding start by making sure that the time-span between the two is minimal (using a min aggregate). Therefore, given a point-based representation we can use SPARQL^{T} to derive an equivalent interval-based representation. Thus, no expressive power is lost using a temporal point-based representation. The theoretical interest of this observation follows from the fact that it generalizes the similar property that correlates point-based and interval-based temporal extensions for the relational model and queries [54]. In practice however, rather than forcing users to find the starts and the ends of intervals as described above, SPARQL^{T} provides the functions *TSTART(?t)* and *TEND(?t)* to enhance users' convenience and the efficiency of query execution. In particular, these constructs enable a simple expression and efficient implementation of queries using Allen's interval algebra operators MEET, BEFORE and DURING [2]. For instance the question "Who succeeded Arnold Schwarzenegger as California governor?" can be answered by the following query, expressing a meet operator, returns the name and the inauguration day of the next governor:

```
SELECT ?mame TSTART(?t2)
WHERE {California  governor  Arnold_Schwarzenegger  attime(?t1) .
          California  governor  ?name  attime(?t2 ) .
          FILTER (TSTART(?t2) = next(TEND(?t1)) }
```

Duration. To support the many temporal queries that involve reasoning on duration, SPARQL^{T} provides the built-in function *LENGTH*, defined as the difference between each TEND and the matching TSTART. If there are multiple non-consecutive intervals for a fact, *LENGTH* returns the length of each interval. Consider the following query:

Example 5. Find each person who served as the governor of California for more than four years and the length of their terms.

```
SELECT ?person  LENGTH(?t)
WHERE { California  governor  ?person  ?t .
FILTER(LENGTH(?t) > 1460 DAY) . }
```

For the current California governor, Jerry Brown, this query will return two separate lengths: one counting the days during his eight years as the 31st California Governor, and the other counting the days as the current 39th governor up to the day in which the query is issued. SPARQL^{T} also provides the function *TOTAL_LENGTH* defined as the sum of the lengths of valid intervals.

Non-temporal queries. SPARQL^T can also be used to query the current version of knowledge base by leaving the temporal subfield empty or equivalently specifying the temporal elements as *now*.

BESTQ versus SPARQL^T. The vertically integrated design of BESTQ and SPARQL^T assures that the translation of queries from the former to the latter is very simple. As shown in Fig. 5, the BESTQ interface features the bottom of the form an "Output Variable" slot and a "Filters" slot. Then, the equivalent SPARQL^T query is formed by copying the contents of these two slots are copied into the SELECT and FILTER fields. Finally the conditions in the WHERE clause are taken for those specified in the slots of the form, along with DBpedia's property name and object which they apply. Thus the SPARQL^T query generated from the screen in Fig. 5, is:

SELECT ?pop
WHERE { Leader_name ?Bob_Filner ?t .
San Diego_county_California Population_total ?pop ?t }

This simple example also illustrates the quantum leap in usability brought in by the BESTQ interface, which spare users from the complication of having to identify the internal names for subjects and properties used in DBpedia. Also observe that, to find San Diego's population when Bob Filner we can modify the query in Fig. 5 by replacing "**?**" with "**notattime(?t)**" which is then copied into the SPARQL^T query. The typical way in which negation is expressed in SPARQL consists of using a NOT EXIST, with conditions specified on the actual internal names for objects and properties used by DBpedia. But a cornerstone of BESTQ usability is that queries are specified directly on the infobox, without users having to deal with DBpedia internals. The new construct **notattime( )** makes it possible to express negation while preserving this key usability requirement. Then the introduction of **notattime( )** in SPARQL^T preserves the straightforward mapping from BESTQ to SPARQL^T.

While there is a simple translation for all BESTQ queries into SPARQL^T queries, the reverse is not true. In particular while BESTQ only allows variables in the object and time fields, SPARQL^T also allows variables to be used in the subject and property fields, as needed to, e.g., find out the state for which Gray Davis served as governor, or all the San Diego public offices, such as mayor and city attorney, covered by a certain person. Furthermore, our RDF-TX system, discussed next, supports efficiently queries performing searches in all the four dimensions S P O T, via multiple indexes and powerful query optimization techniques.

5. Supporting SPARQL^T queries

We considered several alternative approaches as discussed next. A first approach considered for managing temporal RDF triples relies on existing RDBMS. However, for searching both RDF information and temporal information, two sets of indices will be required and, as it will be shown in Section 7, such design results in significant costs in storage and retrieval time. A second natural approach consists of using RDF engines, such as Jena and Virtuoso, which have recently demonstrated remarkable improvements in performance and functionality. However, this would require the use of RDF reification approaches where, for each temporal RDF triple, we must use an entity instance with five properties: *subject, predicate, object, start time* and *end time*, resulting in a great increase in storage cost, and also in the time spent to execute the complex queries thus generated.

Therefore, after experimenting with existing systems, including Virtuoso RDF Quad-Store, we decided to seek better performance by designing and implementing a new system that integrates advanced indexing and data compression techniques. This led to the design and implementation of our RDF-TX system that achieves efficient support for SPARQL^T queries by (i) extensive indexing to expedite search in multiple dimensions, (ii) advanced compression techniques to minimize the memory required, and (iii) sophisticated estimation techniques to optimize traversal of complex RDF graphs [22].

