Representation and Querying of Valid Time of Triples in Linked Geospatial Data ⋆

Konstantina Bereta, Panayiotis Smeros, and Manolis Koubarakis

National and Kapodistrian University of Athens, Greece
{Konstantina.Bereta, psmeros, koubarak}@di.uoa.gr

Abstract. We introduce the temporal component of the stRDF data model and the stSPARQL query language, which have been recently proposed for the representation and querying of linked geospatial data that changes over time. With this temporal component in place, stSPARQL becomes a very expressive query language for linked geospatial data, going beyond the recent OGC standard GeoSPARQL, which has no support for valid time of triples. We present the implementation of the stSPARQL temporal component in the system Strabon, and study its performance experimentally. Strabon is shown to outperform all the systems it has been compared with.

1 Introduction

The introduction of time in data models and query languages has been the subject of extensive research in the field of relational databases [6, 20]. Three distinct kinds of time were introduced and studied: *user-defined* time which has no special semantics (e.g., January 1st, 1963 when John has his birthday), *valid* time which is the time an event takes place or a fact is true in the application domain (e.g., the time 2000-2012 when John is a professor) and *transaction* time which is the time when a fact is current in the database (e.g., the system time that gives the exact period when the tuple representing that John is a professor from 2000 to 2012 is current in the database). In these research efforts, many temporal extensions to SQL92 were proposed, leading to the query language TSQL2, the most influential query language for temporal relational databases proposed at that time [20].

However, although the research output of the area of temporal relational databases has been impressive, TSQL2 did not make it into the SQL standard and the commercial adoption of temporal database research was very slow. It is only recently that commercial relational database systems started offering SQL extensions for temporal data, such as IBM DB2, Oracle Workspace manager, and Teradata [2]. Also, in the latest standard of SQL (SQL:2011), an important new feature is the support for valid time (called *application time*) and transaction time. Each SQL:2011 table is allowed to have at most two *periods* (one for

⋆ This work was supported in part by the European Commission project TELEIOS (http://www.earthobservatory.eu/)

application time and one for transaction time). A *period* for a table T is defined by associating a user-defined name e.g., EMPLOYMENT_TIME (in the case of application time) or the built-in name SYSTEM_TIME (in the case of transaction time) with two columns of T that are the start and end times of the period (a closed-open convention for periods is followed). These columns must have the same datatype, which must be either DATE or a timestamp type (i.e., no new period datatype is introduced by the standard). Finally, the various SQL statements are enhanced in minimal ways to capture the new temporal features.

Compared to the relational database case, little research has been done to extend the RDF data model and the query language SPARQL with temporal features. Gutierrez et al. [8, 9] were the first to propose a formal extension of the RDF data model with valid time support. They also introduce the concept of *anonymous timestamps* in general temporal RDF graphs, i.e., graphs containing quads of the form $(s, p, o)[t]$ where t is a timestamp or an anonymous timestamp x stating that the triple (s, p, o) is valid in some unknown time point x. The work described in [11] subsequently extends the concept of general temporal RDF graphs of [9] to express temporal constraints involving anonymous timestamps. In the same direction, Lopes et al. integrated valid time support in the general framework that they have proposed in [15] for annotating RDF triples. Similarly, Tappolet and Bernstein [22] have proposed the language τ-SPARQL for querying the valid time of triples, showed how to transform τ-SPARQL into standard SPARQL (using named graphs), and briefly discussed an index that can be used for query evaluation. Finally, Perry [19] proposed an extension of SPARQL, called SPARQL-ST, for representing and querying spatiotemporal data. The main idea of [19] is to incorporate geospatial information to the temporal RDF graph model of [9]. The query language SPARQL-ST adds two new types of variables, namely spatial and temporal ones, to the standard SPARQL variables. Temporal variables (denoted by a # prefix) are mapped to time intervals and can appear in the fourth position of a quad as described in [9]. In SPARQL-ST two special filters are introduced: `SPATIAL FILTER` and `TEMPORAL FILTER`. They are used to filter the query results with spatial and temporal constraints (OGC Simple Feature Access topological relations and distance for the spatial part, and Allen's interval relations [3] for the temporal part).

Following the ideas of Perry [19], our group proposed a formal extension of RDF, called stRDF, and the corresponding query language stSPARQL for the representation and querying of temporal and spatial data using linear constraints [13]. stRDF and stSPARQL were later redefined in [14] so that geometries are represented using the Open Geospatial Consortium standards Well-Known-Text (WKT) and Geography Markup Language (GML). Both papers [13] and [14] mention very briefly the temporal dimension of stRDF and do not go into details. Similarly, the version of the system Strabon presented in [14], which implements stRDF and stSPARQL, does not implement the temporal dimension of this data model and query language. In this paper we remedy this situation by introducing all the details of the temporal dimension of stRDF and stSPARQL and implementing it in Strabon.

