SemSorGrid4Env

FP7-223913

Deliverable

D3.2

Distributed data structures and algorithms for a Semantic Sensor Grid registry

Kostis Kyzirakos, Zoi Kaoudi,
Manos Karpathiotakis and
Manolis Koubarakis

August 28th, 2009

Status: Final

Scheduled Delivery Date: August 31st, 2009

Executive Summary

The objective of WP3 is to design, implement and deploy an open, dynamic and scalable registry for the SemsorGrid4Env software architecture defined in WP1. The registry to be developed will allow the description and discovery of Semantic Sensor Grid resources: sensors, sensor networks, data sources, ontologies etc.

In Deliverable D3.1 we studied the problem of designing a data model and a query language for the registry of the SemsorGrid4Env infrastructure. We proposed the data model stRDF and the query language stSPARQL as the data model and query language for the SemsorGrid4Env registry. stRDF extends RDF(S) with the ability to represent spatial and temporal data so that sensor metadata can be represented and queried. stSPARQL extends SPARQL so that spatial and temporal data can be queried using a declarative and user-friendly language.

In this deliverable, we present a semantics and an algebra for stSPARQL. We also present Strabon, an implementation of stSPARQL that is currently been developed in WP3 and will be the basis of the SemsorGrid4Env registry implementation. Finally, we present data structures and algorithms for efficient distributed query processing in the system Atlas and a detailed performance evaluation of these on Planetlab. The implementation techniques developed in Strabon will be later on extended to Atlas to develop a distributed implementation of the SemsorGrid4Env registry.

Note on Sources and Original Contributions

The SemSorGrid4Env consortium is an inter-disciplinary team, and in order to make deliverables self-contained and comprehensible to all partners, some deliverables thus necessarily include state-of-the-art surveys and associated critical assessment. Where there is no advantage to recreating such materials from first principles, partners follow standard scientific practice and occassionaly make use of their own pre-existing intellectual property in such sections. In the interests of transparency, we here identify the main sources of such pre-existing materials in this deliverable:

- Section 2.3 contains material from Deliverable D3.1.
- Section 3.1 contains material from Deliverable D1.3v1.

Document Information

Contract Number	FP7-223913	**Acronym**	SemSorGrid4Env
Full title	SemSorGrid4Env: Semantic Sensor Grids for Rapid Application Development for Environmental Management		
Project URL	www.semsorgrid4env.eu		
Document URL	http://www.semsorgrid4env.eu/home.jsp?content=/sew/viewTerm&content=instance.jsp&_sew_var_name=instance&_sew_instance=D3.2&_sew_instance_set=SemSorGrid4Env&_origin=%2Fhome.jsp		
EU Project officer	Gaëlle Le Gars		

Deliverable	**Number**	3.2	**Name**	Distributed data structures and algorithms for a Semantic Sensor Grid registry		
Task	**Number**	3.2	**Name**	Design distributed data structures and algorithms for registries in SemsorGrid4Env		
Work package	**Number**	3				
Date of delivery	**Contractual**		31/8/2009	**Actual**	31/8/2009	
Code name	D3.2			**Status**	draft ☐	final ☑
Nature	Prototype ☐ Report ☑ Specification ☐ Tool ☐ Other ☐					
Distribution Type	Public ☑ Restricted ☐ Consortium ☐					
Authoring Partner	National and Kapodistrian University of Athens					
QA Partner	Universidad Politecnica de Madrid					
Contact Person	Kostis Kyzirakos					
	Email	kkyzir@di.uoa.gr	**Phone**	+30 210 727 5159	**Fax**	
Abstract (for dissemination)	In this deliverable, we present a semantics and an algebra for stSPARQL. We also present Strabon, an implementation of stSPARQL that is currently been developed in WP3 and will be the basis of the SemsorGrid4Env registry implementation. Finally, we present data structures and algorithms for efficient distributed query processing in the system Atlas and a detailed performance evaluation of these on Planetlab. The implementation techniques developed in Strabon will be later on extended to Atlas to develop a distributed implementation of the SemsorGrid4Env registry.					
Keywords	stRDF, stSPARQL, Strabon, Atlas, registry					

Version log/Date	**Change**	**Author**
0.2 / June 15, 2009	**Contents**	**M. Koubarakis**
0.6 / July 24, 2009	**First draft to be QAed**	**M. Koubarakis, Kostis Kyzirakos, Zoi Kaoudi, Manos Karpathiotakis**
0.7 / July 31, 2009	**QA**	**Oscar Corcho**
0.8 / August 20, 2009	**Final draft to be QAed**	**M. Koubarakis, Kostis Kyzirakos, Zoi Kaoudi, Manos Karpathiotakis**
0.9 / August 24, 2009	**QA**	**Oscar Corcho**
1 / August 28, 2009	**Final version**	**M. Koubarakis, Kostis Kyzirakos, Zoi Kaoudi, Manos Karpathiotakis**

Project Information

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number FP7-223913. The Beneficiaries in this project are:

Partner	Acronym	Contact
Universidad Politécnica de Madrid (Coordinator)	UPM	Prof. Dr. Asunción Gómez-Pérez Facultad de Informática Departamento de Inteligencia Artificial Campus de Montegancedo, sn Boadilla del Monte 28660 Spain #e asun@fi.upm.es #t +34-91 336-7439, #f +34-91 352-4819
The University of Manchester	UNIMAN	Prof. Carole Goble Department of Computer Science The University of Manchester Oxford Road Manchester, M13 9PL, United Kingdom #e carole@cs.man.ac.uk #t +44-161-275 61 95, #f +44-161-275 62 04
National and Kapodistrian University of Athens	NKUA	Prof. Manolis Koubarakis University Campus, Ilissia Athina GR-15784 Greece #@ koubarak@di.uoa.gr #t +30 210 7275213, #f +30 210 7275214
University of Southampton	SOTON	Prof. David De Roure University Road Southampton SO17 1BJ United Kingdom #@ dder@ecs.soton.ac.uk #t +44 23 80592418, #f +44 23 80595499
Deimos Space, S.L.	DMS	Mr. Agustín Izquierdo Ronda de Poniente 19, Edif. Fiteni VI, P 2, 2º Tres Cantos, Madrid – 28760 Spain #@agustin.izquierdo@deimos-space.com #t +34-91-8063450, #f +34-91-806-34-51
EMU Limited	EMU	Dr. Bruce Tomlinson Mill Court, The Sawmills, Durley number 1 Southampton, SO32 2EJ – United Kingdom #@ bruce.tomlinson@emulimited.com #t +44 1489 860050, #f +44 1489 860051

TechIdeas Asesores Tecnológicos, S.L.	TI	Mr. Jesús E. Gabaldón C/ Marie Curie 8-14 08042 Barcelona, Spain #@ jesus.gabaldon@techideas.es #t +34.93.291.77.27, #f ++34.93.291.76.00

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The objective of WP3 is to design, implement and deploy an open, dynamic and scalable registry for the SemsorGrid4Env software architecture defined in WP1. The registry to be developed will allow the description and discovery of Semantic Sensor Grid resources: sensors, sensor networks, data sources, ontologies etc.

In Deliverable D3.1 [27] we studied the problem of designing a data model and a query language for the registry of the SemsorGrid4Env infrastructure and more generally for a registry in a Semantic Sensor Grid. We surveyed related work in the areas most relevant to our work and proposed the data model stRDF and the query language stSPARQL as the data model and query language for the SemsorGrid4Env registry. stRDF extends RDF(S) with the ability to represent spatial and temporal data so that sensor metadata can be represented and queried. stSPARQL extends SPARQL so that spatial and temporal data can be queried using a declarative and user-friendly language.

The main contributions of this deliverable are the following:

- We present a semantics and an algebra for stSPARQL in the spirit of the algebra for SPARQL given in [33]. This will be the algebraic language that will drive the stSPARQL implementation. This development continues the work on stSPARQL initiated in Deliverable 3.1.

- We present the implementation of stSPARQL that is currently been developed in WP3 by extending the Sesame RDF store[1]. This implementation will serve as the basis of the SemsorGrid4Env registry implementation. As the project goes forward in time, this implementation will cover a progressively larger subset of stSPARQL as various features of it are required by the project use cases. This implementation is currently centralized and it will be later on extended to a distributed one as envisaged in the description of work for WP3 in the Technical Annex of the project. We decided to follow this "centralized version first, distributed version later" model of

[1] `http://www.openrdf.org/`, last accessed July 21, 2009.

software development so that an early centralized implementation is available to the partners while the development of a more complex distributed implementation based on our system Atlas[2] is completed. This is consistent with lessons learned in our earlier project, Ontogrid.

- We present data structures and algorithms for efficient distributed query processing in the system Atlas and a detailed performance evaluation of these on Planetlab[3]. The partner leading WP3 (NKUA) has recently joined Planetlab after dedicating two server-class machines that were acquired with SemsorGrid4Env equipment funds to the Planetlab network. This work has resulted in a more mature implementation of Atlas which will be integrated with the centralized implementation of stSPARQL to provide a distributed implementation of the SemsorGrid4Env registry. The features of this new version of Atlas include:
  - Being able to answer queries over RDF(S) databases, not just RDF databases as in our previous work [29, 20, 19]. The ability to query RDF data but also RDFS ontologies of sensors etc. is a crucial aspect of the SemsorGrid4Env project.
  - Using a dictionary encoding technique as in recent work on centralized RDF stores. Dictionary encoding is a method of data compression in which an RDF term is replaced by a positive integer which is the position of that term in a mapping dictionary. This technique allows significant space savings and faster query evaluation. In this respect, we are the first to use and evaluate this technique in a distributed implementation such as Atlas.
  - Utilizing query optimization techniques based on histograms and various heuristics [48, 36].

The organization of this deliverable is as follows. In Chapter 2 we present a semantics and an algebra for stSPARQL. In Chapter 3 we present the centralized implementation of the SemsorGrid4Env registry. In Chapter 4 we present data structures and algorithms for efficient distributed query processing in the system Atlas. Finally, in Chapter 5 we conclude this deliverable and discuss future work.

[2] `http://atlas.di.uoa.gr/`, last accessed July 21, 2009.
[3] `http://www.planet-lab.eu/`, last accessed July 21, 2009.

Chapter 2

Semantics and Algebra for stSPARQL

In this chapter we present a semantics and an algebra for the query language stSPARQL. stSPARQL is an extension of the query language SPARQL, a W3C standard for querying RDF data. stSPARQL is a query language for the stRDF data model. stRDF is an extension of the W3C standard RDF with the ability to encode temporal and spatial data. stRDF and stSPARQL were defined in Deliverable D3.1 [27].

The organization of this chapter is the following. In Section 2.1 we position our work with respect to related research and outline our contributions. In Section 2.2 we present the data model stRDF and in Section 2.3 we present the query language stSPARQL by means of examples. This section is essentially from Deliverable 3.1 and is given here only for completeness. In Section 2.4 we give a formal definition of stSPARQL and define its semantics by following an algebraic approach.