5.1. Temporal indexes

After evaluating several alternatives, we employed the Multiversion B+ Tree [7], which is a temporal index structure with optimal worst case guarantees for data insert, update, and delete. In fact, the complexity of a temporal query on a version is asymptotically equal to the complexity of the query on a B+ tree that maintains all the data valid at that version.

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. 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. A current entry that stores data inserted in version i carries a period of (i, *now*). The delete operation modifies the end version of a live period.

In RDF-TX, we are dealing with timestamps triples (S, P, O) where, S, P, and O respectively stand for Subject, Property, and Object. Object can either be a value or a reference to another subject. This is illustrated by Table 2 where the first entry shows that S = California, P = Governor, O = Gray_Davis was true in the period 01/04/1999 . . . 11/16/2003. Now SPARQL, and SPARQL^T likewise, support queries where any component of the triple can be specified and the other components must be retrieved and filtered through various conditions. Multiple indexes are needed to support such queries efficiently. For instance, a query to find all California's Governors can be supported by an index that has S at the top level and P at the next

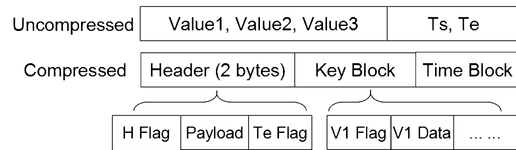


Fig. 6. Compressed MVBT entry.

level and O at the bottom level, i.e. an index (SPO). But a query to find the state for which Gray_Davis served as governor, will use the index (OPS), and this will also support queries where both the office and the state for which Gray_Davis served must be searched. Given that OPS is the search pattern used by BESTQ queries, and a very common pattern in all SPARQL queries, we started by including this index in our RDF-TX implementation, and focusing on eliminating redundancy in the remaining five indexes that provide support for less common queries, whereby RDF-TX now implements the following four indexes SPO, SOP, POS, OPS. These support search where only S, or P or S are specified, but also support the search where any pair of these is specified since, e.g., SOP support search when either SO or OS is specified. Now, the actual SPARQL^T search patterns are derived from the pattern above by adding temporal conditions on the periods associated with the triples, i.e., conditions specified by filters or, as in the case of joins, conditions specified by equality or inequality on time-points in such periods. During query evaluation, the query engine parses the SPARQL^T query and its prefix pattern to identify the corresponding MVBT index. Then our RDF-TX system supports index based search on all the above combinations, but avoids the prohibitively high memory cost of multiple indexes by the very effective data compression approach discussed next.

5.2. Index compression

To expedite the frequent operations of revising the index and consulting it during query evaluation, it is highly desirable that index be kept in main memory—a critical requirement not easily satisfied in practice since many systems do not have enough memory to accommodate the large number of MVBT entries needed in major applications. In RDF-TX, we explore two compression techniques: (i) dictionary encoding and (ii) delta encoding.

Dictionary encoding. RDF-TX employs dictionary encoding in index construction, since this reduces the index size and the time needed to compare literals. Thus, the system replaces each literal by a dictionary ID, whereby the indices store triples consisting of IDs and timestamps. While dictionary encoding alleviates the problem by typically reducing the storage requirements by 10%–20%, that is not enough and a compression technique that can reduce the space of MVBT indices more dramatically is needed for major real-life applications. For that, we have developed the delta compression technique discussed next.

Delta encoding. Our novel delta compression approach exploits two important characteristics of MVBTs. The first is that the entries in the MVBT node are sorted and neighboring entries often share the same prefix, a property that can be utilized to reduce space. The second is that all the node structure operations start from a *version split* that copies all live entries to a new node. This guarantees the query performance but leads to a lot of long intervals.

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.

Fig. 6 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). As shown in Section 7, the size of compressed MVBT is 15%–25% of standard MVBT.

6. Query processing and optimization

The evaluation of SPARQL^T queries consists of four steps:

- Parse the input query and translate the 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.

- Optimize the query plan for efficient join orders.
- Translate the query plan to an execution plan that is evaluated on compressed MVBT indices.

Translating query patterns. Since the temporal RDF graph is stored as interval-based temporal RDF triples, we translate the point-based SPARQL^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, 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* denotes the start-of-time. Here we use Example 2 discussed in Section 4 to explain how we translate the query patterns.

> **SELECT ?pop**
> **WHERE {California TotalPopulation ?pop ?t .**
> **FILTER (?t = 2014-08-?) . }**

In the evaluation, the query pattern in this query is converted to a rectangle in the two dimensional key-time space. The key range is (*California*, *TotalPopulation*, _)–(*California*, *TotalPopulation*, ∞), where _ and ∞ denote the extrema of the string domain. The time range is *2014-08-01–2014-08-31*. All the temporal RDF triples whose key range and time range intersect with this query rectangle are returned.

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 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 in the join order selected by the optimizer.

