SPARQL^T and its User-Friendly Interface for Querying the History of RDF Knowledge Bases

Shi Gao[†], Muhao Chen[†], Maurizio Atzori[‡], Jiaqi Gu[†], and Carlo Zaniolo[†]

[†]University of California, Los Angeles; [‡]University of Cagliari
[†]{gaoshi, muhaochen, gujiaqi, zaniolo}@cs.ucla.edu; [‡]atzori@unica.it

Abstract. As the real world evolves, the information contained in the knowledge bases is updated accordingly and the evolution history of entities and their properties becomes of great interest to users, who thus need effective query tools to explore it. In this paper, we (i) introduce SPARQL^T and its user-friendly interface for expressing powerful structured queries on the history of RDF knowledge bases, and (ii) overview its underlying query engine that supports efficient evaluation of such queries and the management of historical information. To ensure ease of use, we take a *by-example structured query* approach that allows users to specify queries by entering simple conditions into the Infoboxes of Wikipedia pages extended with temporal fields. From this wysiwyg interface, the system derives and executes equivalent SPARQL^T queries, sparing users from having to learn the query language and the underlying knowledge base schema. Our demo will introduce SPARQL^T, its system and user-friendly interface, with the help of simple and intuitive examples.

1 Introduction

There is a growing interest in large scale knowledge bases such as DBpedia and Yago, which play a key role in semantic applications. In reality, large knowledge bases undergo frequent changes. When the information in the real world evolves, the RDF triples stored in the knowledge bases are updated to reflect those changes. The evolution history of knowledge bases captures and describes the changes experienced by real world entities and properties, which is of great interest to users. As a result, the management of historical information has emerged as a critical problem for RDF knowledge bases, motivating the launching of projects such as DBpedia Live [1].

In the project described in this paper, we collect the evolution history of knowledge bases and store it in the temporal RDF model [8]. Then we introduce SPARQL^T, a temporal extension of SPARQL that can express powerful structured queries on temporal RDF triples and describe our efficient in-memory query engine. The most significant contribution of SPARQL^T is that it solves the *Usability* and *Performance* problems that must be tackled when querying the evolution history of knowledge bases. For usability, we demonstrate a query interface that allows users who are unfamiliar with knowledge base schema and SPARQL^T syntax to query the history of knowledge bases. The interface uses the Wikipedia Infoboxes extended with temporal fields, where the user can enter temporal query conditions. From the modified Infoboxes and query conditions,

Predicate	Object	Timestamp
Mayor	Bob Filner	12/04/2012 ... 08/30/2013
	Todd Gloria	08/31/2013 ... 03/02/2014
	Kevin Faulconer	03/03/2014 ... *now*
Population	1322553	12/19/2012 ... 10/01/2013
	1307402	10/02/2013 ... 04/29/2014
	1345895	04/30/2014 ... 05/21/2015
	1381069	05/22/2015 ... *now*

Q1: SELECT ?t
{San_Diego Mayor Todd_Gloria ?t}

Q2: SELECT ?person
{San_Diego Mayor ?person ?t
FILTER(YEAR(?t) = 2013)}

Q3: SELECT ?pop ?t
{San_Diego Mayor Bob_Filner ?t .
San_Diego Population ?pop ?t}

Fig. 1. Left: the evolution history of *San Diego*; **Right**: example SPARQL^T queries

our system derives equivalent queries that are optimized and executed in our query engine. To achieve fast query evaluation, we used efficient storage and index schemes for temporal RDF triples based on Multi-Version B+ Tree (MVBT) [5], and built a query engine that takes full advantage of comprehensive indices to process SPARQL^T queries.