2.1 Background and Contributions

Up to now, little attention has been paid to the problem of extending RDF to represent spatial and/or temporal information such as the one needed for describing resources in a Semantic Sensor Grid registry. The only works that deal with representing temporal information in RDF are [13, 12, 16]. More recently, [23, 22, 34] proposed to represent spatial data in RDF(S) using spatial ontologies e.g., ontologies based on the GeoRSS GML specification [43]. [22] also compares various ways to use SPARQL to query such spatial data, while [34] proposes a useful extension to SPARQL, called SPARQL-ST, to query data expressed in a spatial and temporal extension of RDF. The temporal extension of RDF in [34], uses the model of [13, 12] to represent the valid time of triples.

The work presented in this chapter has the same goal with the papers cited

above: to enrich the Semantic Web with spatial and temporal data by extending RDF and SPARQL. For representing temporal data in RDF, we choose to follow the elegant approach of [13, 12]. However, for representing spatial data in RDF, we diverge significantly from the approach of [23, 22, 34]. Instead, we follow the main ideas of *spatial constraint databases* [17, 35, 38], and represent spatial geometries by *semi-linear* point sets in the n-dimensional space $\mathbb{Q}^n$ i.e., sets that can be defined by quantifier-free formulas in the first-order logic of linear equations and inequalities over $\mathbb{Q}^n$. Semi-linear sets can capture a great variety of spatial geometries, e.g., points, lines, line segments, polygons, k-dimensional unions of convex polygons possibly with holes, thus they give us a lot of expressive power [49].

The main contributions of our approach are the following:

- Following the approach of Dédale [40] and CSQL [26], we develop a constraint-based extension of RDF, called stRDF, that can be used to represent thematic and spatial data that might change over time. The main contribution of stRDF is to bring to the RDF world the benefits of constraint databases and constraint-based reasoning so that spatial and temporal data can be represented in RDF using constraints. In this way, application areas with a rich spatial and temporal component (such as the Geospatial Semantic Web [8] or the Semantic Sensor Web/Grid [42]) can be tackled using Semantic Web technologies.

  Technically, the standard RDF notion of a triple is extended in stRDF, so that the object of a triple is allowed to be a quantifier-free formula with linear constraints (i.e., a finite representation of a semi-linear subset of $\mathbb{Q}^n$). In terms of the W3C specification of RDF, this spatial extension can be realized with the introduction of a new kind of *typed literals*: quantifier-free formulas with linear constraints.

- We also present an extension of SPARQL, called stSPARQL, to query spatial and temporal data expressed in stRDF, in a declarative way. We introduce stSPARQL by example and present a detailed semantics using the algebraic approach pioneered for SPARQL in [33]. This gives us an algebra on which to base an stSPARQL implementation. Technically, stSPARQL follows the ideas in [26]; this allows us to have a useful language for expressing spatial and temporal queries while maintaining closure (i.e., staying within the realm of semi-linear point sets).

2.2 Data Model

To develop stRDF, we follow closely the ideas of constraint databases [17, 38, 25] and especially the work on CSQL [26]. First, we define the formulae that we allow as constraints. Then, we develop stRDF in two steps. The first step is to

define the model sRDF which extends RDF with the ability to represent spatial data. Then, we extend sRDF to stRDF so that thematic and spatial data with a temporal dimension can be represented.

2.2.1 Linear constraints

Constraints will be expressed in the first-order language $\mathcal{L} = \{\leq, +\} \cup \mathbb{Q}$ over the structure $\mathcal{Q} = \langle \mathbb{Q}, \leq, +, (q)_{q\in\mathbb{Q}} \rangle$ of the linearly ordered set of the rational numbers, denoted by $\mathbb{Q}$, with rational constants and addition. The atomic formulae of this language are *linear equations* and *inequalities* of the form: $\sum_{i=1}^{p} a_i x_i \Theta a_0$, where Θ is a predicate among $=$, or $\leq$, the x_i's denote variables and the a_i's are integer constants. Note that rational constants can always be avoided in linear equations and inequalities. The multiplication symbol is used as an abbreviation i.e., $a_i x_i$ stands for $x_i + \cdots + x_i$ (a_i times).

We now define semi-linear subsets of $\mathbb{Q}^k$, where k is a positive integer.

Definition 1. *Let S be a subset of $\mathbb{Q}^k$. S is called* semi-linear *if there is a quantifier-free formula $\phi(x_1, \ldots, x_k)$ of $\mathcal{L}$ where $x_1, \ldots, x_k$ are variables such that $(a_1, \ldots, a_k) \in S$ iff $\phi(a_1, \ldots, a_k)$ is true in the structure $\mathcal{Q}$.*

We will use $\varnothing$ to denote the empty subset of $\mathbb{Q}^k$ represented by any inconsistent formula of $\mathcal{L}$.

2.2.2 The sRDF data model

We now define sRDF. As in theoretical treatments of RDF [33], we assume the existence of pairwise-disjoint countably infinite sets I, B and L that contain IRIs, blank nodes and literals respectively. In sRDF, we also assume the existence of an infinite sequence of sets $C_1, C_2, \ldots$ that are pairwise-disjoint with I,B and L. The elements of each $C_k, k = 1, 2, \ldots$ are the quantifier-free formulae of the first-order language $\mathcal{L}$ with k free variables. We denote with C the infinite union $C_1 \cup C_2 \cup \cdots$.

Definition 2. *An* sRDF triple *is an element of the set*

$$(I \cup B) \times I \times (I \cup B \cup L \cup C).$$

If (s, p, o) is an sRDF triple, s will be called the subject, p the predicate and o the object of the triple. An sRDF graph *is a set of sRDF triples.*

In the above definition, the standard RDF notion of a triple is extended, so that the object of a triple can be a quantifier-free formula with linear constraints. According to Definition 1 such a quantifier-free formula with k free variables is a finite representation of a (possibly infinite) semi-linear subset of $\mathbb{Q}^k$. Semi-linear subsets of $\mathbb{Q}^k$ can capture a great variety of spatial geometries, e.g., points, lines,

line segments, polygons, k-dimensional unions of convex polygons possibly with holes, thus they give us a lot of expressive power. However, they cannot be used to represent other geometries that need higher-degree polynomials e.g., circles[1].

Example 1. *The following are sRDF triples[2]:*

$$(ex{:}lp1, rdf{:}type, ex{:}LandParcel)$$

$$(ex{:}lp1, ex{:}landUse, \text{“}forest\text{”})$$

$$(ex{:}lp1, ex{:}hasPoints, \text{“}0 \leq x \leq 5 \wedge 1 \leq y \leq 4\text{”})$$

In a GIS application the above triples define a land parcel, its use and its 2-dimensional geometry using a conjunction of linear constraints. The last triple is not a standard RDF triple since its object is an element of set C.

In terms of the W3C specification of RDF, sRDF can be realized as an extension of RDF with a new kind of *typed literals*: quantifier-free formulae with linear constraints. The datatype of these literals can be e.g., `strdf:SemiLinearPointSet` and can be defined using XML Schema. We now move on to define stRDF.

2.2.3 The stRDF Data Model

We will now extend sRDF with time. Since [44], database researchers have differentiated among user-defined time, valid time and transaction time. RDF (and therefore sRDF) supports user-defined time since triples are allowed to have as objects literals of the following XML Schema datatypes: `xsd:dateTime`, `xsd:time`, `xsd:date`, `xsd:gYearMonth`, `xsd:gYear`, `xsd:gMonthDay`, `xsd:gDay`, `xsd:gMonth`.

stRDF extends sRDF with the ability to represent the *valid time* of a triple (i.e., the time that the triple was valid in reality) using the approach of Gutierrez et al. [13, 12] where the notion of temporal RDF graphs is introduced. This approach has also been followed in the definition of the data model and language for SPARQL-ST [34].

We will consider time as a discrete, linearly ordered set of *time points*. A *time interval* $[a, b]$ is an ordered pair of time points such that $a \leq b$.

Definition 3. *An* stRDF quad *is an sRDF triple (a, b, c) with a fourth component t which is a temporal label (a natural number). For quads, we will use the notation (a, b, c, t), where t denotes the time point that the triple (a, b, c) is valid in the real world. The expression $(a, b, c, [t_1, t_2])$ is just syntactic sugar for the set of quads $\{(a, b, c, t) |\ t_1 \leq t \leq t_2\}$. An* stRDF graph *is a set of sRDF triples and stRDF quads.*

[1] There has been work on polynomial constraint databases as well, but the evaluation of queries in this case is more difficult [24] so we choose to stay with linear constraints.

[2] In this and remaining examples of this chapter, we will be using namespaces but we will not define them explicitly.

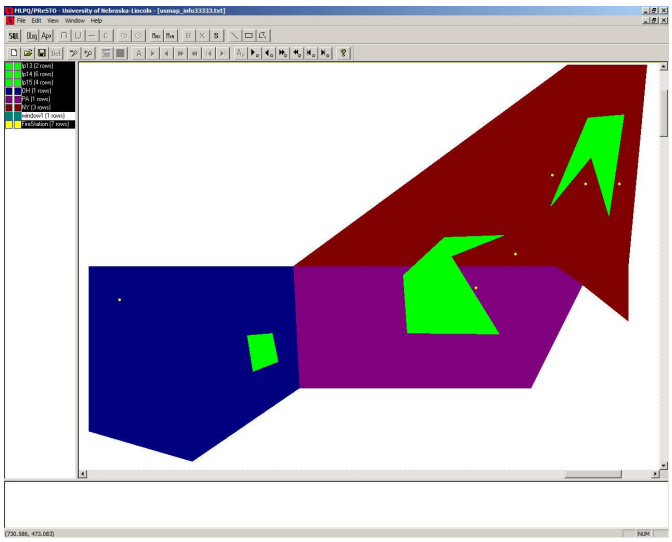


Figure 2.1: Land parcels, states and fire stations

The above definition is essentially from [13] with the difference that stRDF graphs can represent temporal and non-temporal data. Temporal data are represented by quads and non-temporal daya by triples.

We will now define a declarative query language for querying stRDF graphs.

2.3 Query Language

In this section we give the syntax of the query language stSPARQL and we will present its main features by means of examples. A more formal definition of the language is presented later in Section 2.4. The exact syntax of stSPARQL is given in [27].

We will consider a dataset from a typical GIS application that describes land parcels that represent the ground occupancy or the land use of a certain geographic region, such as cropland, pasture or forest. These land parcels may reside inside a state or may intersect several states. Fire stations, where firefighting apparatus is stored, are located near the forests. In Figure 2.1 we can see the states of Ohio, Pennsylvania and New York that are visualized with a blue, purple and maroon polygon respectively, some green polygons that represent the forests in the area and some yellow points that represent the fire stations[3].

The stRDF description of the state of New York, a forest and a fire station that have geometries that are expressed using linear constraints is the following:

[3]For visualization purposes we used the MLPQ/PReSTO System (`http://cse.unl.edu/~revesz/MLPQ/mlpq.htm`), a constraint database system developed at the University of Nebraska-Lincoln by Peter Revesz and his group.

```
ex:s1 rdf:type ex:State
ex:s1 ex:has_name "New York"
ex:s1 ex:has_geometry "(-66x+90y<-8748 and 26y<12974
  and -66x+6y>-52380 and 110y>47630) or (24y < 10392
  and -6x+15y > 1497 and 6x+9y > 8751) or (-6x+15y <1497
  and 18x < 14994 and 12x+15y > 16221)"

ex:lp13 rdf:type ex:LandParcel
ex:lp13 ex:land_use "forest" [1955-01-01T00:00:00,1988-07-18T23:59:59]
ex:lp13 ex:land_use "industrial" [1988-07-19T00:00:00,2008-07-19T00:00:00]
ex:lp13 ex:has_geometry "x+0.2y<=798 and x-2.5y<=-286 and
  -x-0.156864y<=-772 and -x+11.6y<=4078"

ex:fs1 rdf:type ex:FireStation
ex:fs1 ex:has_location "x=796 and y=437"
```

Note that land parcel `ex:lp13` used to be a forest but (sadly!) in 1988, it was declared an industrial area; a status which lasted for the next 20 years.