6.1. Executing query plan

In this section we focus on the implementation of the index scans and temporal joins in RDF-TX and omit the discussion of other operators (e.g. filter) that are implemented in the same ways as their non-temporal counterparts.

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 on our compressed index.

Temporal join. We explored three types of joins: *Merge Join*, *Hash Join*, and *Synchronized Join*. Merge join is very popular and widely used in existing SPARQL engines [41,66]. 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. Thus, in our system we used *Hash Joins* for most queries, except in the special cases discussed next where we used *Synchronized Joins*.

When the size of result is large, the cost of building a hash table may be very expensive. Thus we extend the synchronized join [67]. 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[1]; (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 a page and its predecessors are visited many times. Therefore, we optimize this algorithm by caching recently visited records. Therefore, 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.

[1] Each MVBT node has a key range and a time range which together makes up a rectangle in key-time space and covers the MVBT entries in the node. If two nodes have overlapped key/time range, it is denoted as intersected nodes. The join conditions can be formatted similarly as a query region in key-time space with a key range and a time range.

6.2. Optimization

Non-optimal 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.

For queries that involve multiple temporal joins, we implement a query optimizer that uses the bottom-up dynamic programming strategy [37] 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 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 the use of Compressed MVSBTs [22], that extend the temporal aggregate index Multiversion SB Tree to maintain the statistics of characteristic sets.

7. Experimental evaluation

RDF-TX is implemented in Java as a single-thread main memory query engine, which reflects the fact that most RDF knowledge bases can be easily fit in main memory. To evaluate the performance of our system, we conduct experiments on several real world datasets and compare the results with alternative approaches. We now provide a summary of our results on Yago dataset. Experiments on other datasets such as Wikipedia and Govtrack are described in more details in [22].

Dataset

Yago. Yago2 [30] is a knowledge base derived from Wikipedia, WordNet and GeoNames with more than 30 million RDF triples which are annotated with valid timestamps. The Yago temporal RDF dataset used in our experiments is extracted from Yago2 (v2.4.0_full). We analyzed the raw dataset, parsed the "occursSince" and "occursUntil" relations, and generated about 31.7 million temporal RDF triples. Several minor changes are made to fit the dataset into temporal RDF model and our system. Since in our model every RDF triple has a live period, the facts with unknown start/end time are assigned with a random time point. For the facts that appear before year 1905, we move their start time to 1905 and end time w.r.t. the length of duration. We perform such adjustments because year 1905 is approaching the minimal 32-bit integer limit if represented in UNIX timestamp and some competitor systems (e.g. RDF-3X) only supports 32-bit integer types.

Implementation and configuration

RDF reification. RDF reification is a way to represent an annotated RDF triple as an entity with following properties: *subject, predicate, object, meta knowledge*, which enables both RDF triple and its meta knowledge stored in standard RDF model. Similarly, a temporal RDF triple is represented 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 this approach in three well known RDF engines: Jena v2.13 [60], Virtuoso v7.20 [58] and RDF-3X v0.3.8 [41].

We compare our system with these systems because RDF reification is a natural approach followed by previous works to solve this problem, and Jena/Virtuoso/RDF-3X are the state-of-art RDF engines which show orders of magnitude performance gains comparing with other systems. In the experiments, we set the cache size of Virtuoso as 32 GB that is enough for the database files; Jena and RDF-3X rely on the file cache of operating system. We also explored the use of RAM disk to reduce I/O cost, which delivers similar results with system file cache.

RDBMS-based approach. RDBMS-based approach employs traditional relational database to store RDF triples. Thus 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 [15] 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 [51] that stores temporal information as graph metadata using Jena Named Graph implementation. We also tested Ng4j v0.9.3 implementation [9], but since it proved to be significantly slower than Jena and other approaches, it was not included in our experiments. 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 data loader is paralleled.[2]

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.

[2] Since we have 4 indices, the loading process can be naively paralleled using 4 threads.

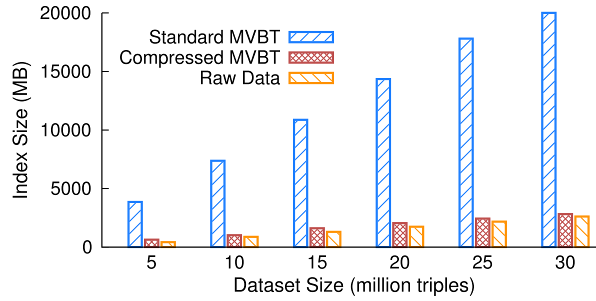


Fig. 7. Space cost and compression saving on Yago dataset.

