T-SPARQL: A TSQL2-Like Temporal Query Language for RDF

Fabio Grandi

Alma Mater Studiorum – Università di Bologna, Italy
fabio.grandi@unibo.it

Abstract. In this paper, we present a temporal extension of the SPARQL query language for RDF graphs. The new language is based on a temporal RDF database model employing triple timestamping with temporal elements, which best preserves the scalability property enjoyed by triple storage technologies, especially in a multi-temporal setting. The proposed SPARQL extensions are aimed at embedding several features of the TSQL2 consensual language designed for temporal relational databases.

1 Introduction

When an RDF graph [11], representing for instance the specification of an ontology, is changed, some applications require the past version to be maintained in addition to the new one. This is the case of the legal domain where ontologies must evolve as a natural consequence of the dynamics involved in normative systems [4]. Agents in such a domain may often have to deal with a past perspective, like a Court having to judge today on a fact committed several years ago. Moreover, several time dimensions are usually important for computer applications in such domains [3].

In the design of semantics based information systems, triple store technology [12] based on the RDF data model is supposed to provide scalability for querying and retrieval. Several temporal extensions of the RDF data model have been proposed [6, 10, 17], often in conjunction with special index structures which allow for efficient processing of temporal queries (e.g. tGRIN [10] and keyTree [17]). In order to preserve the scalability property of the triple storage approach as much as possible also in the presence of temporal semantics, we introduced in [5] a temporal RDF data model aimed at preventing the proliferation of *value-equivalent* triples even in the presence of multiple temporal dimensions. In particular, this has been accomplished through the adoption of *temporal elements* [2, 7] as timestamps and a careful definition of the operational semantics of modification statements.

In this work, we complete the proposal in [5] by introducing a temporal extension of SPARQL [16] which can be used as query language for the temporal data model of [5]. The temporal extension is based on the lesson learned with the design of the temporal query language TSQL2 [14], from which it inherits part of the temporal expressiveness and user friendliness. The TSQL2 language

was born as a follow-up of the 1993 ARPA/NSF International Workshop on an Infrastructure for Temporal Databases [9], after which Richard Snodgrass sent an invitation to form a committee for the design of a consensual temporal extension of the standard database query language SQL-92. The committee, gathering 18 people from the academic and industrial worlds, started its works in July 1993 and, after joint email discussions and voting on every aspect of the language, produced a first draft in March 1994 [13]. The final specification and language commentaries appeared in a book published in 1995 [14] (for a more detailed history, see `http://www.cs.arizona.edu/people/rts/tsql2.html`).

The rest of paper is organized as follows. In Section 2, the main features of the T-SPARQL query language are introduced and the TSQL2 heritage is underlined. In Section 3, the use and functioning of the language is exemplified through the presentation of a few temporal queries. Conclusions will be finally found in Section 4.

2 T-SPARQL Definition

2.1 A multi-temporal RDF database model

We briefly recall here the base definitions of the underlying multi-temporal RDF database model [5], starting from an N-dimensional time domain

$$\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2 \times \cdots \times \mathcal{T}_N$$

where $\mathcal{T}_i = [0, \text{UC}]_i$ is the i-th time domain. Right-unlimited time intervals are expressed as $[t, \text{UC}]$, where UC means "Until Changed", though such a symbol is often used in temporal database literature [7] for transaction time only (whereas, e.g. "forever" or ∞ is used for valid time). Such naming choice refers to the modeling of time-varying data, which are potentially subject to change with respect to all the underlying time dimensions.

A multi-temporal RDF triple is defined as:

$$(s, p, o \,|\, T)$$

where s is a subject, p is a property, o is an object and $T \subseteq \mathcal{T}$ is a *timestamp* assigning a *temporal pertinence* to the RDF triple (s, p, o). We will also call the (non-temporal) triple (s, p, o) the value or the contents of the temporal triple $(s, p, o \,|\, T)$. The temporal pertinence of a triple is a subset of the multidimensional time domain which is represented by a *temporal element* [2, 7], that is a disjoint union of multidimensional temporal intervals, each one obtained as the Cartesian product of one time interval for each of the supported temporal dimensions:

$$T = \bigcup_{1 \le j \le m} \mathcal{I}_j = \bigcup_{1 \le j \le m} [t_j^s, t_j^e]_1 \times [t_j^s, t_j^e]_2 \times \cdots \times [t_j^s, t_j^e]_N$$

where the unioned N-dimensional intervals are all disjoint (i.e. $\mathcal{I}_j \cap \mathcal{I}_k = \varnothing$ for all $1 \le j < k \le m$).