Example 2. Spatial selection (point query). *Which land parcel contains the point (710,400)?*

```
select ?LP, ?USE
where {?LP rdf:type ex:LandParcel . ?LP ex:has_use ?USE .
       ?LP ex:has_geometry %GEO .
       s-filter(%GEO contains "x=710 and y=400")}
```

Let us now explain the new features of stSPARQL by referring to the above example. stSPARQL has a new kind of variables: *spatial variables* that are prefixed by the character `%` (in this, we follow the syntax of SPARQL-ST [34]). Spatial variables can be used in basic graph patterns (e.g., `?LP ex:has_geometry %GEO`) to refer to spatial literals denoting semi-linear point sets. They can also be used in *spatial filters*, a new kind of filter expressions introduced by stSPARQL that is used to compare *spatial terms* using spatial predicates. Spatial filters are introduced by the keyword `s-filter`. Spatial terms include spatial constants (finite representations of semi-linear sets e.g., `"x=710 and y=400"`), spatial variables and complex spatial terms (e.g., `%GEO INTER "x=710 and y=400"`). There are several types of spatial predicates such as topological, distance, directional, etc. that one could introduce in a user-friendly spatial query language. In the current version of stSPARQL only the topological relations of [7, 9] can be used as predicates in a spatial filter expression e.g., `s-filter(%GEO1 inside %GEO2)`. The predicate `inside` returns true when the interior and boundary of `%GEO1` are contained in the interior of `%GEO2`. The spatial terms and predicates allowed will be defined in detail in Section 2.4.

Example 3. Query with intersection of areas and spatial function application. *Find all land parcels that are forests and intersect the state of New York; compute the area of this intersection.*

```
select ?LP, AREA(%GEO1 INTER %GEO2) AS ?LPAREA
where {?LP rdf:type . ex:LandParcel . ?S rdf:type ex:State .
       ?LP ex:has_use "forest" .    ?S ex:has_name "New York".
       ?LP ex:has_geometry %GEO1 .  ?S ex:has_geometry %GEO2 .
       s-filter(%GEO1 anyinteract %GEO2)}
```

In the `select` clause of an stSPARQL query we allow arbitrary spatial terms such as `AREA(%GEO1 INTER %GEO2)` above. Notice also the predicate `anyinteract` in the `s-filter` clause that returns true when two geometries are not disjoint. We also allow expressions like `AREA(%GEO1 INTER %GEO2)` that return the area (or surface) of the spatial term `%GEO1 INTER %GEO2`.

Example 4. Spatial function application (e.g., clipping) and temporal selection. *Find the URIs and parts of all land parcels that were forests on January 1st, 1988 and intersect the rectangle* $R(710, 405, 770, 425)$*; compute this intersection.*

```
select ?LP, %GEO INTER "710 <= x <= 770 and 405 <= y <= 425"
where {?LP rdf:type ex:LandParcel .
       ?LP ex:has_use "forest" #TIME . ?LP ex:has_geometry %GEO .
       s-filter(%GEO anyinteract "710<=x<=770 and 405<=y<=425") .
       t-filter(#TIME contains 2007-01-01T00:00:00)}
```

The above query demonstrates the features of stSPARQL that are used to query the valid times of triples. stSPARQL offers one more new kind of variables in addition to spatial ones: *temporal variables* that are prefixed by the character `#` (in this, we again follow the syntax of SPARQL-ST [34]). Temporal variables can be used as the last term in a new kind of basic graph pattern called *quad pattern* to refer to the valid time of a triple. Temporal variables can also appear in `t-filter` expressions, a new kind of filter that can be used in stSPARQL to constrain the valid time of triples.

The expressions that can appear in a `t-filter` are temporal constraints. A *temporal constraint* is a Boolean combination of atomic temporal constraints among temporal variables or constants. We allow any of the thirteen interval relations identified in [2] to be used as the predicate in an atomic temporal constraint e.g, `contains` in the above example. Other atomic temporal constraints allowed in stSPARQL are discussed in [27].

Example 5. Projection and spatial function application. *Find the URIs of the fire stations that are north of the state of Pennsylvania.*

```
select ?FS
where {?S rdf:type ex:State . ?FS rdf:type ex:FireStation .
       ?S ex:has_name "Pennsylvania".
       ?S ex:has_geometry %GEO . ?FS ex:has_location %FS_LOC .
       s-filter(MAX(%GEO[2]) < MIN(%FS_LOC[2]))}}
```

The above query demonstrates the projection of spatial terms and the application of metric spatial functions to spatial terms. Projections of spatial terms (e.g., `%GEO[2]`) denote the projections of the corresponding point sets on the appropriate dimensions, and are written using the notation `Variable "[" Dimension1 "," ... "," DimensionN "]"`. We also allow expressions like `MAX(%GEO[2])` that return the maximum value of the unary term `%GEO[2]`. The metric functions allowed in stSPARQL will be defined in detail in Section 2.4.

More details of stSPARQL can be found in Deliverable D3.1 [27].

2.4 Formalization and Semantics of stSPARQL

In this section, we give a formal definition of stSPARQL and define its semantics by following an algebraic approach like the one originally pioneered in [33]. We only cover the spatial features of stSPARQL in detail and their interactions with existing SPARQL concepts. The temporal features of stSPARQL can be formalized similarly using ideas from [12] and are omitted.

Let us recall from Section 2.2.2 the definitions of sets I, B, L, $C_1, C_2, \ldots$ and C. We define $ILC = I \cup L \cup C$ and $T = I \cup B \cup L \cup C \cup \mathbb{R}$. We need to include the set of real numbers $\mathbb{R}$ in the set T since as we will see below (Definition 8) the application of certain metric functions such as $AREA$ etc. can result in real numbers as answers to stSPARQL queries (see also Example 3 of Section 2.3).

We also assume the existence of the following disjoint sets of *variables*: (i) the set of non-spatial variables V_{ns}, (ii) an infinite sequence $V_s^1, V_s^2, \ldots$ of sets of variables that will be used to denote elements of the sets $C_1, C_2, \ldots$ and (iii) the set of real variables V_r. We use V_s to denote the infinite union $V_s^1 \cup V_s^2 \cup \ldots$ and V to denote the union $V_{ns} \cup V_s \cup V_r$. The set V is assumed to be disjoint from the set T.

Let us now define a concept of mapping appropriate for stSPARQL by modifying the definition of [33]. A *mapping* μ from V to T is a partial function $\mu : V \rightarrow T$ such that $\mu(x) \in I \cup B \cup L$ if $x \in V_{ns}$, $\mu(x) \in C_i$ if $x \in V_s^i$ for all $i = 1, 2, \ldots$ and $\mu(x) \in \mathbb{R}$ if $x \in V_r$.

The notions of domain and compatibility of mappings is as in [33]. The *domain* of a mapping μ, denoted by $dom(\mu)$, is the subset of V where the mapping is defined. Two mappings μ_1 and μ_2 are *compatible* if for all $x \in dom(\mu_1) \cap dom(\mu_2)$ we have $\mu_1(x) = \mu_2(x)$. For two sets of mappings Ω_1 and Ω_2, the operations of join, union, difference and left outer-join are also defined exactly as in [33]:

$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$

$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}$
$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \text{for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible }\}$
$\Omega_1 \leftthreetimes \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$

Using an algebraic syntax for stSPARQL graph patterns which extends the one introduced for SPARQL in [33], we now define the result of evaluating a graph pattern over an stRDF graph.

Definition 4. *Let G be an stRDF graph over T, p a triple pattern and P_1, P_2 graph patterns. Evaluating a graph pattern P over a graph G is denoted by $[[P]]_G$ and is defined as follows [33]:*

1. $[[p]]_G = \{\mu \mid dom(\mu) = var(p) \text{ and } \mu(p) \in G\}$, *where $var(p)$ is the set of variables occurring in p.*
2. $[[(P_1\ AND\ P_2)]]_G = [[P_1]]_G \bowtie [[P_2]]_G$
3. $[[(P_1\ OPT\ P_2)]]_G = [[P_1]]_G \leftthreetimes [[P_2]]_G$
4. $[[(P_1\ UNION\ P_2)]]_G = [[P_1]]_G \cup [[P_2]]_G$

The semantics of $FILTER$ expressions in stSPARQL are defined as in [33]. To define the semantics of $S\text{-}FILTER$ expressions formally, we first need the following definitions.

Definition 5. *A* k-ary spatial term *is an expression of the following form:*

(i) *a quantifier-free formula of $\mathcal{L}$ from the set C_k.*

(ii) *a spatial variable from the set V_s^k.*

(iii) *$t \cap t'$ (intersection), $t \cup t'$ (union), $t \setminus t'$ (difference), $BD(t)$ (boundary), $MBB(t)$ (minimum bounding box), $BF(t, a)$ (buffer) where t and t' are k-ary spatial terms and a is a rational number.*

(iv) *the* projection *$t[i_1, \ldots, i_{k'}]$ of a k-ary spatial term t where $i_1, \ldots, i_{k'}$ are positive integers less than or equal to k.*

Definition 6. *A* metric spatial term *is an expression of the form $f(t)$ where f is one of the metric functions VOL (volume), $AREA$ (area or surface), LEN (length), MAX (maximal value) or MIN (minimal value) and t is a k-ary spatial term. Area is a quantity expressing the two-dimensional size of a defined part of a surface, typically a region that is bounded by a closed polyline. The term surface refers to the total area of the exposed surface of a 3-dimensional solid for example, such as the sum of the areas of the exposed sides of a polyhedron. So, in the case of $AREA$ we require $k \geq 2$. In the case of LEN, MAX and MIN, we require $k = 1$.*

Note that Definition 6 is not recursive like Definition 5 i.e., f can only be applied once to a k-ary spatial term. The result of the application of f is a real number and the definition of mapping has already catered for this possibility.

Definition 7. *A* spatial term *is a k-ary spatial term or a metric spatial term.*

We will be interested in the value of a k-ary spatial term t for a given mapping μ such that the variables of t are all among the spatial variables of μ. This is captured by the following definition.

Definition 8. *Let t be a spatial term. Let μ be a mapping such that all the spatial variables of t are elements of $dom(\mu)$. The* value *of t for μ is denoted by $\mu(t)$ and is defined as follows:*

(i) If t is an element of C_k then $\mu(t) = t$.

(ii) If t is a spatial variable x then $\mu(t) = \mu(x)$.

(iii) If t is a projection expression of the form $t'[i_1, \ldots, i_{k'}]$ then $\mu(t)$ is a quantifier-free formula ϕ of $\mathcal{L}$ which is obtained after eliminating from $\mu(t')$ the variables corresponding to all the other dimensions except $i_1, \ldots, i_{k'}$.