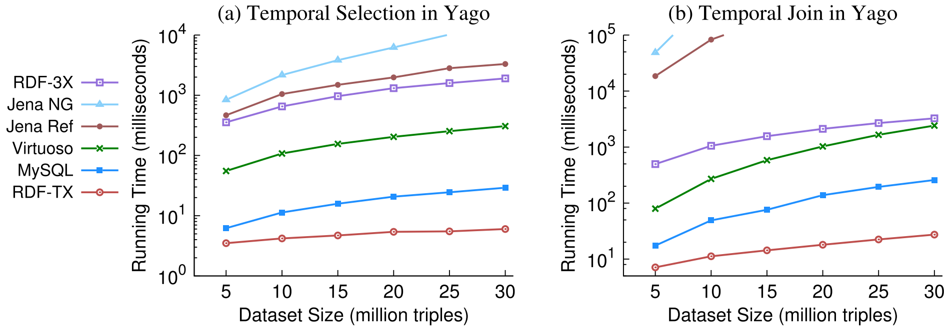


Fig. 8. Query running time in Yago: (a) temporal selection; (b) temporal join.

7.1. RDF-TX index space

We first investigate the effectiveness of our delta encoding techniques (Section 5.2), Thus we implement the standard MVBT indices (4 indices: SPO, SOP, POS, OPS) with numeric keys as baseline, and then measure the space costs of standard MVBT and compressed MVBT as shown in Fig. 7.

On average, the space of our comprehensive indices (4 compressed MVBT + dictionary) is $\sim$ 1.2 times of raw data on Yago database, which means that our index only introduces 20% storage overhead. Note that the space cost of our indices on Yago dataset is smaller than the ones on Wikipedia and Govtrack (1.4–1.8 times of raw data) [22] since Yago has fewer predicate names (about 30), which results in a lot of common prefixes in our indices and thus more space saving using index compression.

The results show that standard MVBT is not suitable for comprehensive indexing since its space cost is about 8–10 times of raw data (four indices). Our delta encoding technique significantly reduces the space cost of MVBT by 85% on the average.

7.2. RDF-TX query performance

To evaluate the query performance of our system, we created two sets of queries: (a) Temporal selection queries, as Example 2 in Section 4, where each query consists of one query pattern and several temporal constraints; (b) Temporal join queries, as Example 3 in Section 4 where each query has 2 query patterns.

We use these two query sets to evaluate the performance as the dataset size increases (from 5 million to 30 million temporal triples). For all the implementations, we report the average warm-cache query execution time.[3] Figs. 8(a) and 8(b) show the query execution time for temporal selection and join in Yago2 dataset.

As we study the results shown in Fig. 8, we see that for temporal selection queries, SPARQL^{T} and MySQL are much faster than other approaches. RDF-TX outperforms MySQL on all subsets with better scalability. In the largest dataset (30 million temporal RDF triples), our implementation is approximately four times faster than MySQL. The reason behind this is that our index supports two-dimensional (key and time) range query in one operation, while traditional index scans would require multiple passes and additional joins. For results of other engines, Virtuoso comes next, and it is about one order of magnitude faster than RDF-3X. The column-store traits of Virtuoso make it efficient in dealing with small cardinality cases of predicates. RDF-3X is slow in temporal selection due to its poor support of constraints (most historical queries involve temporal constraints). Since numbers in RDF-3X are encoded as strings, for temporal constraints, RDF-3X converts strings back to numeric values at running time to evaluate the constraints, whereby performance is significantly impacted. Since

[3] Since the temporal facts in Yago2 only involve 30 predicates, it is difficult to create complex queries. Interested readers are referred to [22] for the evaluation of complex queries in RDF-TX.

Jena index scan is slower than most approaches and the named graph approach on Jena incurs in extra overhead, Jena Ref and Jena Ng are the slowest among all approaches—about 2 to 3 orders of magnitude slower than RDF-TX on selection.

For temporal join queries, RDF-TX is one order of magnitude faster than MySQL due to our fast index scan and efficient temporal join algorithms. Virtuoso is one order of magnitude slower than MySQL. RDF-3X is still slow since the condition of temporal join (e.g. OVERLAP and MEET) is expressed as constraints in FILTER clause, but the result curve shows that it actually scales better than Virtuoso and performs nearly the same on 30 million dataset. A clear difference on the join result is that the gap between Jena based and other approaches becomes huge (more than 2 orders of magnitude). We believe it is the result of Jena index structures, which makes the performance of Jena based approaches to underperform on joins.

In summary, RDF-TX performs 1–2 orders of magnitude faster than most competitors for selection and join. The superior performance results obtained by our system are largely due to the index compression and query optimization techniques we used. The interested reader is referred to [22] for a further discussion and experimental evaluation of these techniques.

8. Cliopedia, bitemporalism, and provenance

In the previous sections we have presented a powerful system that allows users to query the history of RDF knowledge bases, such as the Cliopedia Historical Knowledge Base that is described next. Cliopedia is a temporal RDF knowledge base describing the evolution history of Wikipedia Infoboxes for the last 13 years (2003–2015), and was built by parsing Wikipedia dumps to extract Infobox updates. To capture the changes in the Infobox, we parsed all the Wikipedia dumps and identified the edits that change the Infobox part. Extensive data cleaning was then performed on the results. We detected and removed all the updates that (i) violate the type or format constraints of Wikipedia, (ii) refer to deprecated attributes, and (iii) were obviously caused by typos. Deprecated attributes are those that were used only once in the history and are not currently valid. The typos are identified using the edit distance. The resulting knowledge base tracks the history of 2.1 millions of pages for a total of 86 millions of updates. Therefore, Cliopedia contains twice the historical information provided by WHAD [1].