A multi-temporal RDF database is defined as a set of timestamped RDF triples:

$$\text{RDF-TDB} = \{\, (s, p, o \,|\, T) \,|\, T \subseteq \mathcal{T} \,\}$$

with the integrity constraint:

$$\forall (s, p, o \,|\, T), (s', p', o' \,|\, T') \in \text{RDF-TDB}\text{: } s = s' \wedge p = p' \wedge o = o' \Longrightarrow T = T'$$

which requires that no value-equivalent distinct triples exist.

The adoption of timestamps made-up of temporal elements instead of (multi-temporal) simple intervals avoids the duplication of triples in the presence of a temporal pertinence with a complex shape. In fact, we store different triple versions only once with a complex timestamp rather than storing multiple copies of them with a simple timestamp as in [6, 10, 17]. The memory saving we obtain grows with the dimensionality of the time domain, but it can even be appreciated with a monodimensional time domain, when the temporal pertinence of a triple is not a convex interval. For example, the temporal triples $(s, p, o \,|\, [t_1, t_2])$ and $(s, p, o \,|\, [t_3, t_4])$, where $t_2 + 1 < t_3$, can be merged with temporal element timestamping into a single triple $(s, p, o \,|\, [t_1, t_2] \cup [t_3, t_4])$. Whereas the same space is basically required for globally storing the timestamps in both cases (i.e. the space needed by four time points), the space required for storing one occurrence of the triple contents (s, p, o) is saved in the latter case. Moreover, with element timestamping, according to the integrity constraint introduced above, no two temporal triples can have the same non-temporal contents and, thus, checking of uniqueness and of functional properties constraints can be performed more easily. As shown in [5], the semantics of modification operations can be defined in such a way that the integrity constraints concerning temporal elements are automatically preserved.

The controlled growth of value-equivalent triples made possible by temporal elements preserves, in a temporal setting, the scalability property of the triple storage approach. Furthermore, since temporal elements are closed under set union, intersection and complementation operations, they lead to query languages that are more natural [2].

2.2 Time representation and manipulation

As in the TSQL2 model, time is considered discrete, with a minimal system-dependent representation unit called *chronon* [7]. A mono-temporal chronon corresponds to an elementary interval on the time axis, whereas a multi-temporal chronon corresponds to a unit hypercube in the N-dimensional time domain (Cartesian product of one chronon per each dimension). As for SQL-92, three base temporal types have been defined for TSQL2 at the conceptual level: *datetime*, *period* and *interval*. The first one corresponds to an instantaneous event, without duration, which can be conventionally represented via a single chronon. The second one correspond to a set of consecutive chronons along the time axis and is characterized by two datetime constants which represent its boundaries. The third corresponds to a pure duration, non anchored on the time axis, and

can be represented as a multiple of the chronon. Whereas the first and the third temporal types correspond to the XML Schema primitive datatypes `xs:dateTime` and `xs:duration` [18], respectively, we assume for the second one a special data type `xs:period` has been defined and a constructor function:

```
fn:period($arg1 as xs:dateTime, $arg2 as xs:dateTime) as xs:period
```

is available to build a monodimensional time period from two `xs:dateTime` (or `xs:date` or `xs:time`) values representing the period boundaries. However, for the sake of simplicity, we will also use in the rest of the paper period literal expressions like:

```
"[2010-01-01,2010-01-31]"^^xs:period
```

as a shorthand for the use of the `xs:period` constructor like:

```
fn:period("2010-01-01"^^xs:date, "2010-01-31"^^xs:date)
```

We further assume compatibility between the `xs:period` datatype and the standard `gYearMonth` and `gYear` datatypes, which leads for example to the equivalences:

```
"[2010-01-01,2010-01-31]"^^xs:period = "2010-01"^^xs:gYearMonth
```

and:

```
"[2009-01-01,2009-12-31]"^^xs:period = "2009"^^xs:gYear
```

Like in TSQL2, the `xs:period` datatype is equipped with two built-in functions:

```
fn:begin($arg1 as xs:period) as xs:dateTime
fn:end($arg1 as xs:period) as xs:dateTime
```

to extract the left and right period boundaries, respectively. For instance:

```
fn:begin("[2010-01-01,2010-01-31]"^^xs:period) = "2010-01-01"^^xs:date
```