2 Data Model and Query Language

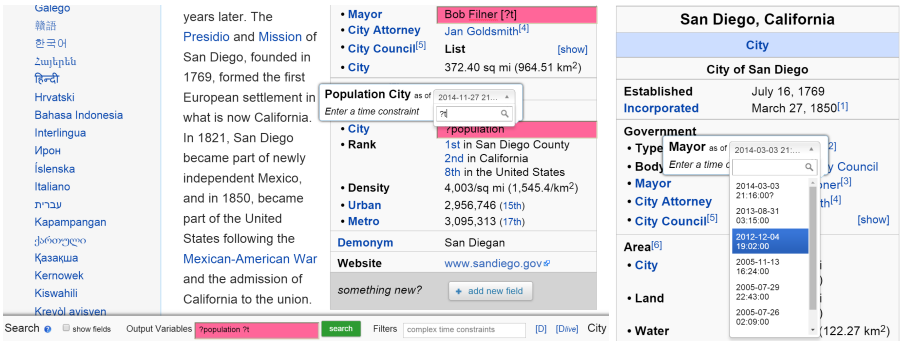
Knowledge bases can be represented as RDF graphs which consist of a set of RDF triples in the format of (*subject*, *predicate*, *object*), where *subject* and *predicate* are Uniform Resource Identifiers (URI), and *object* can be a URI or a literal value. For example, the statement "The mayor of San Diego is Kevin Faulconer" is represented by: (*San_Diego Mayor Kevin_Faulconer*).[1] Since the basic RDF graphs are designed for static information, we represent the evolution history of RDF knowledge bases using the temporal RDF graphs proposed in [8] that rely on a point-based temporal representation. Given a temporal domain $\mathcal{T}$, a *Temporal RDF Graph* consists of a set of RDF triples annotated with temporal elements $t \in \mathcal{T}$. Figure 1 shows the temporal RDF triples for the evolution history of subject *San Diego*, extracted from real world Wikipedia edit history. We use DAY as the granularity of time and *now* as current time.

To query such temporal RDF graphs, we propose SPARQL^T that extends SPARQL query pattern with a temporal element to match the temporal RDF triples. The SPARQL^T query patterns are quadruplets $\{s\ \ p\ \ o\ \ t\}$, where each element is either a literal or a variable. Thus, a SPARQL^T query is a set of SPARQL^T query patterns annotated with temporal conditions, such as those for the basic operators discussed next.

Temporal Selection. We start from temporal selection queries with one SPARQL^T query pattern. A typical example is the "when" query. By specifying *s, p, o* as literals and leaving *t* as a variable, users can retrieve the valid timestamps of given facts. This is the case of Q1 in Figure 1 that finds when Todd Gloria served as the mayor of San Diego. The SPARQL^T query pattern also allows users to perform RDF triple matching with temporal constraints (e.g., at a previous version or within a certain period), such as Q2 in Figure 1 which returns the mayor of San Diego in 2013.

Temporal Join. Another important query class is temporal join query. In SPARQL^T, a temporal join is expressed by multiple query patterns sharing temporal elements, i.e. days in our examples since this is the granularity of our point-based representation. An example of temporal join query is Q3 in Figure 1 that returns the population of San Diego when Bob Filner served as the mayor.

[1] For the sake of simplicity, we omit the prefix parts of URIs (e.g. http://dbpedia.org/resource/).



(a) (b)

Fig. 2. (a) Query Interface (b) Navigation Window for Property Mayor

The temporal constraints are expressed in the FILTER clause. We support a set of built-in functions including TSTART/TEND, YEAR/MONTH/DAY and PERIOD. With these functions, SPARQL^T readily expresses complex temporal conditions and Allen's operations [3]. For example, the MEET operatrion is expressed with constraint TEND(?t1) = TSTART(?t2). More features and query examples are discussed in [2, 6].

These examples illustrate that the point-based temporal model leads to queries that are simple to express using SPARQL^T . This property is not shared by other approaches [7, 9, 10] that have been proposed to support the queries on temporal RDF. In particular, the languages proposed in these works use the interval-based temporal model which leads to complex expressions for temporal queries requiring join and coalescing [11]. Because of lack of space, we refer our reader to the full paper [6] for a detailed discussion of the various approaches with respect to usability and performance.