Since Cliopedia records and reports the time in which changes were made upon Wikipedia data shown to the public, in temporal database taxonomies Cliopedia will be surely classified as a Transaction-Time database (a.k.a. System Time database). As such, Cliopedia reports the precise history of what was shown to Wikipedia users over time. In this role, Cliopedia fulfills an important function, which is similar to that of public libraries keeping copies of the newspapers published in the last several years. Of course, as in any kind of publication, there are significant issues of timeless and accuracy that affect Wikipedia and therefore Cliopedia. Thus the natural question which arises is whether Cliopedia should time-stamp its information with both valid-time and transaction time as bi-temporal databases do.[4] However, we see major challengers if want to transition to bitemporal support for KBs derived from Wikipedia. One first difficulty pertains to ease of use, since exposing the complexities of bitemporalism to end-users would represent a major hurdle for any user-friendly interface. But even before that, there is the problem that valid-time timestamping requires dedicated and responsible curators. The volunteer-based crowdsourcing approach that now makes possible Wikipedia and DBpedia falls clearly short of satisfying those requirements. Moreover, if we assume that in the future Wikipedia will be managed by curators that update it in a very timely and accurate fashion, then this will reduce the differences between the transaction-time KB and its valid-time revision to a point that the cost and complexity of dealing bi-temporal KBs will become increasing questionable in the face of such diminished benefits. Therefore, in the next section we discuss alternative approaches that are practically appealing in situations where only limited curator resources are available, and thus they are used selectively to provide more timely and accurate historical information on entities and properties of particular interest.[5]

8.1. Provenance-based bitemporalism

We are currently exploring a more nuanced approach based on explicit provenance for Cliopedia. In this approach, the original transaction-time entries for Cliopedia will still be generated from the Wikipedia dumps as described in the previous sections, but revisions are then allowed and will recorded as annotations into a provenance log which is also kept alongside Cliopedia. This information will be used under two possible scenarios. In the first scenario, the KB will remain the transaction-time history derived from the Wikipedia dump, however corrections to this information can be retrieved from the provenance annotations, which besides the actual correction (such as, the revised spelling for the name of the mayor of San Diego, or the revised inauguration day for a mayor whose name is correctly spelled) also records the following information: (i) the time when the correction was made, (ii) the author of the correction, and (iii) the external source/authority for the corrected information.

We argue that for most practical situations where timely and extensive curation is not possible, this approach is practically more sensible than approaches like that of TSQL2 which encourages unlimited sequences of corrections to be entered into the database without requiring any supporting information, other than the time when the revision was made. Under

[4] We assume that valid-time alone is unacceptable since this would be tantamount to a misrepresentation of the actual history of Wikipedia.

[5] This is actually what happens now in Wikipedia, where there is always plenty of volunteer crowdsourcers for popular people and things, but not for others.

this one-history+ annotations approach, the current query language and user friendly interfaces will remain unchanged, but the system will provide users with easy access to provenance annotations that are relevant to the results returned by their queries.

Furthermore we expect that the annotation-based approach will also support a graceful evolution to a second scenario, in which the transaction-time history is replaced by valid-time history in selected situations where the quality of the curator approach justify this transition. In every case, casual users, using friendly endpoints, will only see one history, which is basically valid time history for the segments of the KB where this has been created and activated, and the transaction time history for the rest, since this remains the best approximation of the valid time history. Clearly this is the best solution for casual users; moreover, the more inquisitive historian and scholarly user will be provided with the ability of carrying our a more in-depth investigation using the annotations. We previously discussed how Cliopedia's annotated transaction-time history allow user to infer possible valid-time information. But the symmetric situation holds when Cliopedia store instead valid time history that has been generated by modifying the transaction-time history with the information taken from the annotation; in this second scenario users could flash-back to original transaction-time history, or part thereof when they need to do so. In our current project, support for this more advanced functionality has been left for further work, except for the extension to the current MVBT system briefly discussed next.

The transaction time MVBT index can be easily extended. Each entry in the MVBT leaf node is extended with a list of annotations. The actual annotation values can be strings or numbers. Besides the four indices for transaction time history, we also build four indices for valid time history. The structure of valid time indices is almost the same with transaction time index except that each entry has one pointer that points to the corresponding entry in the transaction time index. In reality, only a small portion of Infobox is often associated with valid time history. Thus the space cost of additional valid time indices is relatively small. Currently we annotate the temporal RDF triples in Cliopedia with valid timestamps from Yago2 [30] and Wikidata [59].

Our study about the evolution of Wikipedia pages also confirm our belief that the management of historical information for knowledge bases and databases is as much an art as it is a science. Cliopedia underscores this fact, by proposing solutions that are very different from those of TSQL2, but also by its very name: in Greek mythology Clio ($k\lambda\epsilon\iota\omega$) is the muse who presides to the art of history.