and:

```
fn:end("2009"^^xs:gYear) = "2009-12-31"^^xs:date
```

Without entering into details, we assume a suitable datatype `xs:temporalElement`, equipped with constructor and built-in functions, is available to manage temporal elements. In particular, like in TSQL2, we consider the two built-in functions:

```
fn:first($arg1 as xs:temporalElement) as xs:period
fn:last($arg1 as xs:temporalElement) as xs:period
```

to have been defined to extract the first and last period from a monodimensional temporal element, respectively. For instance:

```
"[2008-06-01,2009-07-15]+[2009-11-01,2010-02-21]"
```

is a valid `xs:temporalElement` literal and

```
fn:first("[2008-06-01,2009-07-15]+[2009-11-01,2010-02-21]")
```

yields `"[2008-06-01,2009-07-15]"^^xs:period`. In order to extract the first (last) chronon of a temporal element, we assume that the `fn:begin()` (`fn:end()`) function can also be directly applied to temporal elements (e.g. if T is a temporal element, `fn:begin(`T`) = fn:begin(fn:first(`T`))`).

Other functions and operators defined for XQuery and XPath [19] to manipulate time and duration datatypes are assumed to be available. We further assume that, as it happens for TSQL2, casting from another temporal datatype to a duration can be used to calculate the overall duration of a time period or element by means of, for instance, type constructor functions:

```
xs:duration($arg1 as xs:period) as xs:duration
xs:duration($arg1 as xs:temporalElement) as xs:duration
```

For example, the expression:

```
xs:yearMonthDuration("[2009-02,2009-07]+[2009-11,2010-03]")
```

which can also be written using the explicit `cast` operator as:

```
"[2009-02,2009-07]+[2009-11,2010-03]"^^xs:temporalElement
  cast as xs:yearMonthDuration
```

yields 11 (months).

The TSQL2 language, which is based on a bitemporal data model, provides for functions to access the valid and transaction time components from a timestamp. Likewise, we assume similar functions to be available also for T-SPARQL: if T is a multi-dimensional time element, the expressions:

`VALID(`T`)`
`TRANSACTION(`T`)`

can thus be used to express conditions on the valid and transaction time components of T, respectively. Similar functions can be defined for other time dimensions in a multi-temporal setting (e.g. `EFFICACY()` for *efficacy time* [3, 5]). Notice that such functions do not imply a projection onto the corresponding time axis with reduction of dimensionality of the timestamp: all the conditions expressed on different time dimensions are to be matched concurrently against the same multi-dimensional intervals composing the temporal element. For instance, if $T = [10, 20]_t \times [30, 40]_v \cup [30, 40]_t \times [10, 20]_v$, then T does not qualify for the selection condition $12 \in \texttt{TRANSACTION}(T) \wedge 17 \in \texttt{VALID}(T)$.

2.3 Temporal selection

Temporal selection is the most qualifying feature of a temporal query language, as it allows to select data on the basis of their temporal properties. In order to add temporal selection capabilities to the SPARQL language, we extend the syntax of the basic graph pattern in the `WHERE` clause of the `SELECT` statement. As RDF triples are correspondingly augmented with the timestamp in the data model, graph patterns to be used in the T-SPARQL `WHERE` clause are extended with an optional fourth position where matching with the triple timestamps can be specified. For instance, in the graph pattern

```
_:e ex:Dept "Toys" | ?t
```

the variable `?t` binds to the timestamp of a temporal triple representing the fact that an employee denoted by the blank node `_:e` has been working in the Toys department. This syntax, matching the $(s, p, o \,|\, t)$ triple structure in the temporal RDF data model, seems more natural than introducing a distinguished variable type to denote timestamps (like in [8]). If the fourth position (along with the " | " separator) in the pattern is not used, that is a standard SPARQL three-position pattern is used, the matching with a temporal triple is made regardless of its timestamp.