The original contributions of this paper are the following. We present in detail, for the first time, the valid time dimension of the data model stRDF and the query language stSPARQL. Although the valid time dimension of stRDF and stSPARQL is in the spirit of [19], it is introduced in a language with a much more mature geospatial component based on OGC standards [14]. In addition, the valid time component of stSPARQL offers a richer set of functions for querying valid times than the ones in [19]. With the temporal dimension presented in this paper, stSPARQL also becomes more expressive than the recent OGC standard GeoSPARQL [1]. While stSPARQL can represent and query geospatial data that changes over time, GeoSPARQL only supports static geospatial data.

We discuss our implementation of the valid time component of stRDF and stSPARQL in Strabon. We evaluate the performance of our implementation on two large real-world datasets and compare it to three other implementations: (i) a naive implementation based on the native store of Sesame which we extended with valid time support, (ii) AllegroGraph, which, although it does not offer support for valid time of triples explicitly, it allows the definition of time instants and intervals and their location on a time line together with a rich set of functions for writing user queries, and (iii) the Prolog-based implementation of the query language AnQL[1], which is the only available implementation with explicit support for valid time of triples. Our results show that Strabon outperforms all other implementations.

This paper is structured as follows. In Section 2 we introduce the temporal dimension of the data model stRDF and in Section 3 we present the temporal features of the query language stSPARQL. In Section 4 we describe how we extended the system Strabon with valid time support. In Section 5 we evaluate our implementation experimentally and compare it with other related implementations. In Section 6 we present related work in this field. Section 7 concludes this paper.

2 Valid Time Representation in the Data Model stRDF

In this section we describe the valid time dimension of the data model stRDF presented in [14]. The *time line* assumed is the (discrete) value space of the datatype `xsd:dateTime` of XML-Schema. Two kinds of time primitives are supported: time instants and time periods. A *time instant* is an element of the time line. A *time period* (or simply period) is an expression of the form $[B,E)$, $(B,E]$, (B,E), or $[B,E]$ where B and E are time instants called the *beginning* and the *ending* of the period respectively. Since the time line is discrete, we often assume only periods of the form $[B,E)$ with no loss of generality. Syntactically, time periods are represented by literals of the new datatype `strdf:period` that we introduce in stRDF. The value space of `strdf:period` is the set of all time periods covered by the above definition. The lexical space of `strdf:period` is trivially defined from the lexical space of `xsd:dateTime` and the closed/open pe-

[1] `http://anql.deri.org/`

riod notation introduced above. Time instants can also be represented as closed periods with the same beginning and ending time.

Values of the datatype `strdf:period` can be used as objects of a triple to represent *user-defined time*. In addition, they can be used to represent *valid times* of temporal triples which are defined as follows. A *temporal triple (quad)* is an expression of the form `s p o t.` where `s p o.` is an RDF triple and `t` is a time instant or a time period called the *valid time* of a triple. An *stRDF graph* is a set of triples and temporal triples. In other words, some triples in an stRDF graph might not be associated with a valid time.

We also assume the existence of temporal constants `NOW` and `UC` inspired from the literature of temporal databases [5]. `NOW` represents the current time and can appear in the beginning or the ending point of a period. It will be used in stSPARQL queries to be introduced in Section 3. `UC` means "Until Changed" and is used for introducing valid times of a triple that persist until they are explicitly terminated by an update. For example, when John becomes an associate professor in 1/1/2013 this is assumed to hold in the future until an update terminates this fact (e.g., when John is promoted to professor).

Example 1. The following stRDF graph consists of temporal triples that represent the land cover of an area in Spain for the time periods [2000, 2006) and [2006, UC) and triples which encode other information about this area, such as its code and the WKT serialization of its geometry extent. In this and following examples, namespaces are omitted for brevity. The prefix `strdf` stands for `http://strdf.di.uoa.gr/ontology` where one can find all the relevant datatype definitions underlying the model stRDF.

```
corine:Area_4 rdf:type corine:Area .
corine:Area_4 corine:hasID "EU-101324" .
corine:Area_4 corine:hasLandCover corine:coniferousForest
              "[2000-01-01T00:00:00,2006-01-01T00:00:00)"^^strdf:period .
corine:Area_4 corine:hasLandCover corine:naturalGrassland
              "[2006-01-01T00:00:00,UC)"^^strdf:period .
corine:Area_4 corine:hasGeometry "POLYGON((-0.66 42.34, ...))"^^strdf:WKT .
```

The stRDF graph provided above is written using the N-Quads format[2] which has been proposed for the general case of adding context to a triple. The graph has been extracted from a publicly available dataset provided by the European Environmental Agency (EEA) that contains the changes in the CORINE Land Cover dataset for the time period [2000, UC) for various European areas. According to this dataset, the area `corine:Area_4` has been a coniferous forest area until 2006, when the newer version of CORINE showed it to be natural grassland. Until the CORINE Land cover dataset is updated, `UC` is used to denote the persistence of land cover values of 2006 into the future. The last triple of the stRDF graph gives the WKT serialization of the geometry of the area (not all vertices of the polygon are shown due to space considerations). This dataset will be used in our examples but also in the experimental evaluation of Section 5.