9. Related work

Query languages and systems for temporal RDF

Several query languages [26,36,45,48,51] have been proposed for temporal RDF triples. T-SPARQL [26] 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 [51] uses the temporal RDF model [28] 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. [45] 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 [48] extends the temporal RDF model [28] with indeterminate temporal annotations. The temporal queries are evaluated using a tGrin index that clusters the temporal RDF triples based on graphical-temporal distance. However, tRDF only supports a subset of the temporal queries discussed in this paper. Most significantly, temporal joins are not supported since tGrin indexes used the temporal distance to filter the triples, while the temporal distance between two temporally joined patterns can not be determined. The STUN [32] system supports queries on annotated RDF, but it is not scalable for large temporal datasets.

Temporal indexing and clustering

A topic that is important for the efficient implementation of our system is that of temporal indexing and clustering techniques. There has been a large body of research on temporal index in the literature [7,31,34,40,55]. MAP21 [40] is an index over B+Tree based on mapping time ranges to one dimensional points; thus time intervals/points can be used as keys and queried in a B+Tree. OB+tree [55] 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 [31] enables branched versions along with the temporal index. The time in our system is linear, i.e. no branching. MVBT [7] and TSB-Tree [34] are bi-dimensional indices. Another kind of indexes are TSB-Trees which are temporal indexes very similar to MVBT and that are used in Immortal DB [33] of the Microsoft SQL Server, providing 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.

User-friendly interfaces to KBs

The problem of designing user-friendly interfaces to RDF KBs has been recognized as very important and has been the subject of much previous work (although we are not aware of previous work on user-friendly interfaces to temporal KBs).

A survey on search interfaces can be found in [61]. In addition to keyword-based search interfaces [64,43,18,6,52] which focus on improving the presentation within search engine results, a number of novel interfaces for semantic knowledge-base querying have been proposed. Most related to this paper are research on Exploratory Browsing and Faceted Search [29].

Some work has been done in the area of context-enhanced search, such as a Yahoo! product [65] and the IntelliZap [20,19]. Such technologies appears to be still focused on old-style keyword search (empowered by the text found around the keyword, i.e., the context), therefore based on other well-known research areas such as query-refinement and document summarization.

Another interesting contribution comes from a Google commercial project [25] where an extension of Wikipedia was developed for a service called Custom Search Engine (CSE) that allows keyword queries with results restricted to Wikipedia pages that are linked with the current user's page. No other kind of structured queries are allowed.

Work in [62] describes an interface for semantically annotated user content that will help semantic web-engines indexing. In [35], authors investigate iterative querying (IQ). The OpenLink iSPARQL visual query builder provides a graphical interface for formulating SPARQL queries against DBpedia.

There has also been a significant amount of work on providing natural language question answering on KBs [24,56], but temporal questions have not been considered. This provides further evidence of the difficulty of the problem that our system has been the first to solve successfully.

10. Conclusion

In this paper, we have described a user-friendly interface, a query language, and a vertically integrated system to support historical KBs, and demonstrated the significant benefits and applications they make possible once they are applied to Cliopedia which contains the history of Wikipedia infoboxes from the last 13 years.

Therefore, after discussing the powerful and yet user-friendly BESTQ interface, we have presented SPARQL^T and the RDF-TX system which supports powerful queries over the evolution history of RDF knowledge bases. SPARQL^T enables the expression of a wide variety of temporal queries via simple extension of SPARQL query patterns and built-in functions. We then discussed how SPARQL^T queries are efficiently evaluated in the RDF-TX query engine that achieves excellent performance by exploiting MVBT as index and leveraging fast algorithms for range selection and temporal join on MVBT. RDF-TX also features a query optimizer that uses the statistics of temporal RDF graphs to identify an efficient join order 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.

Finally, to demonstrate the benefits delivered by these new tools we have collected the history of the infoboxes of Wikipedia pages for the last thirteen years and demonstrated how powerful queries can now be written with ease on this historical KB that we named Cliopedia. We also discussed how these powerful capabilities might best be exploited in real world scenarios that are quite different from the temporal database scenarios now assumed by current SQL standards.