In the T-SPARQL `FILTER` clause, TSQL2 temporal (binary infix) predicates can be used, with the same semantics, to specify constraints over timestamp variables. For instance, the clause:

```
FILTER ( VALID(?t) CONTAINS "2009-06-01"^^xs:date )
```

only matches the timestamps bound to `?t` whose (valid time) value contains the date 2009, June 1st. The available comparison operators are the following:

Operator	Definition
A `PRECEDES` B	`END`(A) is earlier than `BEGIN`(B)
$A = B$	A and B are identical (i.e. contain the same chronons)
A `OVERLAPS` B	the intersection of A and B is not empty
A `MEETS` B	`END`(A) immediately precedes `BEGIN`(B)
A `CONTAINS` B	each chronon in B is also contained in A

They can be used to compare (monodimensional) temporal elements, periods and time points. Since all temporal types can be reduced to sets of chronons, such operators can also be used to compare operands with different temporal types [14]. For instance, if A is an element and B is a period, then the expression "A `PRECEDES` B" is true if the last chronon belonging to A precedes the left boundary of B. All the comparison operators can be implemented on the basis of a primitive operator "$Before()$" which defines the relation order on the time axis. It can also be easily checked that such operators guarantee the *temporal completeness* of the resulting language, as they allow users to check the occurrence of all the possible relationships between two periods or events [1]. Such operator set has been chosen for TSQL2 also considering the user-friendliness of the language among the design principle. This lead to a non minimal set of comparison operators which are closer to their meaning in natural language than the artificial definition of operators which equip other temporal languages (e.g. based on Allen's algebra [1]).

The `OVERLAPS` and `CONTAINS` operators can also be defined to work on multidimensional timestamps in a straightforward way.

2.4 Temporal projection

Temporal projection is the operation which specifies the value of the timestamps to be assigned to the retrieved data. TSQL2 supports a `VALID` clause to specify valid-time projection, as the transaction time assigned to query results is always the current time and cannot be changed by the user. In T-SPARQL, temporal projection is only relevant where the query result has to be a temporal RDF graph consistent with the underlying data model. Otherwise, temporal data can simply be mixed with other data by putting temporal variables in the target list (e.g. see the first example in the next section). In the former case, the basic and most important operation is the construction of a new (temporal) RDF graph as a temporally consistent subset extracted from the multi-version RDF database, which takes the form of a *snapshot query* or *timeslice query* [7].

Snapshot queries are used to extract a single temporal version from a multi-version RDF graph. For instance, if the temporal RDF database encodes the definition of a multi-version ontology, the result of a snapshot query is a standard (non-temporal) RDF graph, which can be interpreted as a consistent single

ontology version valid at a given time point. Given a multidimensional time point $\bar{t} = (t_1, t_2, \ldots, t_N) \in \mathcal{T}$, we can define the snapshot valid at $\bar{t}$ as:

$$\text{RDF-TDB}(\bar{t}) = \{\, (s,p,o) \mid (s,p,o \,|\, T) \in \text{RDF-TDB} \wedge \bar{t} \in T \}$$

In T-SPARQL, the snapshot query above could be expressed via the following statement:

```
CONSTRUCT { ?s ?p ?o }
WHERE { TGRAPH <http://myExample.org/tGraph> { ?s ?p ?o | ?t } .
        FILTER ?t CONTAINS "(t1, t2, ..., tN)" . }
```

where the URI `http://myExample.org/tGraph` denotes a multi-temporal RDF triple store. The results make up a non temporal RDF graph.

Timeslice queries are used to extract a temporally consistent set of consecutive temporal versions from a multi-version RDF graph. If the temporal RDF database encodes the definition of a multi-version ontology, the result of a timeslice query is a temporal RDF graph, which can be interpreted as the collection of all the temporally consistent ontology versions valid in a given period. Given a multi-dimensional time period $\bar{\mathcal{I}} = I_1 \times I_2 \times \cdots \times I_N \subseteq \mathcal{T}$, we can define the timeslice valid in $\bar{\mathcal{I}}$ as:

$$\text{RDF-TDB}(\bar{\mathcal{I}}) = \{\, (s,p,o \,|\, T') \mid (s,p,o \,|\, T) \in \text{RDF-TDB} \wedge T' = T \cap \bar{\mathcal{I}} \neq \varnothing \}$$

In T-SPARQL, the snapshot query above could be expressed by means of the statement which follows:

```
TCONSTRUCT { ?s ?p ?o | INTERSECT( ?t, "I1 × I2 × ··· × IN" ) . }
WHERE { TGRAPH <http://myExample.org/tGraph> { ?s ?p ?o | ?t } }
```

The timestamps assigned to the triples in the result are computed as the intersection of the timestamp of the retrieved triples with the query period $\bar{\mathcal{I}}$. If the intersection is empty, the triple does not contribute to the results (and, thus, no additional temporal selection conditions are required in the `WHERE` clause).