[2] http://sw.deri.org/2008/07/n-quads/

3 Querying Valid Times Using stSPARQL

The query language stSPARQL is an extension of SPARQL 1.1. Its geospatial features have been presented in [12] and [14]. In this section we introduce for the first time the valid time dimension of stSPARQL. The new features of the language are:
Temporal Triple Patterns. Temporal triple patterns are introduced as the most basic way of querying temporal triples. A *temporal triple pattern* is an expression of the form `s p o t.`, where `s p o.` is a triple pattern and `t` is a time period or a variable.
Temporal Extension Functions. Temporal extension functions are defined in order to express temporal relations between expressions that evaluate values of the datatypes `xsd:dateTime` and `strdf:period`. The first set of such temporal functions are 13 Boolean functions that correspond to the 13 binary relations of Allen's Interval Algebra. stSPARQL offers nine functions that are "syntactic sugar" i.e., they encode frequently-used disjunctions of these relations.

There are also three functions that allow relating an instant with a period:

- `xsd:Boolean strdf:during(xsd:dateTime i2, strdf:period p1)`: returns true if instant `i2` is during the period `p1`.
- `xsd:Boolean strdf:before(xsd:dateTime i2, strdf:period p1)`: returns true if instant `i2` is before the period `p1`.
- `xsd:Boolean strdf:after(xsd:dateTime i2, strdf:period p1)`: returns true if instant `i2` is after the period `p1`.

The above point-to-period relations appear in [16]. The work described in [16] also defines two other functions allowing an instant to be equal to the starting or ending point of a period. In our case these can be expressed using the SPARQL 1.1. operator = (for values of `xsd:dateTime`) and functions `period_start` and `period_end` defined below.

Furthermore, stSPARQL offers a set of functions that construct new (closed-open) periods from existing ones. These functions are the following:

- `strdf:period strdf:period_intersect(period p1, period p2)`: This function is defined if `p1` intersects with `p2` and it returns the intersection of period `p1` with period `p2`.
- `strdf:period strdf:period_union(period p1, period p2)`: This function is defined if period `p1` intersects `p2` and it returns a period that starts with `p1` and finishes with `p2`.
- `strdf:period strdf:minus(period p1, period p2)`: This function is defined if periods `p1` and `p2` are related by one of the Allen's relations `overlaps, overlappedBy, starts, startedBy, finishes, finishedBy` and it returns the a period that is constructed from period `p1` with its common part with `p2` removed.
- `strdf:period strdf:period(xsd:dateTime i1, xsd:dateTime i2)`: This function constructs a (closed-open) period having instant `i1` as beginning and instant `i2` as ending time.

There are also the functions `strdf:period_start` and `strdf:period_end` that take as input a period p and return an output of type `xsd:dateTime` which is the beginning and ending time of the period `p` respectively.

Finally, stSPARQL defines the following functions that compute temporal aggregates:

- `strdf:period strdf:intersectAll(set of period p)`: Returns a period that is the intersection of the elements of the input set that have a common intersection.
- `strdf:period strdf:maximalPeriod(set of period p)`: Constructs a period that begins with the smallest beginning point and ends with the maximum endpoint of the set of periods given as input.

The query language stSPARQL, being an extension of SPARQL 1.1, allows the temporal extension functions defined above in the SELECT, FILTER and HAVING clause of a query. A complete reference of the temporal extension functions of stSPARQL is available on the Web[3].

Temporal Constants. The temporal constants `NOW` and `UC` can be used in queries to retrieve triples whose valid time has not ended at the time of posing the query or we do not know when it ends, respectively.

The new expressive power that the valid time dimension of stSPARQL adds to the version of the language presented in [14], where only the geospatial features were presented, is as follows. First, a rich set of temporal functions are offered to express queries that refer to temporal characteristics of some non-spatial information in a dataset (e.g., see Examples 2, 3 and 6 below). In terms of expressive power, the temporal functions of stSPARQL offer the expressivity of the qualitative relations involving points and intervals studied by Meiri [16]. However, we do not have support (yet) for quantitative temporal constraints in queries (e.g., $T_1 - T_2 \leq 5$). Secondly, these new constructs can be used together with the geospatial features of stSPARQL (geometries, spatial functions, etc.) to express queries on geometries that change over time (see Examples 4 and 5 below). The temporal and spatial functions offered by stSPARQL are orthogonal and can be combined with the functions offered by SPARQL 1.1 in arbitrary ways to query geospatial data that changes over time (e.g., the land cover of an area) but also moving objects [10] (we have chosen not to cover this interesting application in this paper).

In the rest of this section, we give some representative examples that demonstrate the expressive power of stSPARQL.

Example 2. Temporal selection and temporal constants. Return the current land cover of each area mentioned in the dataset.

```
SELECT ?clcArea ?clc
WHERE {?clcArea rdf:type corine:Area;
               corine:hasLandCover ?clc ?t . FILTER(strdf:during(NOW, ?t))}
```

This query is a temporal selection query that uses an extended Turtle syntax that we have devised to encode temporal triple patterns. In this extended syntax, the

[3] `http://www.strabon.di.uoa.gr/stSPARQL`

fourth element is optional and it represents the valid time of the triple pattern. The temporal constant `NOW` is also used.