(iv) If t is the intersection *$t' \cap t''$ of two k-ary spatial terms then $\mu(t) = \mu(t' \cap t'') = \mu(t') \wedge \mu(t'')$.*[4]

(v) If t is the union *$t' \cup t''$ of two k-ary spatial terms then $\mu(t) = \mu(t' \cup t'') = \mu(t') \vee \mu(t'')$.*

(vi) If t is the difference *$t' \setminus t''$ of two k-ary spatial terms then $\mu(t) = \mu(t' \setminus t'') = \mu(t') \wedge \neg\mu(t'')$.*

(vii) If t is $MBB(t')$ where t' is a k-ary spatial term, then $\mu(t)$ is a quantifier-free formula of the language $\mathcal{L}$ that represents the minimum bounding box of t'.

(viii) If t is $BD(t')$ where t' is a k-ary spatial term, then $\mu(t)$ is a quantifier-free formula of the language $\mathcal{L}$ that represents the boundary of t'.

(ix) If t is $BF(t', a)$ where t' is a k-ary spatial term and a is a rational number, then $\mu(t)$ is a quantifier-free formula of the language $\mathcal{L}$ that represents the buffer of t' with distance a. The buffer of t contains t and a zone of width a around t.

[4] In this and subsequent definitions, we assume that standardization of variables takes place before forming the conjunction etc.

(x) If t is $VOL(t')$, $AREA(t')$ or $LEN(t')$ where t' is a k-ary spatial term, then $\mu(t)$ is a real number that represents the volume, surface (or area) or length of t'.

(xi) If t is $MIN(t')$, $MAX(t')$ where t' is a unary spatial term, then $\mu(t)$ is a real number that represents the minimum or the maximum value of t'.

To guarantee closure of stSPARQL it is important to point out that the value $\mu(t)$ in the above definition is a well-defined formula of $\mathcal{L}$ in the cases *(i)-(ix)* and a real number in the cases of *(x)* and *(xi)*. This is easy to see for cases *(i)-(vi)*. For the case $t = MBB(t')$, $\mu(t)$ is $\bigwedge_{i=1}^{k}(l_i \leq x_i \wedge x_i \leq u_i)$ where l_i, u_i are the minimum and maximum values of x_i for which the formula $\mu(t')$ holds in the structure $\mathcal{Q}$. For the case $t = BD(t')$, the formula $\mu(t)$ can be constructed by performing quantifier elimination in the quantified formula defining the boundary given in Proposition 3.1 of [49]. For the case $t = BF(t', a)$ and the standard definition of buffer that uses the Euclidean distance [28], the formula $\mu(t)$ is not general an element of $\mathcal{L}$ (e.g., $BF(\text{“}x=0 \wedge y=0\text{”}, 1)$ is the unit circle with center $(0, 0)$). There are two alternative non-standard definitions of BF that allow us to stay in the realm of linear constraints. In the first case, BF can be defined using the *Manhattan distance* which measures the distance between two points along axes at right angles. For example, in the case of two dimensions, the formula $\mu(t)$ would now be the formula that remains if we eliminate variables x', y' from the formula:

$$(\phi(x', y') \wedge 0 \leq x - x' \leq a \wedge 0 \leq y - y' \leq a) \vee$$

$$(\phi(x', y') \wedge 0 \leq x - x' \leq a \wedge 0 \leq y' - y \leq a) \vee$$

$$(\phi(x', y') \wedge 0 \leq x' - x \leq a \wedge 0 \leq y - y' \leq a) \vee$$

$$(\phi(x', y') \wedge 0 \leq x' - x \leq a \wedge 0 \leq y' - y \leq a)$$

where $\phi(x', y')$ is the formula $\mu(t')$. If using Manhattan distance seems like a crude alternative to the standard definition then more detailed alternatives are likely. For example, if t defines a polygon then $BF(t, a)$ is a new polygon that contains t and the zone of width a around the polygon (however, “circular” curves are approximated by polylines). Note that the same approach is followed by vector data models e.g. the computational geometry library CGAL[5]. The cases*(x)* and *(xi)* are straightforward.

Definition 9. *An atomic* spatial condition *is an expression in any of the following forms:*

[5] `http://www.cgal.org/`

(i) $t_1\ R\ t_2$ *where* t_1 *and* t_2 *are k-ary spatial terms and* R *is one of the topological relationships DISJOINT, TOUCH, EQUALS, INSIDE, COVEREDBY, CONTAINS, COVERS, OVERLAPBDDISJOINT (overlap with disjoint boundaries) or OVERLAPBDINTER (overlap with intersecting boundaries) of [9].*

(ii) a linear equation or inequality of $\mathcal{L}$ *with metric spatial terms in the place of variables.*

Note that the form (ii) does not destroy closure of our language since these equations/inequalities allows linear equations or inequalities with terms that evaluate to real numbers and they will only be checked for satisfaction (see Definition 11), not used as constraints i.e., as elements of sets C_k*.*

Definition 10. *A* spatial condition *is a Boolean combination of atomic spatial conditions.*

Definition 11. *A mapping* μ satisfies *a spatial condition* R*, denoted by* $\mu \models R$*, if:*

1. *R is atomic and the spatial condition that results from substituting every spatial variable x of R with $\mu(x)$ holds for semi-linear sets in $\mathbb{Q}^n$.*
2. *R is $(\neg R_1)$, R_1 is a spatial condition, and it is not the case that $\mu \models R_1$.*
3. *R is $(R_1 \vee R_2)$, R_1 and R_2 are spatial conditions, and $\mu \models R_1$ or $\mu \models R_2$.*
4. *R is $(R_1 \wedge R_2)$, R_1 and R_2 are spatial conditions, and $\mu \models R_1$ and $\mu \models R_2$.*

The semantics of $S\text{-}FILTER$ expressions can now be defined as follows.

Definition 12. *Given an stRDF graph G over T, a graph pattern P and a spatial condition R, we have:*

$$[[P\ S\text{-}FILTER\ R]]_G = \{\mu \in [[P]]_G \mid \mu \models R\}.$$

Now we can define the semantics of the `SELECT` clause of an stSPARQL expression where variables (spatial or non-spatial) are selected and new spatial terms are computed. To capture the peculiarities of the `SELECT` clause of stSPARQL, we first need the following definitions.

Definition 13. *Let t be a spatial (resp. metric spatial) term and z a spatial (resp. real) variable that does not appear in t. Then, $z : t$ is called an* extended spatial term *with* target variable z*.*

Definition 14. *A* projection specification *is a set consisting of non-spatial variables, spatial variables and extended spatial terms such that all the target variables of the extended spatial terms are different from each other and different from each spatial variable.*

Definition 15. *Let μ be a mapping and W a projection specification with spatial and non-spatial variables $x_1, \ldots, x_l$ and extended spatial terms $z_1 : t_1, \ldots, z_m : t_m$. Then, $\pi_W(\mu)$ is a new mapping such that*

(i) $dom(\pi_W(\mu)) = \{x_1, \ldots, x_l, z_1, \ldots, z_m\}$.

(ii) $\pi_W(\mu)(x_i) = \mu(x_i)$ *for* $1 \leq i \leq l$ *and* $\pi_W(\mu)(z_j) = \mu(t_j)$ *for* $1 \leq j \leq m$.

The next definition closes this section by giving the semantics of an arbitrary stSPARQL query.

Definition 16. *An stSPARQL query is a pair (W, P) where W is a projection specification and P is a graph pattern. The* answer *to an stSPARQL query (W, P) over a graph G is the set of mappings $\{\pi_W(\mu) \mid \mu \in [[P]]\}$.*

2.5 Summary

In this chapter we reminded the reader of the data model stRDF and the query language stSPARQL defined in Del. 3.1. We gave a formal definition of stRDF, introduced stSPARQL by examples and presented a detailed semantics of stSPARQL using the algebraic approach pioneered for SPARQL in [33]. As a result, we now have an algebra that will be used as the basis of the stSPARQL implementation presented in the next chapter.

Chapter 3

Strabon: A Centralized Implementation of the SemsorGrid4Env Registry

In this chapter we present a centralized implementation of the SemsorGrid4Env registry currently under development in the context of WP3. The SemSorGrid4Env architecture has three major classes of services that can be characterized as an *Application Tier*, a *Middleware Tier* and a *Data Tier*[10]. The *Application Tier* comprises of services that provide domain specific functionality to consumers and applications. The *Middleware Tier* provides two main classes of services: the *Semantic Registration and Discovery* service that can be seen as a storage and querying service that can be accessed by prospective clients who want to manage thematic, spatial and temporal metadata and the *Semantic Integration* service that virtualize semantically heterogeneous data sources. The *Data Tier* comprises of services that virtualize one or more data sources in such a way as to reconcile any heterogeneity other than semantic ones.

The *Semantic Registration and Discovery Service* implements the registry of the SemsorGrid4Env project. It is built on top of our implementation of stSPARQL which we have nick-named Strabon[1].

The organization of this chapter is as follows. In Section 3.1 we present how the Semantic Registration and Discovery Service of SemsorGrid4Env relates with Strabon. In Section 3.2 we present the conceptual architecture of Strabon and in Section 3.3 we present the systems Sesame and PostGIS that we chose to use as the basis of our implementation. Finally, in Section 3.4 we present some implementation details for Strabon.

[1]Strabon (Greek: Στράβων) was a Greek historian, geographer and philosopher (`http://en.wikipedia.org/wiki/Strabo/`).

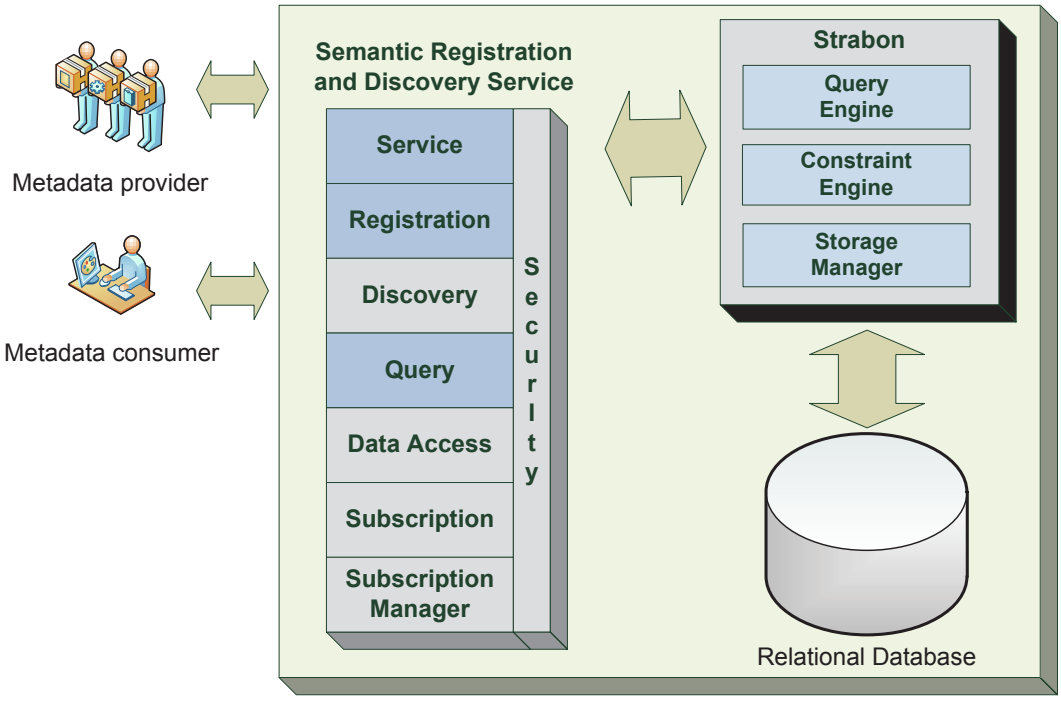


Figure 3.1: Semantic Registration and Discovery Service and Strabon

3.1 Strabon and the Semantic Registration and Discovery Service

The *Semantic Registration and Discovery Service* relates with Strabon in the way depicted in Figure 3.1. Strabon lies between clients i.e., metadata providers or metadata consumers, and the underlying DBMS that stores the metadata that are produced by the metadata providers. For example, a Web Mashup may have both roles, as it may register sensor descriptions to the registry and query it afterwards to discover sensors (published by itself or by other metadata providers) that are located inside a given region.