Within the `WHERE` clause that can be added to such `CONSTRUCT` or `TCONSTRUCT` statements, temporal and non-temporal selection (via graph pattern matching) can be combined with temporal projection in a single statement.

3 Query Examples

Due to space limitations, the syntax and semantics of the T-SPARQL language is just illustrated by means of a few example queries shown in this section. We assume that `ex:` is a prefix referencing a namespace involving the definition of employee data:

```
@prefix ex: <http://myExample.org/employee/> .
```

The following is an example of query involving both temporal selection and projection, although the result is not organized as a temporal RDF graph:

```
SELECT ?salary INTERSECT(?t,"[2007-01-01,2009-12-31]") WHERE {
  ?emp rdf:type ex:emp ;
       ex:Name "Tom" ;
       ex:Salary ?salary | ?t .
  FILTER ( VALID(?t) OVERLAPS "[2007-01-01,2009-12-31]"^^xs:period ) . }
```

The (current) history of the the Tom's salary from 2007 to 2009 is retrieved. Following the same rule of TSQL2, a default `TRANSACTION(?t) CONTAINS fn:current- date()` conjunct in the `FILTER` clause is implied and, thus, can always be omitted when we are interested in current data. Indeed, an explicit condition involving transaction time must be specified when we want to roll-back the RDF database to a past point in time, as in the query:

```
SELECT ?salary INTERSECT(?t,"[2007-01-01,2009-12-31]") WHERE {
   ?emp rdf:type ex:emp ;
        ex:Name "Tom" ;
        ex:Salary ?salary | ?t .
   FILTER ( VALID(?t) OVERLAPS "[2007-01-01,2009-12-31]"^^xs:period
            && TRANSACTION(?t) CONTAINS "2008-01-01"^^xs:date ) . }
```

which retrieves the history of the the Tom's salary from 2007 to 2009, as of the beginning of 2008.

The query which follows, which retrieves the name of the employees who have worked in the Toys department longer than Ann has made $20,000, performs a sort of temporal join involving durations between two employees' data:

```
SELECT ?ename WHERE {
   ?emp1 rdf:type ex:emp ;
         ex:Name "Ann" ;
         ex:Salary ?salary | ?ts .
   ?emp2 rdf:type ex:emp ;
         ex:Name ?ename ;
         ex:Dept "Toys" | ?tt .
   FILTER ( ?salary > 20000
            && xs:duration(VALID(?tt)) > xs:duration(VALID(?ts)) ) . }
```

An optional modifier `PERIOD`, which corresponds to the *partitioning unit* which can be associated to tuple variables in the `FROM` clause of TSQL2 [15], can be specified in the declaration of temporal variables. When used, the timestamp bound to the variable is partitioned into maximal periods over which the variable ranges, yielding triple timestamping with periods. As many queries are interested in maximal periods, being able to partition a temporal element into such periods is highly useful. For instance, the query:

```
SELECT ?ename WHERE {
   ?emp rdf:type ex:emp ;
        ex:Name ?ename ;
        ex:Dept "Sales" | ?t .
   FILTER ( xs:duration(VALID(?t)) > "P2Y"^^xs:duration ) . }
```

which retrieves the names of the employees who worked in the Sales department for more than two years (altogether), can be modified as follows:

```
SELECT ?ename WHERE {
   ?emp rdf:type ex:emp ;
        ex:Name ?ename ;
        ex:Dept "Sales" | ?t PERIOD .
   FILTER ( xs:duration(VALID(?t)) > "P2Y"^^xs:duration ) . }
```

to retrieve the names of the employees who worked continuously in the Sales department for a period longer than two years. Hence an employee who worked in the Sales department from January 2006 to July 2007 and from March 2009 to April 2010 qualifies for the former query but not for the latter, since he/she worked in Sales for 20 months altogether but at most for 17 consecutive months. This powerful tool also allows us to reference consecutive periods within the same data history as in the following query:

```
SELECT ?ename ?job WHERE {
   ?emp rdf:type ex:emp ;
        ex:Name ?ename ;
        ex:Job ?job | ?t1 PERIOD ;
        ex:Job "Director" | ?t2 PERIOD ;
        ex:Job ?job | ?t3 PERIOD .
   FILTER ( VALID(?t1) MEETS VALID(?t2)
            && VALID(?t2) MEETS VALID(?t3) ) . }
```

which retrieves the name of the employees who returned to their previous job (which is also retrieved) after having been directors for some time.