Example 3. Temporal selection and temporal join. Give all the areas that were forests in 1990 and were burned some time after that time.

```
SELECT ?clcArea
WHERE{?clcArea rdf:type corine:Area ;
      corine:hasLandCover corine:ConiferousForest ?t1 ;
      corine:hasLandCover corine:BurnedArea ?t2 ;
      FILTER(strdf:during(?t1, "1990-01-01T00:00:00"^^xsd:dateTime) && strdf:after(?t2,?t1))}
```

This query shows the use of variables and temporal functions to join information from different triples.

Example 4. Temporal join and spatial metric function. Compute the area occupied by coniferous forests that were burnt at a later time.

```
SELECT ?clcArea (SUM(strdf:area(?geo)) AS ?totalArea)
WHERE {?clcArea rdf:type corine:Area;
                corine:hasLandCover corine:coniferousForest ?t1 ;
                corine:hasLandCover corine:burntArea ?t2 ;
                corine:hasGeometry ?geo .
       FILTER(strdf:before(?t1,?t2))} GROUP BY ?clcArea
```

In this query, a temporal join is performed by using the temporal extension function `strdf:before` to ensure that areas included in the result set were covered by coniferous forests *before* they were burnt. The query also uses the spatial metric function `strdf:area` in the SELECT clause of the query that computes the area of a geometry. The aggregate function `SUM` of SPARQL 1.1 is used to compute the total area occupied by burnt coniferous forests.

Example 5. Temporal join and spatial selection. Return the evolution of the land cover use of all areas contained in a given polygon.

```
SELECT ?clc1 ?t1 ?clc2 ?t2
WHERE {?clcArea rdf:type corine:Area ;
                corine:hasLandCover ?clc1 ?t1 ; corine:hasLandCover ?clc2 ?t2 ;
                clc:hasGeometry ?geo .
       FILTER(strdf:contains(?geo, "POLYGON((-0.66 42.34, ...))"^^strdf:WKT)
       FILTER(strdf:before(?t1,?t2))}
```

The query described above performs a temporal join and a spatial selection. The spatial selection checks whether the geometry of an area is contained in the given polygon. The temporal join is used to capture the temporal evolution of the land cover in pairs of periods that preceed one another .

Example 6. Update statement with temporal joins and period constructor.

```
UPDATE {?area corine:hasLandCover ?clcArea ?coalesced}
WHERE {SELECT (?clcArea AS ?area) ?clcArea (strdf:period_union(?t1,?t2) AS ?coalesced)
       WHERE {?clcArea rdf:type corine:Area ;
                      corine:hasLandCover ?clcArea ?t1; corine:hasLandCover ?clcArea ?t2 .
              FILTER(strdf:meets(?t1,?t2) || strdf:overlaps(?t1,?t2))}}
```

In this update, we perform an operation called *coalescing* in the literature of temporal relational databases: two temporal triples with exactly the same subject, predicate and object, and periods that overlap or meet each other can be “joined” into a single triple with valid time the union of the periods of the original triples [4].

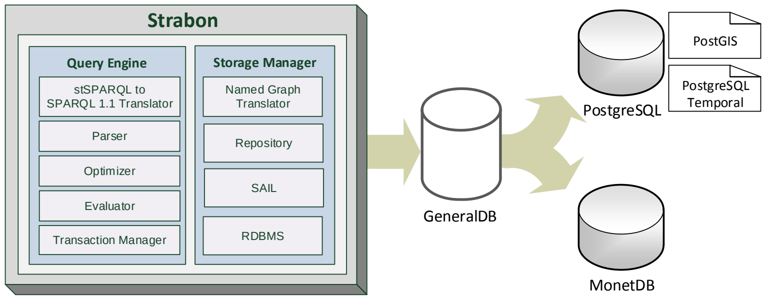


Fig. 1. Architecture of the system Strabon enhanced with valid time support

4 Implementation of Valid Time Support in Strabon

Figure 1 shows the architecture of the system Strabon presented in [14], as it has been extended for valid time support. We have added new components and extended existing ones as we explain below.

As described in [14], Strabon has been implemented by extending Sesame[4] 2.6.3 and using an RDBMS as a backend. Currently, PostgreSQL and MonetDB can be used as backends. To support the geospatial functionality of stSPARQL efficiently as we have shown in [14], Strabon uses PostGIS, an extension of PostgreSQL for storing and querying spatial objects and evaluating spatial operations. To offer support for the valid time dimension of stSPARQL discussed in this paper, the following new components have been added to Strabon.

Named Graph Translator. This component is added to the storage manager and translates the temporal triples of stRDF to standard RDF triples following the named graphs approach of [22] as we discuss below.

stSPARQL to SPARQL 1.1 Translator. This component is added to the query engine so that temporal triple patterns are translated to triple patterns as we discuss below.

PostgreSQL Temporal. This is a temporal extension of PostgreSQL which defines a `PERIOD` datatype and implements a set of temporal functions. This datatype and its associated functions come very handy for the implementation of the valid time suport in Strabon as we will see below. PostgreSQL Temporal also allows the use of a GiST index on `PERIOD` columns. Using this add-on, PostgreSQL becomes "temporally enabled" as it adds support for storing and querying `PERIOD` objects and for evaluating temporal functions.