References

[1] E. Alfonseca, G. Garrido, J.-Y. Delort, A. Penas, WHAD: Wikipedia historical attributes data, LRE (2013) 1163–1190.
[2] J.F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843.
[3] Apache Jena, http://jena.apache.org/.
[4] M. Atzori, C. Zaniolo, SWiPE: searching Wikipedia by example, in: WWW, 2012, pp. 309–312.
[5] M. Atzori, C. Zaniolo, Expressivity and accuracy of by-example structured queries on Wikipedia, in: 24th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2015, Larnaca, Cyprus, June 15–17, 2015, 2015, pp. 239–244.
[6] Z. Bar-Yossef, N. Kraus, Context-sensitive query auto-completion, in: Proceedings of the 20th International Conference on World wide web, WWW '11, ACM, New York, NY, USA, 2011, pp. 107–116.
[7] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, An asymptotically optimal multiversion B-tree, VLDB 5 (4) (1996) 264–275.
[8] J.V.d. Bercken, B. Seeger, Query processing techniques for multiversion access methods, in: VLDB, 1996, pp. 168–179.
[9] C. Bizer, R. Cyganiak, E.R. Watkins, Ng4j-named graphs Api for Jena, in: ESWC, 2005.
[10] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, Dbpedia – a crystallization point for the web of data, J. Web Semant. 7 (3) (2009) 154–165.
[11] M.H. Böhlen, R. Busatto, C.S. Jensen, Point-versus interval-based temporal data models, in: Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23–27, 1998, 1998, pp. 192–200.
[12] K.D. Bollacker, C. Evans, P. Paritosh, T. Sturge, J. Taylor, Freebase: a collaboratively created graph database for structuring human knowledge, in: SIGMOD, 2008.
[13] S.C. Buraga, G. Ciobanu, A RDF-based model for expressing spatio-temporal relations between Web sites, in: WISE, 2002, pp. 355–361.
[14] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E.R. Hruschka Jr., T.M. Mitchell, Toward an architecture for never-ending language learning, in: AAAI, 2010.
[15] J.J. Carroll, C. Bizer, P.J. Hayes, P. Stickler, Named graphs, Provenance and Trust, in: WWW, 2005, pp. 613–622.
[16] C.X. Chen, C. Zaniolo, Universal temporal extensions for database languages, in: ICDE, 1999, pp. 428–437.
[17] DBPedia, http://wiki.dbpedia.org/, 2012.
[18] P. Ferragina, A. Gulli, A personalized search engine based on web-snippet hierarchical clustering, in: A. Ellis, T. Hagino (Eds.), WWW (Special Interest Tracks and Posters), ACM, 2005, pp. 801–810.