4 Conclusions

In this paper, we presented T-SPARQL, a temporal SPARQL extension suited to the temporal RDF database model employing triple timestamping with temporal elements introduced in [5], which best preserves in the multi-temporal setting the scalability property enjoyed by triple storage technologies.

The language T-SPARQL is equipped with the basic temporal constructs which have been designed for the well-known TSQL2 relational query language [14] and work with an extended set of the temporal datatypes, functions and operators already present in the SPARQL specification [16, 18, 19]. Advanced TSQL2 features (e.g. involving temporal aggregates, granularities, indeterminacy) could also easily be added to the T-SPARQL specification, provided that they can be supported by an underlying query engine.

In future research, we will consider the design and implementation of a query engine supporting the execution of T-SPARQL queries (possibly via the extension of a canonical SPARQL engine) and the adoption of suitable index and storage structures to facilitate the execution of T-SPARQL queries on temporal RDF graphs.

References

1. J.F. Allen. Maintaining Knowledge about Temporal Intervals, *Communications of the ACM*, 26(11):832–843, 1983.
2. S. Gadia. A homogeneous relational model and query language for temporal databases, *ACM Transactions on Database Systems*, 13(3):418–448, 1998.
3. F. Grandi, F. Mandreoli, and P. Tiberio. Temporal modelling of normative documents in XML format. *Data & Knowledge Engineering*, 54:327–354, 2005.

4. F. Grandi, and M.R. Scalas. The Valid Ontology: A simple OWL temporal versioning framework. In *Proc. of SEMAPRO Conf.*. IEEE Computer Society, 2009.
5. F. Grandi. Multi-temporal RDF Ontology Versioning. In *Proc. of IWOD Workshop*. CEUR-WS, 2009.
6. C. Gutierrez, C. Hurtado and A. Vaisman. Introducing time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2007.
7. C.S. Jensen, C.E. Dyreson (eds.), M. Böhlen, J. Clifford, R. Elmasri, S.K. Gadia, F. Grandi, P. Hayes, S. Jajodia, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J.F. Roddick, N.L. Sarda, M R. Scalas, A. Segev, R.T. Snodgrass, M.D. Soo, A. Tansel, P. Tiberio, and G. Wiederhold. The consensus glossary of temporal database concepts - February 1998 version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases — Research and Practice*. Springer-Verlag, 1998. LNCS No. 1399.
8. M. Perry, A.P. Sheth, and P. Jain. SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries. Tech.Rep. KNOESIS-TR-09-01. Kno.e.sis Center, `http://knoesis.org/students/prateek/sparql-st-www09-tr.pdf`
9. N. Pissinou, R.T. Snodgrass, R. Elmasri, I.S. Mumick, M.T. Özsu, B. Pernici, A. Segev, B. Theodoulidis and U. Dayal. Towards an Infrastructure for temporal Databases: Report of an Invitational ARPA/NSF Workshop. *ACM SIGMOD Record* 23(1):35–51, 1994.
10. A. Pugliese, O. Udrea, and V.S. Subrahmanian. Scaling RDF with Time. In *Proc. of WWW Conf.*. ACM Press, 2008.
11. Resource description framework. W3C Consortium, `http://www.w3.org/RDF/`.
12. K. Rohloff, M. Dean, I. Emmons, D. Ryder and J. Summer. An evaluation of triple-store technologies for large data stores. In *Proc. of OTM Workshops*. Springer-Verlag, 2007. LNCS No. 4806.
13. R.T. Snodgrass, I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y.C. Leung, N. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada. TSQL2 Language Specification. *ACM SIGMOD Record* 23(1):65–86, 1994.
14. R.T. Snodgrass (ed.), I. Ahn, G. Ariav, D. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T.Y. Cliff Leung, N. Lorentzos, R. Ramakrishnan, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, 1995.
15. R.T. Snodgrass, C.S. Jensen, and F. Grandi. The From Clause. In [14], Ch. 12.
16. SPARQL query language for RDF. W3C Consortium, `http://www.w3.org/TR/rdf-sparql-query/`.
17. J. Tappolet, and A. Bernstein. Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL. In *Proc. of ESWC Conf.*. Springer-Verlag, 2009. LNCS No. 5554.
18. XML Schema Part 2: Datatypes. W3C Consortium, `http://www.w3.org/TR/xmlschema-2/`.
19. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Consortium, `http://www.w3.org/TR/xpath-functions/`.