Storing Temporal Triples. When a user wants to store stRDF data in Strabon, she makes them available in the form of an N-Quads document. This document is decomposed into temporal triples and each temporal triple is processed separately by the storage manager as follows. First, the temporal triple is translated into the named graph representation. To achieve this, a URI is created and it is assigned to a named graph that corrresponds to the validity period of the

[4] `http://www.openrdf.org/`

triple. To ensure that every distinct valid time of a temporal triple corresponds to exactly one named graph, the URI of the graph is constructed using the literal representation of the valid time annotation. Then, the stored triple in the named graph identified by this URI and the URI of the named graph is associated to its corresponding valid time by storing the following triple in the default graph: `(g, strdf:hasValidTime, t)` where `g` is the URI of the graph and `t` is the corresponding valid time. For example, temporal triple

```
corine:Area_4 corine:hasLandCover corine:naturalGrassland
             "[2000-01-01T00:00:00,2006-01-01T00:00:00)"^^strdf:period
```

will be translated into the following standard RDF triples:

```
corine:Area_4 corine:hasLandCover corine:naturalGrassland
corine:2000-01-01T00:00:00_2006-01-01T00:00:00 strdf:hasValidTime
                                   "[2000-01-01T00:00:00,2006-01-01T00:00:00)"^^strdf:period
```

The first triple will be stored in the named graph with URI `corine:2000-01-01T00:00:00_2006-01-01T00:00:00` and the second in the default graph. If later on another temporal triple with the same valid time is stored, its corresponding triple will end-up in the same named graph.

For the temporal literals found during data loading, we deviate from the default behaviour of Sesame by storing the instances of the `strdf:period` datatype in a table with schema *period_values(id int, value period)*. The attribute *id* is used to assign a unique identifier to each period and associate it to its RDF representation as a typed literal. It corresponds to the respective *id* value that is assigned to each URI after the dictionary encoding is performed. The attribute *value* is a temporal column of the `PERIOD` datatype defined in PostgreSQL Temporal. In addition, we construct a GiST index on the *value* column.

Querying Temporal Triples. Let us now explain how the query engine of Strabon presented in [14] has been extended to evaluate temporal triple patterns. When a temporal triple pattern is encountered, the query engine of Strabon executes the following steps. First, the stSPARQL to SPARQL 1.1 Translator converts each temporal triple pattern of the form `s p o t` into the graph pattern `GRAPH ?g  s p o . ?g strdf:hasValidTime t.` where `s`, `p`, `o` are RDF terms or variables and `t` is either a variable or an instance of the datatypes `strdf:period` or `xsd:dateTime`. Then the query gets parsed and optimized by the respective components of Strabon and passes to the evaluator which has been modified as follows: If a temporal extension function is present, the evaluator incorporates the table *period_values* to the query tree and it is declared that the arguments of the temporal function will be retrieved from the *period_values* table. In this way, all temporal extension functions are evaluated in the database level using PostgresSQL Temporal. Finally, the RDBMS evaluation module has been extended so that the execution plan produced by the logical level of Strabon is translated into suitable SQL statements. The temporal extension functions are respectively mapped into SQL statements using the functions and operators provided by PostgreSQL Temporal.

5 Evaluation

For the experimental evaluation of our system, we used two different datasets. The first dataset is the GovTrack dataset[5], which consists of RDF data about US Congress. This dataset was created by Civic Impulse, LLC[6] and contains information about US Congress members, bills and voting records. The second dataset is the CORINE Land Cover changes dataset that represents changes for the period [2000, UC), which we have already introduced in Section 2.

The GovTrack dataset contains temporal information in the form of instants and periods, but in standard RDF format using reification. So, in the preprocessing step we transformed the dataset into N-Quads format. For example the 5 triples

```
congress_people:A000069 politico:hasRole _:node17d3oolkdx1 .
_:node17d3oolkdx1 time:from _:node17d3oolkdx2 .
_:node17d3oolkdx1 time:to _:node17d3oolkdx3 .
_:node17d3oolkdx2 time:at "2001-01-03"^^xs:date .
_:node17d3oolkdx3 time:at "2006-12-08"^^xs:date .
```

were transformed into a single quad:

```
congress_people:A000069 politico:hasRole _:node17d3oolkdx1
                        "[2001-01-03T00:00:00, 2006-12-08T00:00:00]"^^strdf:period .
```

The transformed dataset has a total number of 7,900,905 triples, 42,049 of which have periods as valid time and 294,636 have instants.

The CORINE Land Cover changes dataset for the time period [2000, UC) is publicly available in the form of shapefiles and it contains the areas that have changed their land cover between the years 2000 and 2006. Using this dataset, we created a new dataset in N-Quads form which has information about geographic regions such as: unique identifiers, geometries and periods when regions have a landcover. The dataset contains 717,934 temporal triples whose valid time is represented using the `strdf:period` datatype. It also contains 1,076,901 triples without valid times. Using this dataset, we performed temporal and spatial stSPARQL queries, similar to the ones provided in Section 3 as examples.