The interfaces exposed by the Semantic Registration and Discovery Service are presented in detail in Deliverable D1.3 [10]. For the convenience of the reader, we discuss below only the interfaces that are mandatory for a Semantic Registration and Discovery Service (colored azure in Figure 3.1).

- *Service Interface.* This interface enables clients to make well-informed and well-formed interactions with the service. For example, a client may access the Service Interface to be informed about the interfaces that are offered by the registry.

- *Registration Interface*. This interface enables a metadata provider to register an RDF(S) document that describes Semantic Sensor Web/Grid resources (e.g. sensors, sensor networks, data sources, etc.).

- *Query Interface*. This interface allows a metadata consumer to query the contents of the registry in order to discover relevant resources using a formal query language.

3.2 The Architecture of Strabon v0.1

The implementation of Strabon is being developed on top of standard technologies for data storage, indexing and optimizing of the SPARQL and SQL query languages. Strabon is built on top of Sesame and extends Sesame's components to be able to manage thematic and spatial metadata. The purpose of this section is to present the architecture of Strabon v0.1. The proposed architecture can be seen to satisfy the functional requirements of Strabon v0.1 that are tied in closely with services that need to be provided to clients as they have been discussed in Deliverable D3.1 [27]. However, there is one piece of functionality discussed in Deliverable D3.1 which is missing from Strabon v0.1. We have decided to omit support for time in this version since support for space/geometry appears more urgent for the project use cases, and time in stRDF is orthogonal to space/geometry so it can be added later[2].

Figure 3.2 presents the system architecture of Strabon. The architecture consists of the following modules:

- *Query Engine*: This module is used to evaluate stSPARQL queries posed by metadata consumers. The Query Engine receives all the stSPARQL queries from the clients, parses the queries, optimizes them, prepares an execution plan and returns the appropriate response to the client.

- *Constraint Engine*: This module is responsible for processing the part of the stSPARQL queries on spatial data represented by constraints. Although spatial objects in stRdf and stSPARQL are represented by constraints (see Chapter 2), we want our implementation to cater for the case that spatial objects provided by the users are expressed in other spatial data models e.g., in the vector based model. For this reason, the *Representation Translator* sub-module has been introduced to translate spatial objects between the equivalent constraint and vector-based representation.

[2]So the reader familiar by now with the jargon of Deliverable D3.1 and Chapter 2, might prefer to read sRDF and sSPARQL whenever we write stRDF or stSPARQL.

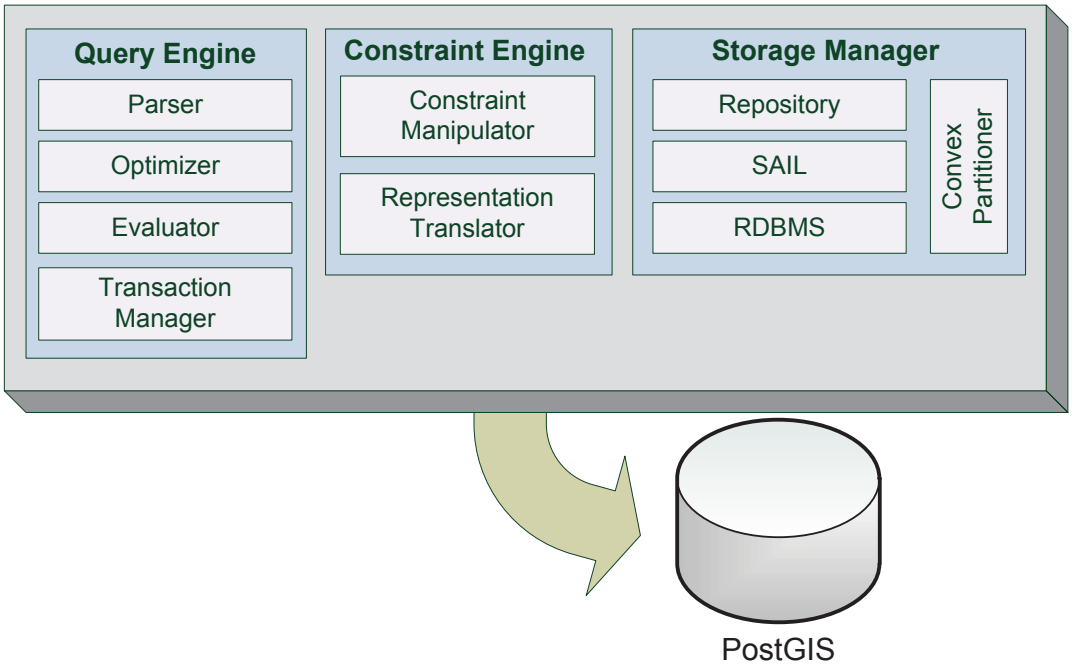


Figure 3.2: System Architecture

- *Storage Manager*: This module is responsible for storing data in the underlying RDBMS. Non-convex objects must be convexified prior to storage, since efficient computational geometry algorithms exist for various operations on convex objects. The *Convex Partitioner* sub-module may be used during data loading to decompose a non-convex object to a set of non-overlapping convex objects.

- *PostGIS*: An Object Relational Database Management System that is used for storing the stRDF data.

3.3 Implementation Choices

One of the implementation choices we had to make during the design of the system Strabon was to choose which RDF framework to extend. We examined numerous alternatives, among which were AllegroGraph RDFStore[3], Virtuoso RDF[4], OpenRDF Sesame[5] and Jena Toolkit[6]. Sesame's and Jena's open-source nature was an important factor that influenced our decision to concentrate on

[3] `http://www.franz.com/agraph/allegrograph/`, last accessed July 21, 2009.

[4] `http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSRDF`, last accessed July 21, 2009.

[5] `http://www.openrdf.org/`, last accessed July 21, 2009.

[6] `http://jena.sourceforge.net/`, last accessed July 21, 2009.

these two systems. Since we aimed to implement a system whose data model would be an extension of RDF, support for OWL, that is Jena's main advantage, was not of use to us.

On the other hand, we were greatly influenced by the fact that Sesame appeared to be more scalable than Jena. The layered architecture it is based on allows implementations of Sesame on top of a wide variety of repositories without changing any of Sesame's other components. What is more, the ability to intercept the query evaluation process in Sesame is either already supported on some layers, or it can be easily implemented. Moreover, Sesame provides full support for all the basic functions needed i.e., full compliance with SPARQL grammar, a statement that cannot be said for every alternative framework that we took under consideration. Sesame's APIs handling of inferencing tasks was another reason we leaned towards using it instead of Jena. Particularly, Sesame tends to handle inferencing tasks internally. The reason for this is that the efficiency of the inferencing and the actual storage model being used are strongly connected. Therefore, since implementations based on Sesame have complete understanding of the storage model, assumptions based on inference are more efficient in Sesame.

In this section we will present the Sesame framework that we chose to extend and the PostGIS ORDBMS that will be used as the relational back-end of the system.

3.3.1 OpenRDF Sesame

OpenRDF Sesame is a framework developed by Aduna[7] in order to facilitate the storage and querying of RDF and RDFS metadata. Sesame has a layered architecture that consists of the following layers: Access APIs and Storage and Inference Layer (SAIL) APIs. The highest ones, which are typically accessed by client programs, are the Access APIs. Access APIs are basically two: the Repository API and the Graph API. The Repository API is mostly used for querying and updating data. Since it is placed so high on the layers' hierarchy, the level of abstraction that is achieved is high as well. In other words, its goal is to conceal a substantial amount of implementation information in order to be more developer-friendly. The Graph API provides a more fine-grained service, since it represents RDF Graphs using Java Objects.

Storage and Inference Layer (SAIL) is the layer below the Access APIs. SAIL API's role is to offer storage support to the entire application, while the Repository API mentioned above just provides transparent access to SAIL. SAIL API includes interfaces, whose implementations enable numerous types of stores to be placed on the bottom layer of the Sesame Layer Stack. These stores can vary from Native RDF to RDBMS storage, such as PostgreSQL and MySQL. They are also the ones determining the functions that are available to (higher-level)

[7] `http://www.aduna-software.com/`, last accessed July 21, 2009.

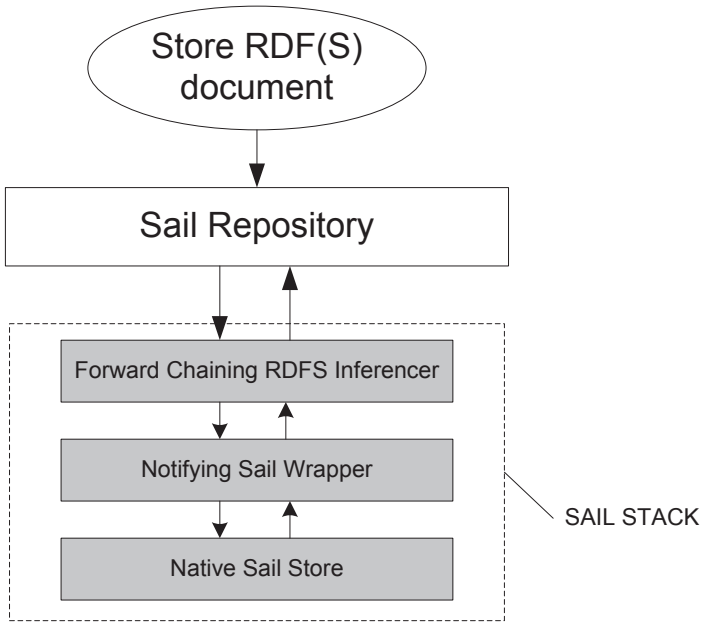


Figure 3.3: Example SAIL stack

repositories.

A very important aspect of Sesame is the ability to stack layers, be they SAIL or Repository implementations. This feature benefits Sesame users in many ways, mostly because it becomes possible for them to use a SAIL/Repository to monitor the layers below it with the use of listeners, so that a query can be intercepted, or even so that reasoning and inferencing can be performed. Thus, the system reaches high levels of scalability. In Figure 3.3 we can see SAIL stack. By stacking a SAIL on top of another, all calls to the lower SAILs pass through the SAILs that are on top of them. In this example we can see a user storing an RDF file by calling the relevant function of a *Repository*. The call is propagated to the lower stacked SAIL layers; once the new statements reach the *Forward Chaining Inferencer*, new triples may be generated and propagated to the lower layers. The *Notifying Sail Wrapper* in this example, is a simple layer that produces Notifications whenever a change occurs on the underlying data that are stored in the bottom layer.