[19] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, E. Ruppin, Placing search in context: the concept revisited, in: WWW, 2001, pp. 406–414.
[20] L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, E. Ruppin, Placing search in context: the concept revisited, ACM Trans. Inf. Syst. 20 (1) (2002) 116–131.
[21] S. Gao, M. Chen, M. Atzori, J. Gu, C. Zaniolo, SPARQLT and its user-friendly interface for managing and querying the history of RDF knowledge bases, in: ISWC (Demo Track), 2015.
[22] S. Gao, J. Gu, C. Zaniolo, RDF-TX: a fast, user-friendly system for querying the history of RDF knowledge bases, in: EDBT, 2016.
[23] GeoNames, http://www.geonames.org/.
[24] C.Z. Giuseppe Mazzeo, Answering controlled natural language questions on RDF knowledge bases, in: EDBT, 2016.
[25] Google, A contextual search experience for Wikipedia (blog page), http://googlecustomsearch.blogspot.com/2009/10/contextual-search-experience-for.html, 2011.
[26] F. Grandi, T-SPARQL: a TSQL2-like temporal query language for RDF, in: GraphQ, 2010, pp. 21–30.
[27] C. Gutiérrez, C.A. Hurtado, A.A. Vaisman, Temporal RDF, in: ESWC, 2005, pp. 93–107.
[28] C. Gutierrez, C.A. Hurtado, A.A. Vaisman, Introducing time into RDF, Trans. Knowl. Data Eng. 19 (2) (2007) 207–218.
[29] R. Hahn, C. Bizer, C. Sahnwaldt, C. Herta, S. Robinson, M. Bürgle, H. Düwiger, U. Scheel, Faceted Wikipedia search, in: BIS, 2010, pp. 1–11.
[30] J. Hoffart, F.M. Suchanek, K. Berberich, G. Weikum, Yago2: a spatially and temporally enhanced knowledge base from Wikipedia, Artif. Intell. 194 (2013) 28–61.
[31] L. Jiang, B. Salzberg, D.B. Lomet, M.B. García, The BT-tree: a branched and temporal access method, in: VLDB, 2000, pp. 451–460.
[32] C. Kang, A. Pugliese, J. Grant, V.S. Subrahmanian, STUN: spatio-temporal uncertain (social) networks, in: ASONAM, 2012, pp. 543–550.
[33] D.B. Lomet, R.S. Barga, M.F. Mokbel, G. Shegalov, R. Wang, Y. Zhu, Immortal DB: transaction time support for SQL server, in: SIGMOD, 2005, pp. 939–941.
[34] D.B. Lomet, M. Hong, R.V. Nehme, R. Zhang, Transaction time indexing with version compression, Proc. VLDB 1 (1) (2008) 870–881.
[35] Y. Mass, M. Ramanath, Y. Sagiv, G. Weikum Iq, The case for iterative querying for knowledge, in: CIDR, 2011, pp. 38–44, www.crdrdb.org.
[36] B. McBride, M. Butler, Representing and querying historical information in RDF with application to E-discovery, HP Laboratories Technical Report HPL-2009-261, 2009.
[37] G. Moerkotte, 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, 2006, pp. 930–941.
[38] H. Mousavi, M. Atzori, S. Gao, C. Zaniolo, Text-mining, structured queries, and knowledge management on web document corpora, SIGMOD Rec. 43 (3) (2014) 48–54.
[39] MusicBrainz, http://musicbrainz.org/.
[40] M.A. Nascimento, M.H. Dunham, Indexing valid time databases via B^+-trees, Trans. Knowl. Data Eng. 11 (6) (1999) 929–947.
[41] T. Neumann, G. Weikum, The RDF-3X engine for scalable management of RDF data, VLDB J. 19 (1) (2010) 91–113.
[42] OPENCYC, http://www.cyc.com/platform/opencyc.
[43] P. Papadakos, S. Kopidaki, N. Armenatzoglou, Y. Tzitzikas, Exploratory web searching with dynamic taxonomies and results clustering, in: M. Agosti, J.L. Borbinha, S. Kapidakis, C. Papatheodorou, G. Tsakonas (Eds.), ECDL, in: Lect. Notes Comput. Sci., vol. 5714, Springer, 2009, pp. 106–118.
[44] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of sparql, arXiv:cs/0605124 [cs.DB], 2006.
[45] M. Perry, A.P. Sheth, F. Hakimpour, P. Jain, Supporting complex thematic, spatial and temporal queries over semantic web data, in: GeoS, 2007, pp. 228–246.
[46] F. Persia, S. Helmer, An event detection framework supported by a smart graphical user interface, in: 3rd International Workshop on Information Integration in Cyber Physical Systems, IICPS'16, Pittsburgh, Pennsylvania, 2016.
[47] E. Prud'hommeaux, A. Seaborne, Sparql query language for rdf, W3C working draft, 4 (January), 2008.
[48] A. Pugliese, O. Udrea, V.S. Subrahmanian, Scaling RDF with time, in: WWW, 2008, pp. 605–614.
[49] P. Reisner, Human factors studies of database query languages: a survey and assessment, ACM Comput. Surv. 13 (1) (1981) 13–31.
[50] P. Singh, T. Lin, E.T. Mueller, G. Lim, T. Perkins, W.L. Zhu, Open mind common sense: knowledge acquisition from the general public, in: Confederated International Conferences DOA, CoopIS and ODBASE, London, UK, 2002.
[51] J. Tappolet, A. Bernstein, Applied temporal RDF: efficient temporal querying of RDF data with SPARQL, in: ESWC, 2009, pp. 308–322.
[52] A. Termehchy, M. Winslett, Keyword search over key-value stores, in: Proceedings of the 19th International Conference on World Wide Web, WWW '10, ACM, New York, NY, USA, 2010, pp. 1193–1194.
[53] D. Toman, Point vs. interval-based query languages for temporal databases, in: PODS, 1996, pp. 58–67.
[54] D. Toman, Point vs. interval-based query languages for temporal databases, in: Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Montreal, Canada, June 3–5, 1996, 1996, pp. 58–67.
[55] T. Tzouramanis, Y. Manolopoulos, N.A. Lorentzos, Overlapping B^+-trees: an implementation of a transaction time access method, Data Knowl. Eng. 29 (3) (1999) 381–404.
[56] C. Unger, C. Forascu, V. Lopez, A.N. Ngomo, E. Cabrio, P. Cimiano, S. Walter, Question answering over linked data (QALD-4), in: Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15–18, 2014, 2014, pp. 1172–1180.
[57] C. Unger, C. Forascu, V. Lopez, A.N. Ngomo, E. Cabrio, P. Cimiano, S. Walter, Question answering over linked data (QALD-5), in: Working Notes of CLEF 2015 – Conference and Labs of the Evaluation Forum, Toulouse, France, September 8–11, 2015, 2015.
[58] Virtuoso, https://github.com/openlink/virtuoso-opensource.
[59] Wikidata, http://www.wikidata.org.
[60] K. Wilkinson, C. Sayers, H.A. Kuno, D. Reynolds, Efficient RDF storage and retrieval in Jena2, in: SWDB, 2003, pp. 131–150.
[61] M.L. Wilson, B. Kules, M.C. Schraefel, B. Shneiderman, From keyword search to exploration: designing future search interfaces for the web, Found. Trends Web Sci. 2 (1) (2010) 1–97.
[62] G. Wu, M. Yang, K. Wu, G. Qi, Y. Qu, Falconer: once sioc meets semantic search engine, in: Proceedings of the 19th International Conference on World wide web, WWW '10, ACM, New York, NY, USA, 2010, pp. 1317–1320.
[63] W. Wu, H. Li, H. Wang, K.Q. Zhu, Probase: a probabilistic taxonomy for text understanding, in: SIGMOD Conference, 2012.
[64] S. Xu, T. Jin, F.C.M. Lau, A new visual search interface for web browsing, in: Proceedings of the Second ACM International Conference on Web Search and Data Mining, WSDM '09, ACM, New York, NY, USA, 2009, pp. 152–161.
[65] Yahoo!, Y!q contextual search tool, http://searchenginewatch.com/article/2066478/Yahoo-Offers-New-YQ-Contextual-Search-Tool.
[66] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, L. Liu, TripleBit: a fast and compact system for large scale RDF data, Proc. VLDB 6 (7) (2013) 517–528.
[67] D. Zhang, V.J. Tsotras, B. Seeger, Efficient temporal join processing using indices, in: ICDE, 2002, pp. 103–113.