Our experiments were conducted on an Intel Xeon E5620 with 12MB L3 caches running at 2.4 GHz. The system has 24GB of RAM, 4 disks of striped RAID (level 5) and the operating system installed is Ubuntu 12.04. We ran our queries three times on cold and warm caches, for which we ran each query once before measuring the response time. We compare our system with the following implementations.

The Prolog-based implementation of AnQL. We disabled the inferencer and we followed the data model and the query language that is used in [15], e.g., the above quad is transformed into the following AnQL statement:

```
congress_people:A000069 politico:hasRole _:node1 :[2001-01-03, 2006-12-08] .
```

[5] http://www.govtrack.us/data/rdf/

[6] http://www.civicimpulse.com/

AllegroGraph. AllegroGraph offers a set of temporal primitives and temporal functions, extending their Prolog query engine, to represent and query temporal information in RDF. AllegroGraph does not provide any high level syntax to annotate triples with their valid time, so, for example, the GovTrack triple that we presented earlier was converted into the following graph:

```
congress_people:A000069 politico:hasRole _:node1 graph:2001-01-03T... .
graph:2001-01-03T...  allegro:starttime "2001-01-03T00:00:00"^^xsd:dateTime .
graph:2001-01-03T...  allegro:endtime "2001-01-03T00:00:00"^^xsd:dateTime .
```

As AllegroGraph supports the N-Quads format, we stored each triple of the dataset in a named graph, by assigning a unique URI to each valid time. Then, we described the beginning and ending times of the period that the named graph corresponds to, using RDF statements with the specific temporal predicates that are defined in AllegroGraph[7]. We used the AllegroGraph Free server edition[8] that allows us to store up to five million statements, so we could not store the full version of the dataset.
Naive implementation. We developed a baseline implementation by extending the Sesame native store with the named graph translators we use in Strabon so that it can store stRDF graphs and query them using stSPARQL queries. We also developed in Java the temporal extension functions that are used in the benchmarks. A similar implementation has been used as a baseline in [14] where we evaluated the geospatial features of Strabon.

We evaluate the performance of the systems in terms of query response time. We compute the response time for each query posed by measuring the elapsed time from query submission till a complete iteration over the results had been completed. We also investigate the scalability with respect to database size and complexity of queries.

We have conducted four experiments that are explained below. Twenty queries were used in the evaluation. Only two queries are shown here; the rest are omitted due to space considerations. However, all datasets and the queries that we used in our experimental evaluation are publicly available[9].
Experiment 1. In this experiment we ran the same query against a number of subsets of the GovTrack dataset of various size, as we wanted to test the scalability of all systems with respect to the dataset size. To achieve this, we created five instances of the GovTrack dataset, each one with exponentially increasing number of triples and quads. The query that is evaluated against these datasets is shown in Figure 2.

Figure 3(a) shows the results of this experiment. As the dataset size increases, more periods need to be processed and as expected, the query response time grows for all systems. This is expected, as posing queries against a large dataset is challenging for memory-based implementations. Interestingly, the AnQL response time in the query Q2 is decreased, when a temporal filter is added to the temporal graph pattern of the query Q1. The use of a very selective temporal

[7] http://www.franz.com/agraph/support/documentation/current/temporal-tutorial.html
[8] http://www.franz.com/agraph/downloads/
[9] http://www.strabon.di.uoa.gr/temporal-evaluation/experiments.html

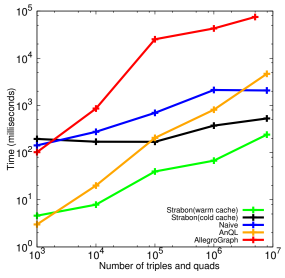
stSPARQL	**AnQL**	**AllegroGraph**
SELECT DISTINCT ?x ?name	SELECT DISTINCT ?x ?name	(select0-distinct (?x ?name)
WHERE {?x gov:hasRole ?term ?t .	WHERE {?x gov:hasRole ?term ?t .	(q ?x !gov:hasRole ?term ?t)
OPTIONAL {?x foaf:name ?name .}	OPTIONAL {?x foaf:name ?name .}	(optional (q ?x !foaf:name ?name))
FILTER(strdf:after(?t,"[...]"^^strdf:period))}	FILTER(beforeany([[...]],?t))}	(interval-after-datetimes ?t "..."))

Fig. 2. Query of Experiment 1.

filter reduces the number of the intermediate results. Also, it the implementation of AnQL performs better in workloads of up to 100,000 triples and quads, as it is a memory-based implementation. The poor performance of the baseline implementation compared to Strabon is reasonable, as Strabon evaluates the temporal extension functions in the RDBMS level using the respective functions of PostgreSQL Temporal and a GiST index on period values, while in the case of the baseline implementation a full scan over all literals is required. AllegroGraph is not charged with the cost of processing the high level syntax for querying the valid time of triples, like the other implementations, therefore it stores two triples to describe each interval of the dataset. This is one of the reasons that it gets outperformed by all other implementations. One can observe that Strabon achieves better scalability in large datasets than the other systems due to the reasons explained earlier. The results when the caches are warm are far better, as the intermediate results fit in main memory, so we have less I/O requests.