Due to the fact that Sesame is available under an Open Source License, modifying its code in order to accommodate someone's needs is a common phenomenon. Patches contributed by its users are often published. As a result of this, it is a constantly evolving application that can easily be extended, or altered according to its user's intentions. Sesame is a rather fast RDF database, while at the same time it is widely available, since it is open-source. It is possible to achieve various levels of abstraction while using it. Specifically, one can decide the level of detail he wants to be involved with. Last but not least, it should be

mentioned that Sesame supports RDF query languages (such as SPARQL and SeRQL), and it generally provides a flexible environment for anyone attempting to get involved with Semantic Web.

3.3.2 PostGIS

Many organizations, especially the ones dealing with mapping applications, have at some point faced the problem of storing spatial data required for their operations. Numerous solutions have emerged in order to address that issue. Some of these are Spatial MySQL[8], IBM DB2 Spatial Extender[9], SQL Server 2008 spatial[10] and Oracle Spatial[11]. The RDBMS that we will describe in this section and aim to use in the context of SemsorGrid4Env is PostGIS[12].

PostGIS is a spatial language extension to the PostgreSQL[13] object-relational database, that complies with the standards set by the Open GeoSpatial Consortium (OGC). PostGIS enables PostgreSQL to support geographic objects. Therefore, PostgreSQL enhanced with PostGIS can be used as a backend storage database for Geographic Information Systems.

The objects that are supported by PostGIS are points, linestrings and polygons. In addition to the ability to store spatial data, PostGIS offers extensive querying capabilities. Spatial querying with use of PostGIS is highly effective, because of the existence of R-tree-over-GiST spatial indexes. Moreover, it can be used as a data analysis tool, since it includes a variety of spatial operators and predicates. All in all, PostGIS extends PostgreSQL with more than 300 operators, spatial functions, indexing extensions and data types.

The language usually used for the representation of the spatial objects is Well Known Text (WKT)[14]. WKT is a format specified by OGC that can convert geospatial information from a binary format to a textual display format, and vice versa.

PostGIS itself uses a great variety of libraries in its background. The ones worth mentioning are the following: The Computational Geometry Algorithms

[8] `http://dev.mysql.com/doc/refman/6.0/en/spatial-extensions.html`, last accessed July 21, 2009.

[9] `http://www-01.ibm.com/software/data/spatial/db2spatial/`, last accessed July 21, 2009.

[10] `http://www.microsoft.com/sqlserver/2008/en/us/spatial-data.aspx`, last accessed July 21, 2009.

[11] `http://www.oracle.com/technology/products/spatial/index.html`, last accessed July 21, 2009.

[12] `http://postgis.refractions.net/`, last accessed July 21, 2009.

[13] `http://www.postgresql.org/`, last accessed July 21, 2009.

[14] `http://portal.opengeospatial.org/files/?artifact_id=10378`, last accessed July 21, 2009.

Library (CGAL)[15], the Cartographic Projections Library PROJ.4[16] and the Geometry Open Source (GEOS)[17]. CGAL is a C++ library used to provide access to a range of geometric algorithms whose implementation is reliable and efficient. PROJ.4 is a cartographic projection program, while GEOS simply works as a C++ port of the Java Topology Suite[18]. In other words, it provides access to the OpenGIS Simple Features Interface Standard (SFS)[19] functions and operators. Thus, GEOS can be described as the backbone of the PostGIS extension.

3.4 Data Storage and Query Evaluation

To better understand how the Strabon components relate to one another we sketch two scenarios: 1) a user wants to store stRDF metadata in Strabon and 2) a user wants to query Strabon using stSPARQL.

3.4.1 Storing stRDF data

When a user wants to store RDF(S) metadata in Sesame, she makes them available in the form of an RDF document. The document is decomposed into RDF triples by the Rio (RDF/IO) library of Sesame, a set of parsers and writers for different RDF serialization formats[20]. Each triple is stored in the underlying repository according to the storage schema of Sesame. Currently Sesame supports two storing schemes. A "monolithic" scheme where all triples are stored to a single table, and a vertical partitioning scheme that stores triples in a per-predicate schema. In both cases, the data is stored using dictionary encoding. Dictionary encoding is the term used for a method of data compression in which each word is replaced by a number which is the position of that word in a dictionary. The use of this method enables the achievement of high information density. What is more, it facilitates the efficient further translation into various other machine languages.

Let us explain the dictionary encoding technique in more detail. It is not uncommon for URIs and literal values in RDF triples to be long strings. Therefore, their storage in RDF stores is hindered by their excessive size. A solution to this problem is not to store the entire strings in the triple tables of the RDF store. Instead, the use of shortened versions or keys is promoted. Sesame uses dictionary encoding in the following manner: It maps string URIs to positive integer identifiers. As a result, data is normalized in the following tables:

[15] `http://www.cgal.org/`, last accessed July 21, 2009.
[16] `http://trac.osgeo.org/proj/`, last acccessed July 21, 2009.
[17] `http://trac.osgeo.org/geos/`, last accessed July 21, 2009.
[18] `http://www.vividsolutions.com/jts/jtshome.htm`, last accessed July 21, 2009.
[19] `http://www.opengeospatial.org/standards/sfa`, last accessed July 21, 2009.
[20] `http://www.openrdf.org/`, last accessed July 20, 2009.

- A triples table, in case the "monolithic" storing scheme is used, or multiple per-predicate tables otherwise, using identifiers for each value.
- A mapping table, that maps the identifiers to their corresponding strings.

By using dictionary encoding, RDF triple storage overhead is kept to a minimum. Moreover, since every resource and literal is encoded using an integer value as its id field, lookups are much faster.

Example 6. *Consider the following triple that is to be stored in an RDF store:* `exstaff:85740 exterms:age "27"^^xsd:integer`. *Let* 1, 2 *and* 3 *be the integers that are assigned to the subject, predicate and object of the triple by the dictionary encoding. The subject of the triple will be stored in the RDBMS table named "uri_values", along with the integer deriving from its dictionary encoding (1). The literal's label will be stored in the RDBMS table named "label_values", along with the integer deriving from its dictionary encoding (3). In addition, the literal's datatype will be stored in the table named "datatype_values", together with the integer 3. Assuming that the per-predicate storing scheme is used, an additional tuple to RDBMS table named "age_2" will be inserted that will contain the values* 1 *and* 3 *that correspond to the subject and the object of the initial triple.*

We use the following table for storing spatial literals:

SpatialLiterals		
id	integer	The unique identification number for the spatial literal
value	varchar	The string representation of the spatial object
mbb	geometry	The minimum bounding box of the spatial object
constraint	SemiLinearPointSet	The geometry represented by constraints

Whenever an stRDF triple is to be stored, it is checked in order to find out whether its object is a spatial literal. If so, a tuple is inserted to the table mentioned above. Each tuple in this table has an *id* that is the unique identification number that corresponds to the spatial literal according to the dictionary encoding that has taken place. The string representation of the spatial geometry is stored in the *value* attribute. Additionally, the minimum bounding box of the spatial geometry is computed and stored in the *mbb* attribute of the table. This attribute is used in the query evaluation proccess, as a first step to select those objects whose mbb's satisfy a spatial predicate. Finally the constraint representation of the spatial object is stored in the *constraint* attribute. The internal form for the constraint representation satisfies the following requirements so that efficient computational algorithms may be used for various spatial operations:

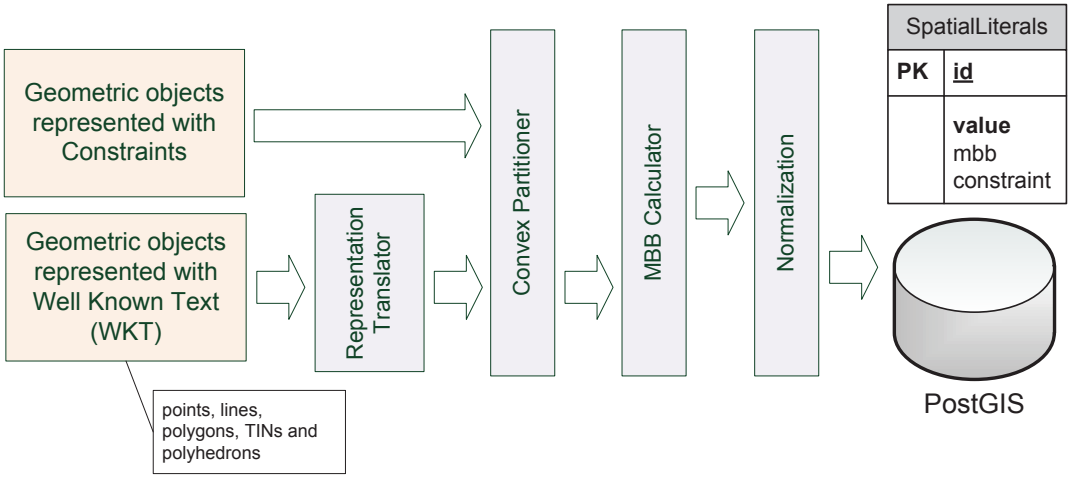


Figure 3.4: Two data loading scenarios

- Non-convex geometries are convexified prior to storage. The constraint representation of the spatial geometry is in disjunctive normal form (DNF) where each conjunct represents a convex object.

- No redundant constraints exists in each conjunct.

- The constraints are stored in a specific order, to speed up certain operations like the translation between the constraint representation and the vector-based representation.

In Figure 3.4 we present two scenarios of data loading. In the first, a geometric object may already be represented with constraints(as in stRDF), so we need to partition the geometry to convex geometries if necessary, calculate the minimum bounding box of the geometry and normalize the constraint representation prior to storage. Finally, we store the normalized constraint representation to the SpatialLiterals table according to the storing scheme mentioned above. In the second scenario, we assume a user wants to store a geometric object that is represented with Well Known Text (WKT), a format that is proposed by the Open Geospatial Consortium (OGC) and is widely accepted. If the object is non-convex, it is partitioned to convex geometries. Afterwards, the minimum bounding box of the geometry is computed and the new normalized geometry is inserted into the table SpatialLiterals according to the storing scheme described earlier.

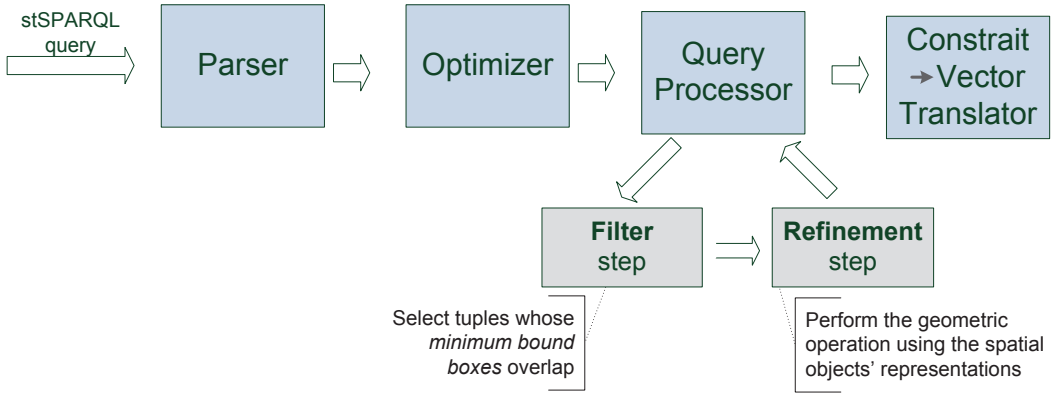


Figure 3.5: Query Evaluation

3.4.2 Evaluating stSPARQL queries

The most important module of Strabon is the query engine sub-system. The design of the query engine is classical and is illustrated in Figure 3.5. The query engine consists of a *Parser*, an *Optimizer*, a *Query Processor* and a *Constraint to Vector Translator*.