3 User Interface and Query Engine

Our system consists of two main components: (i) a user interface that enables users to browse and search the knowledge and its history, and (ii) the SPARQL^T query engine that manages the temporal data and evaluates SPARQL^T queries.

User Interface. Our system provides a user-friendly interface that addresses the problem of usability by extending the *By-Example Structured Query (BEStQ)* [4] approach with editable temporal fields. The interface supports (i) querying the current knowledge base and its history and (ii) browsing the history of entities and properties. Our user who wants to find the population of San Diego when Bob Filner served as the mayor, might start by loading subject San Diego in our interface, as shown in Figure 2(a). Since all the fields are editable, he enters "Bob Filner" in the *Government Mayor* and variable "?population" in *City Population* and these two InfoBoxes are set with the same temporal variable "?t" to indicate the temporal join. Then our system generates and executes SPARQL^T query Q3 in Figure 1.

Our system also provides navigation toolbars so that users can browse the history of entities and properties. For example, if the user clicks on a property, the interface will show a navigation window where the user can see the property type and its history and specify temporal conditions, as shown in Figure 2(b). The interface is implemented as middleware systems on Wikipedia.

Query Engine. In our system, the temporal RDF triples are stored using in-memory MVBT indices that support fast range query processing. Taking a SPARQL^T query, we first parse the query and generate an execution plan in which every SPARQL^T query pattern is converted into a query pattern (k, i) on MVBT to retrieve all the temporal RDF triples with keys in range k and intervals overlapping interval i. A hybrid method combining dictionary based compression and prefix encoding is adopted 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.

Our query engine is implemented in Java and evaluated on the Wikipedia Infobox History (38.5 million triples). We compare the query performance of our implementation with (i) the reification approach using Jena and (ii) the SQL-based approach using MySQL memory engine. The test query set contains 20 temporal selection and temporal join queries, as those in Figure 1. In our implementation, the average execution times for temporal selection and temporal join queries are 5 and 41 milliseconds respectively. On average, our approach is 1 order of magnitude faster than the SQL-based approach and 2 orders of magnitude faster than the reification approach. Interested readers are referred to [6] for more details on the implementation and experiments.

4 Demonstrating SPARQL^T

The conference participants attending our demo will be able to explore structured knowledge bases and their history in a simple and intuitive way. The demonstration starts with a brief introduction of our system, followed by a hands-on phase in which the participants can ask interesting queries on the history of people and places they are familiar with, and test the performance of our query engine. We prepare two real world datasets: Wikipeda Infobox History and GovTrack, with 38.5 million and 22 million temporal RDF triples respectively. A tutorial video is available in [2], as well as complete syntax of SPARQL^T and more query examples.

References

1. DBpedia Live. http://live.dbpedia.org/.
2. Example Queries & Tutorial Video. http://yellowstone.cs.ucla.edu/sparqlt-demo/index.html.
3. J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, Nov. 1983.
4. M. Atzori and C. Zaniolo. Swipe: Searching wikipedia by example. In *WWW*, pages 309–312, 2012.
5. B. Becker, S. Gschwind, et al. An asymptotically optimal multiversion b-tree. *VLDB*, 5(4):264–275, Dec. 1996.
6. S. Gao and C. Zaniolo. Sparqlt: A fast, user-friendly system for querying the history of rdf knowledge bases. CSD Technical Report #150004, UCLA, 2015.
7. F. Grandi. T-sparql: a tsql2-like temporal query language for rdf. *In International Workshop on on Querying Graph Structured Data*, pages 21–30, 2010.
8. C. Gutierrez, C. A. Hurtado, et al. Introducing time into rdf. *TKDE*, 19(2):207–218, 2007.
9. A. Pugliese, O. Udrea, et al. Scaling rdf with time. In *WWW*, pages 605–614, 2008.
10. J. Tappolet and A. Bernstein. Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In *ESWC*, pages 308–322, 2009.
11. X. Zhou, F. Wang, et al. Efficient temporal coalescing query support in relational database systems. In *DEXA*, pages 676–686, 2006.