Experiment 2. We carried out this experiment to measure the scalability of all systems with respect to queries of varying complexity. The complexity of a query depends on the number and the type of the graph patterns it contains and their selectivity. We posed a set of queries against the GovTrack dataset and we increased the number of triple patterns in each query. As explained earlier, the AllegroGraph repository contains five million statements.

First, in Q2, we have a temporal triple pattern and a temporal selection on its valid time. Then, Q3 is formed by adding a temporal join to Q2. Then Q4 and Q5 are formed by adding some more graph patterns of low selectivity to Q3. Queries with low selectivity match with large graphs of the dataset and as a result the response time increases. This happens basically because in most cases the intermediate results do not fit in the main memory blocks that are available, requiring more I/O requests In the queries Q6 and Q7 we added graph patterns with high selectivity to the previous ones and the response time was decreased. This happened because of the highly selective graph patterns used. The respective response times in warm caches are far better, as expected. What is interesting in this case, is that while in cold caches the reponse time slightly increases from the query Q6 to the query Q7, in warm caches it decreases. This happens because with warm caches, the computational effort is widely reduced and the response time is more dependent of the number of the intermediate results which are produced. The query Q7 produces less intermediate results because it is more selective than Q6. AllegroGraph has the best performance in Q2, which contains only a temporal triple pattern, but when temporal functions are introduced (queries Q3-Q7), it performs worse than any other implementation. Obviously, the evaluation of a temporal join is very costly, as it internally



(a)

System	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Strabon (warm caches)	98	67	92	109	131	35	31
Strabon (cold caches)	326	437	571	647	664	412	430
Naive	3056	5860	5916	6260	6462	2594	2604
AnQL	2028	1715	4275	4379	5802	6913	7472
AllegroGraph	1016	58155	91736	121835	154561	33824	156408

(b)

System	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
Strabon (warm caches)	217	212	209	207	82	84	81	208	200
Strabon (cold caches)	485	381	377	370	250	248	248	285	376
Naive	6206	6148	6162	6196	6097	6248	6388	6332	6258

(c)

System	Q17	Q18	Q19	Q20
Strabon (warm caches)	3	13	12	208
Strabon (cold caches)	268	368	532	792
Naive	404	2868	2388	200705

(d)

Fig. 3. (a) Experiment 1: Query response time with respect to dataset size. (b), (c), (d) Experiments 2, 3, 4: Query response time in milliseconds.

maps the variables that take part in the temporal join to the respective intervals of the dataset, retrieves their beginning and ending timestamps and then evaluates the temporal operators. The AnQL implementation performs very well in queries of low selectivity but in queries of high selectivity it is outperformed by the baseline implementation. Strabon, even with cold caches, performs significantly better than the other implementations due to the efficient evaluation of the queries in the database level and the use of a temporal index.
Experiment 3. In this experiment we posed temporal queries against the GovTrack dataset in order to test the performance of different temporal operators in the FILTER clause of the query that are typically used to express a temporal join. The triple patterns in the queries posed (Q8-Q16) are identical so the queries differ only in the temporal function used in the FILTER clause of the query. For example query Q8 is the following:

```
SELECT DISTINCT ?x1 ?x2 WHERE {?x1 gov:hasRole ?term ?t1 .
                               ?x2 gov:hasRole ?term ?t2 . FILTER(strdf:during(?t1,?t2))}
```

The results of the experiment are shown in the table of Figure 3(c). For each system, the differences in performance with respect to the different temporal operators used in queries are minor, especially in the case of the naive implementation. As expected, Strabon continues to perform much better than the naive implementation as the implementation of each operator is more efficient.
Experiment 4. In this experiment we evaluate the spatiotemporal capabilities of Strabon and the baseline implementation . We used the CORINE Land Cover changes 2000-2006 dataset. This is a spatiotemporal dataset that contains more temporal triples, but there are only two distinct valid time values. Query Q17 retrieves the valid times of the temporal triples, while query Q18 is more selective

and performs a temporal join. Query Q19 is similar to Q20 but it also retrieves geospatial information so the response time is increased. Query 20 performs a temporal join and a spatial selection, so the reponse time is increased for both systems. Strabon peforms better because the temporal and the spatial operations are evaluated in the database level and the respective indices are used, while in the naive implementation these functions are implemented in Java.

6 Related Work

To the best of our knowledge, the only commercial RDF store that has good support for time is AllegroGraph[10]. AllegroGraph allows the introduction of points and intervals as resources in an RDF graph and their situation on a time line (by connecting them to dates). It also offers a rich set of predicates that can be used to query temporal RDF graphs in Prolog. As in stSPARQL, these predicates include all qualitative relations of [16] involving points and intervals. Therefore, all the temporal queries expressed using Prolog in AllegroGraph can also be expressed by stSPARQL in Strabon.

In [7] another approach is presented for extending RDF with temporal features, using a temporal element that captures more than one time dimensions. A temporal extension of SPARQL, named T-SPARQL, is also proposed which is based on TSQL2. Also, [17] presents a logic-based approach for extending RDF and OWL with valid time and the query language SPARQL for querying and reasoning with RDF, RDFS and OWL2 temporal graphs. To the best of our knowledge, no public implementation of [7] and [17] exists that we could use to compare with Strabon. Similarly, the implementations of [19] and [22] are not publicly available, so they could not be included in our comparison.