The *Parser* extends Sesame's parser to parse stSPARQL queries and check their syntax. The next step is the optimization of the query according to rewriting rules that are related to the storing scheme that is used ("monolithic" or "per-predicate"). The *Query Processor* takes as input the Query Execution Plan that is selected by the *Optimizer* and translates it to an SQL query that is executed by the underlying RDBMS. The *Query Processor* will be extended so that the evaluation of a spatial predicate is performed in two steps: a *filtering* step and a *refinement* step. The *filtering* step selects those spatial objects whose mbb's satisfy the spatial predicate. The result of the filtering step is a superset of the solution. In the *refinement* step, this superset is sequentially scanned and the spatial predicate is evaluated against the precise geometry of each spatial object. This two-step process is also used in [40] and allows the detection of spatial objects that will not satisfy the spatial predicate at a very low cost. Finally, spatial geometries represented by constraints may be translated to the equivalent vector mode representation if the user that posed the query desires so.

3.5 Summary

In this chapter we presented the centralized implementation of the SemsorGrid registry that is currently under development in WP3. We presented the conceptual and system architecture and implementation details for Strabon.

Chapter 4

Recent Extensions to Atlas

In this chapter we present some extensions to the system Atlas that we undertook recently so that we can have a more mature state-of-the-art implementation that could then be integrated with the centralized implementation of Chapter 3 to provide a distributed implementation of the SemsorGrid4Env registry. The new features in Atlas are:

- Being able to answer queries over RDF(S) databases; not just RDF databases as in our previous work [29, 20, 19]. The ability to query RDF data but also RDFS ontologies of sensors etc. is a crucial aspect of the SemsorGrid4Env project.
- Using a dictionary encoding as in recent work on centralized RDF stores. Dictionary encoding is a method of data compression in which an RDF term is replaced by a number which is the position of that word in a dictionary. In this respect, we are the first to use and evaluate this technique in a distributed implementation such as Atlas.
- Utilizing query optimization techniques based on histograms and various heuristics [48, 36].

Al the extensions of Atlas that we present in this chapter have been full evaluated experimentally on PlanetLab.

4.1 An Example

In the previous deliverable we have presented several example queries of the stSPARQL query language focusing on the syntax of the language and its spatial and temporal features. In stRDF and stSPARQL, RDFS inference is also very crucial as the spatial and temporal extensions that a query language should be able to handle. For example, consider a scenario where many sensor networks are integrated to enable the observation of various phenomena such as fire, flood,

hurricanes, earthquakes etc. An ontology describes how all different phenomena can be categorized under a hierarchy. Figure 4.1 shows a fraction of such an ontology taken from the GEONGRID[1] project.

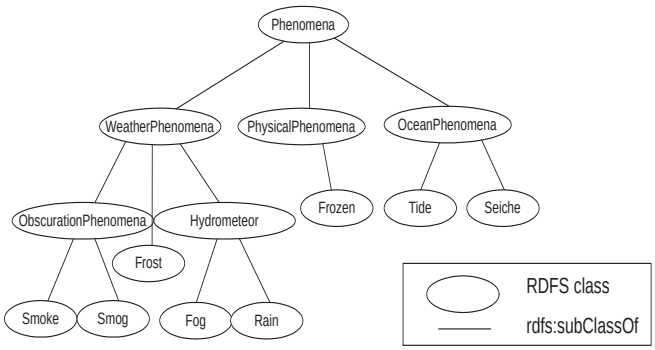


Figure 4.1: Fraction of GEON Phenomena ontology

Imagine now that we have the following dataset described in stRDF:

```
sotonmet:s1 rdf:type sit:Sensor
sotonmet:s1 sit:senses sit:parameters1
sit:parameter rdf:type geon:Smoke
sotonmet:s1 sit:location "lon=50.8839 and lat=-1.3936"

sotonmet:s2 rdf:type sit:Sensor
sotonmet:s2 sit:senses sit:parameters2
sit:parameter rdf:type geon:Fog
sotonmet:s2 sit:location "lon=50.8839 and lat=-1.3936"

chimet:s3 rdf:type sit:Sensor
chimet:s3 sit:senses sit:parameters3
sit:parameter rdf:type geon:Tide
chimet:s3 sit:location "lon=50.77556 and lat=-0.94611"
```

A simple query is: *Find all sensors that sense weather phenomena inside the rectangle R(50.744072, -1.577019 , 50.917442, -0.79875).* The query can be written in stSPARQL as follows:

```
select ?SENS, %POS
where {?SENS rdf:type sit:Sensor .
       ?SENS sit:senses ?par .
```

[1] http://www.geongrid.org/

```
?par rdf:type geon:WeatherPhenomena
?SENS sit:location %POS .
s-filter(%POS inside
     "50.744072 <= lon <= 50.917442 and
     -1.577019 <= lan <= -0.79875")}
```

The evaluation of this query over the above dataset would not find any sensors that sense weather phenomena. Therefore, RDFS inference should take place to infer that smoke and fog are weather phenomena based on the RDFS schema shown in Figure 4.1.

In the following sections, we show how we have implemented and evaluated RDFS reasoning in Atlas together with some optimization techniques. Our contributions can be summarized as follows:

- We present a query processing algorithm that deals with SPARQL basic graph pattern queries over RDF(S) databases (Section 4.3.1). The algorithm extends the algorithm QC of [29] by utilizing the backward chaining approach of [18] to enable RDFS inference.
- As triples might contain long strings, we implement a distributed *mapping dictionary* and replace entire URIs and literals by integer identifiers in our storage scheme (Section 4.3.2). Although this is by now standard in centralized RDF stores [30, 6, 50], our paper is the first that discusses how to implement and evaluate this scheme in a DHT environment.
- We present query optimization techniques that improve query response time and network bandwidth consumption (Section 4.4). Our techniques rely on a simple variation of the standard bound-is-easier heuristic of relational and datalog query processing [48] and estimations of the selectivity of triple patterns using VOptimal histograms [36].
- We implement the algorithms and optimization techniques presented as extensions to our prototype Atlas (`http://atlas.di.uoa.gr/`) and evaluate them in PlanetLab (`http://www.planet-lab.org/`) using the Lehigh University Benchmark (LUBM) [11] (Section 4.5).

4.2 System and Data Model

4.2.1 System model

We assume a structured overlay network where peers are organized according to a DHT protocol. DHTs are structured P2P systems which try to solve the *lookup problem*; given a data item x, find the peer which holds x. Each peer and each data item is assigned a unique m-bit identifier by using a hash function (e.g.,

Table 4.1: Adorned RDFS Entailment Rules

	Head	**Body**
1a	$type^{kf}(?x, ?y)$	$triple^{kbf}(?x, rdf{:}type, ?y)$
2a(rdfs2)	$type^{kf}(?x, ?y)$	$triple^{kff}(?x, ?p, ?z)$, $triple^{fbf}(?p, rdfs{:}domain, ?y)$
3a(rdfs3)	$type^{kf}(?x, ?y)$	$triple^{ffk}(?z, ?p, ?x)$, $triple^{fbf}(?p, rdfs{:}range, ?y)$
4a(rdfs9)	$type^{kf}(?x, ?y)$	$triple^{kbf}(?x, rdf{:}type, ?z)$, $subClass^{ff}(?z, ?y)$
5a	$subProperty^{kf}(?x, ?y)$	$triple^{kbf}(?x, rdfs{:}subPropertyOf, ?y)$
6a(rdfs5)	$subProperty^{kf}(?x, ?y)$	$triple^{kbf}(?x, rdfs{:}subPropertyOf, ?z)$, $subProperty^{ff}(?z, ?y)$
7a	$subClass^{kf}(?x, ?y)$	$triple^{kbf}(?x, rdfs{:}subClassOf, ?y)$
8a(rdfs11)	$subClass^{kf}(?x, ?y)$	$triple^{kbf}(?x, rdfs{:}subClassOf, ?z)$, $subClass^{ff}(?z, ?y)$
9a	$infTriple^{kff}(?x, ?p, ?y)$	$triple^{kff}(?x, ?p, ?y)$
10a(rdfs7)	$infTriple^{kff}(?x, ?p, ?y)$	$triple^{kff}(?x, ?p1, ?y)$, $subProperty^{ff}(?p1, ?p)$

SHA-1). The identifier of a peer can be computed by hashing its IP address. For data items, we first have to determine a *key* k and then hash this key to obtain the identifier id_k. The lookup problem is then solved by providing a simple interface of two methods; PUT(id_k, x) and GET(id_k). In Bamboo [39], when a peer receives a PUT request, it efficiently routes the request to a peer with an identifier that is numerically closest to id_k using a technique called prefix routing. This peer is responsible for storing the data item x. We will call this peer *responsible peer for key* k. In the same way, when a peer receives a GET request, it routes it to the responsible peer for k to fetch data item x. Such requests can be done in $O(logN)$ hops, where N is the number of network peers.

4.2.2 Data model

We define the data model that we will use in the rest of the paper. We assume that the reader is familiar with the notions of RDF *triple* and *triple pattern*. RDF data as well as RDFS schemas (we will further use the term RDF(S) to refer to both) can be encoded by sets of triples. An *RDF(S) database* is a set of triples. To support RDFS reasoning, we will use the RDFS entailment rules of RDF Semantics [15] expressed in datalog.

Let I, L, V denote the non-empty and pairwise disjoint sets of IRIs, literals and variables, respectively. Let $Pred$ and $Const$ denote the sets of datalog predicates and constants respectively that are allowed in our rules. $Pred$ is equal to the set {`subClass, subProperty, type, infTriple, triple`} and $Const$ is equal to the set $I \cup L$. Predicate `triple` is the only extensional database relation (edb) and predicates `subClass, subProperty, type, infTriple` are the intensional database relations (idb). A datalog *atom* of arity n is an expression $p(t_1, .., t_n)$ with $p \in Pred$ and $t_1, .., t_n \in (Const \cup V)$. To avoid confusion with the double meaning of the word *predicate*, from now on we will refer to the predicate of a triple with the word *property* and to the term of a datalog rule with the word *predicate*.

We extend the concept of rule adornment from recursive query processing

[48] in order to exploit the distributed philosophy of DHTs. An *adornment* of a predicate p with n arguments at a peer i is an ordered string a of k's, b's and f's of length n, where k indicates an argument of p which is the *key* that peer i is responsible for, b indicates a bound argument which is not the *key*, and f a free argument.