In stRDF we have not considered transaction time since the applications that motivated our work required only user-defined time and valid time of triples. The introduction of transaction time to stRDF would result in a much richer data model. We would be able to model not just the history of an application domain, but also the system's knowledge of this history. In the past the relevant rich semantic notions were studied in TSQL2 [20], Telos (which is very close to RDF) [18] and temporal deductive databases [21].

7 Conclusions

In future work, we plan to evaluate the valid time functionalities of Strabon on larger datasets, and continue the experimental comparison with AllegroGraph as soon as we obtain a license of its Enterprise edition. We will also study optimization techniques that can increase the scalability of Strabon. Finally, it would be interesting to define and implement an extension of stSPARQL that offers the ability to represent and reason with qualitative temporal relations in the same way that the Topology vocabulary extension of GeoSPARQL represents topological relations.

[10] `http://www.franz.com/agraph/allegrograph/`

References

1. Open Geospatial Consortium. OGC GeoSPARQL - A geographic query language for RDF data. OGC Candidate Implementation Standard (2012)
2. Al-Kateb, M., Ghazal, A., Crolotte, A., Bhashyam, R., Chimanchode, J., Pakala, S.P.: Temporal Query Processing in Teradata. In: ICDT (2013)
3. Allen, J.F.: Maintaining knowledge about temporal intervals. CACM 26(11) (1983)
4. Boelen, M.H., Snodgrass, R.T., Soo, M.D.: Coalescing in Temporal Databases. IEEE CS 19, 35–42 (1996)
5. Clifford, J., Dyreson, C., Isakowitz, T., Jensen, C.S., Snodgrass, R.T.: On the semantics of now in databases. ACM TODS 22(2), 171–214 (1997)
6. Date, C.J., Darwen, H., Lorentzos, N.A.: Temporal data and the relational model. Elsevier (2002)
7. Grandi, F.: T-SPARQL: a TSQL2-like temporal query language for RDF. In: International Workshop on Querying Graph Structured Data. pp. 21–30 (2010)
8. Gutierrez, C., Hurtado, C., Vaisman, R.: Temporal RDF. In: Gmez-Prez, A., Euzenat, J. (eds.) ESWC. LNCS, vol. 3532, pp. 93–107. Springer (2005)
9. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing Time into RDF. IEEE TKDE 19(2), 207–218 (2007)
10. Güting, R.H., Böhlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., Vazirgiannis, M.: A foundation for representing and querying moving objects. ACM TODS 25(1), 1–42 (2000)
11. Hurtado, C.A., Vaisman, A.A.: Reasoning with Temporal Constraints in RDF. In: Alferes, J., Bailey, J., May, W., Schwertel, U. (eds.) PPSWR. LNCS, vol. 4187, pp. 164–178. Springer (2006)
12. Koubarakis, M., Karpathiotakis, M., Kyzirakos, K., Nikolaou, C., Sioutis, M.: Data Models and Query Languages for Linked Geospatial Data. In: Eiter, T., Krennwallner, T. (eds.) RR. LNCS, vol. 7487, pp. 290–328. Springer (2012)
13. Koubarakis, M., Kyzirakos, K.: Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In: Aroyo, L., et al. (eds.) ESWC. LNCS, vol. 6088, pp. 425–439. Springer (2010)
14. Kyzirakos, K., Karpathiotakis, M., Koubarakis, M.: Strabon: A Semantic Geospatial DBMS. In: Cudr-Mauroux, P., et al. (eds.) ISWC. LNCS, vol. 7649, pp. 295–311. Springer (2012)
15. Lopes, N., Polleres, A., Straccia, U., Zimmermann, A.: AnQL: SPARQLing Up Annotated RDFS. In: Patel-Schneider, P., et al. (eds.) ISWC. LNCS, vol. 6496. Springer (2010)
16. Meiri, I.: Combining qualitative and quantitative constraints in temporal reasoning. Artificial Intelligence 87(1-2), 343–385 (1996)
17. Motik, B.: Representing and Querying Validity Time in RDF and OWL: A Logic-Based Approach. Journal of Web Semantics 12–13, 3–21 (2012)
18. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: representing knowledge about information systems. ACM TIS (1990)
19. Perry, M.: A Framework to Support Spatial, Temporal and Thematic Analytics over Semantic Web Data. Ph.D. thesis, Wright State University (2008)
20. Snodgrass, R.T. (ed.): The TSQL2 Temporal Query Language. Springer (1995)
21. Sripada, S.M.: A logical framework for temporal deductive databases. In: Bancilhon, F., DeWitt, D. (eds.) VLDB. pp. 171–182. M. Kaufmann Publ. Inc. (1988)
22. Tappolet, J., Bernstein, A.: Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In: Aroyo, L., et al. (eds.) ESWC. LNCS, vol. 5554, pp. 308–322. Springer-Verlag (2009)