One adorned version of each RDFS entailment rule is shown in Table 4.1 (other used adornments are omitted due to space limitations). A careful inspection of Table 4.1 reveals the semantics of the given datalog rules which can be informally expressed as follows. The predicates `type, subProperty, subClass` encode all the facts that can be retrieved from an RDF(S) database DB regarding instantiation, property hierarchies and class hierarchies respectively or can be inferred from DB using the RDFS entailment rules of [15] appearing in Table 4.1. The predicate `infTriple` represents all the other knowledge present in an RDF(S) database: triples explicitly created that do not capture instantiation, subproperty or subclass relationships plus inferred triples using the rule $rdfs7$ of [15].

A *basic graph pattern* (BGP) of a SPARQL query is a set of triple patterns [37]. For the rest of the paper, we deal only with BGP queries. We assume that the reader is familiar with the semantics of BGP queries and only present some definitions that will be used throughout the paper.

A pair of triple patterns or datalog atoms will be called *joined* if they share at least one variable. We will use the equivalence of triple patterns to datalog atoms as defined above to navigate freely among these two representations.

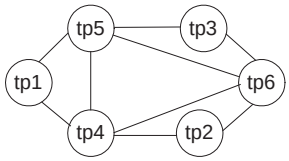
Let DB be an RDF(S) database. The *answer* to a BGP query is defined as the answer to the same query posed over the logic program formed by the union of the triples in DB and the RDFS entailment rules of Table 4.1.

We define a query plan adopting the graph-based approach also used in [46, 30] (such techniques are also standard in relational DBMS starting with System R). A BGP is represented as a set G of undirected connected graphs that are pairwise disconnected. The execution order of the query graphs in G does not affect the query performance since the final answer corresponds to the Cartesian product of the partial result sets of each element of G. For that reason, in the rest of the paper, we focus on single connected graphs that are assumed to be members of G and are defined as follows. A *query graph* g is a pair (N, E), where N (the nodes of g) is a set of triple patterns and E (the edges of g) is a set of pairs of joined triple patterns. The size N_g of a query graph g is the number of nodes of g, i.e., the number of triple patterns. An *execution plan* p_g for g is a total order of the nodes of g. The size of the execution plan space is $N_g!$.

In Fig. 4.2(a), we present an example query that will be used throughout the paper. The query is SPARQL LUBM Q9 [11] and asks for the students who take courses taught by their advisor. The query graph of Q9 is shown in Fig. 4.2(b).

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x ?y ?z
WHERE {
?x rdf:type ub:Student .   (tp1)
?y rdf:type ub:Faculty .   (tp2)
?z rdf:type ub:Course .    (tp3)
?x ub:advisor  ?y.         (tp4)
?x ub:takesCourse  ?z .    (tp5)
?y ub:teacherOf  ?z)       (tp6) }
```

(a) SPARQL query



(b) Query graph

Figure 4.2: LUBM Query 9

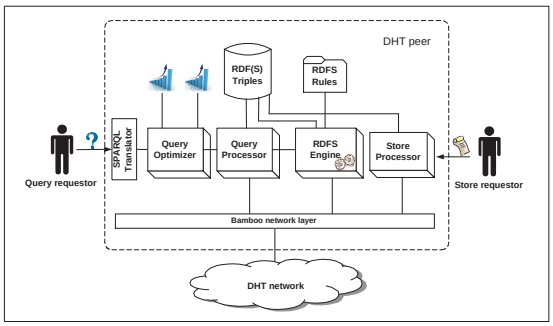


Figure 4.3: System architecture

4.2.3 System architecture

A higher level view of each peer's architecture as implemented in Atlas is shown in Fig. 4.3. Any peer can accept either a request for storing RDF(S) data in the network (STORE request) or a request for evaluating a SPARQL query (QUERY request). Upon receiving a STORE request, the store processor is responsible for initiating the protocol that distributes the given triples in the network using Bamboo's PUT interface. Every triple that arrives at the responsible peer, it is stored in the local RDF(S) database of this peer. In case of a QUERY request, the SPARQL translator receives the SPARQL query and translates it to the equivalent query graph. Then three modules are responsible for the distributed evaluation of the query: the query optimizer, the query processor and the RDFS engine. Sections 4.3 and 4.4 explain in detail how this is accomplished.

4.3 Query Evaluation Algorithms

In [18], we presented both forward and backward chaining algorithms for RDFS reasoning on top of DHTs. In this paper, we extend this work and, based on our previous work on RDF query processing on top of DHTs [21, 29], we present an algorithm for evaluating SPARQL BGP queries over *RDF(S)* databases on top of DHTs. The algorithm implements the needed RDFS reasoning functionality by incorporating and adapting the *backward chaining* approach of [18] using the adorned datalog rules. We could have also used *forward chaining* where all

inferred triples are precomputed and stored in the network a priori and then exploit the RDF query processing algorithms of [21, 29]. However, after having experimented with both RDFS reasoning algorithms in [18], we have shown that a straightforward forward chaining algorithm in a distributed setting, such as a DHT, has two important disadvantages: excessive space requirements and inability to reach a fixpoint at a reasonable amount of time. Therefore, we choose to use the backward chaining algorithm of [18].

Storage scheme. Firstly, let us briefly explain how RDF(S) triples are stored in the network after a STORE request. As in [18], we have adopted the triple indexing algorithm originally presented in [5] where each triple is indexed in the DHT *three times*, once for its subject, once for its property and once for its object. Whenever a node receives a request to store a triple, it sends three DHT PUT requests using as key the subject, property and object respectively, and the triple itself as the item. The key is hashed to create the identifier that leads to the peer where the triple is stored. We exploit the fact that many triples share a key (i.e., they have the same subject, property or object) and should be stored at the same peer. So, instead of sending different PUT messages for each triple, we group triples in lists based on their distinct keys, hash these keys and send the triples to the peer that will store them using a single message [18]. This storage scheme actually creates three indexes on a giant triples table which is distributed in the network. The benefit of this scheme for the work we present in this paper is that it enables complete and efficient RDFS reasoning using the backward chaining algorithm of [18]. In our earlier work [29], we have also experimented with indexing all possible combinations of subject, property and object. Independently, this method has also been utilized in most recent centralized RDF stores [30, 50, 14] where several indexes are maintained for faster retrieval. The use of these more exhaustive indexing schemes in the problem studied in this paper is the subject of future work.

4.3.1 Evaluating BGP queries using backward chaining

Let us, firstly, introduce the notation in our algorithms description. Keyword **event** precedes every event handler for handling messages, while keyword **procedure** declares a procedure. In both cases, the prefix is the peer identifier in which the handler or the procedure is executed. Local procedure calls are not prefixed. Keyword **sendto** declares the message that we want to send to a peer with known its identifier. Keyword **receive** is used similarly.

Query evaluation starts when peer x receives a QUERY request. The SPARQL translator of peer x translates the query to a set of query graphs G and delivers G to the optimizer. For each graph $g \in G$, the optimizer is responsible for generating an efficient execution plan p_g (details of how this is done will be explained in Section 4.4). Then, each execution plan is delivered to the query processor which creates a `queryReq` message and sends it in the network. If

more than one query graphs where created, peer x sends the relevant messages in parallel, gathers all the result sets and produces the final answer by computing the Cartesian product of these result sets. Notice that the Cartesian product is computed at peer x and is not transmitted in the network.

The main idea of Algorithm 1 is that each triple pattern in a query graph is evaluated at a (potentially) different peer possibly by performing RDFS reasoning using the rules of Table 4.1. Then, the results are joined with the intermediate results from previous triple patterns. To evaluate a triple pattern, a *key* from the triple pattern is chosen and then hashed to create the identifier that will lead to the responsible peer. The key is one of the constant parts of the triple pattern. When there are multiple constant parts, we select keys in the order "subject, object, property" based on the fact that we prefer keys with lower selectivity and the reasonable assumption that subjects or objects have more distinct values than properties. In that way, we achieve a better load distribution among the peers [29]. We will call the peer that is able to evaluate a triple pattern *responsible peer for the triple pattern*. The bindings of the triple pattern's variables are retrieved from the matching triples stored locally at the peer.

The left part of Algorithm 1 shows the pseudocode for the case when a `queryReq` message arrives at a peer p. The message contains the triple pattern tp which should be evaluated at peer p, the execution plan p_g which has not been evaluated yet, the intermediate results $partialRes$ so far (initially empty), the answer variables $vars$ and the IP address $retIP$ of the peer that posed the query. Peer p firstly transforms the triple pattern to the equivalent adorned datalog atom and calls local procedure BC-RDFS. This procedure outputs a new relation $localR$ which contains all the bindings of the triple pattern's variables including the inferences. If $partialRes$ is empty, peer p is the first peer to participate in the the query evaluation of the query and assigns $localR$ to $partialRes$. Otherwise, it computes the natural join of $localR$ and $partialRes$ and assigns it to $partialRes$. This procedure terminates in two possible ways. Either, the execution plan p_g becomes empty or the relation $partialRes$ becomes empty. In the latter case, peer p returns an empty set to the peer that issued the query. In the former case, the peer computes the projection of relation $partialRes$ on the answer variables $vars$ of the query and returns the result set to the peer that issued the query. If nothing of the above occurs, the query evaluation continues with the next triple pattern tp' and a new `queryReq` message is sent to the next peer responsible for tp'.

The right part of Algorithm 1 shows the pseudocode for the procedure BC-RDFS which illustrates how the backward chaining algorithm works. When BC-RDFS is called, the input predicate p^a is checked against the head of the adorned rules of Table 4.1. Rules that can be applied to the predicate are added to the list $adornedRules$. Each rule can have one or two predicates in its body. Rules that have one predicate in their body (e.g., rule 1a) actually imply information that comes from the local triples that are stored in the peer's database and can

Algorithm 1: BGP Query evaluation algorithm with backward chaining

```
1  event n.queryReq (id, key, tp, p_g,
   partialRes, vars, retIP) from m
2      Let p^a be the adorned atom of tp;
3      localR=BC-RDFS (p^a);
4      if partialRes = ∅ then
5          partialRes = localR;
6      else
7          partialRes = localR ⋈ partialRes;
8      end
9      if partialRes = ∅ then
10         sendto retIP.queryResp(∅);
11         return;
12     end
13     if p_g = ∅ then
14         answer = π_vars(partialRes);
15         sendto retIP.queryResp(answer);
16         return;
17     end
18     tp' = GetNextTriplePattern(p_g);
19     p_g ←Remove(tp', p_g);
20     key' = FindKey(tp');
21     id' = Hash(key');
22     sendto id'.queryReq(id', key', tp', p_g,
       partialRes, vars, retIP)
23 end event
```

```
1  procedure p.BC-RDFS (p^a)
2      adornedRules =ApplyRule(p^a);
3      forall rules in adornedRules do
4          r ← RemoveFirst(adornedRules);
5          if r has one predicate then
           R=MatchPredicate (p^a);
6          else
7              Let p_k be the adorned predicate of r
               with a k element in its adornment and
               p_f the free predicate;
8              R' = MatchPredicate (p_k);
9              if R' = ∅ then return R;
10             for each value v_i of the shared variable
               Z in R' do
11                 id_i =Hash (v_i);
12                 rewrite p_f to p'_f;
13                 sendto id_i.BC-RDFSReq(p'_f)
14                 receive BC-RDFSResp(R_i) from
                   id_i
15                 R = R ∪ R_i;
16             end
17         end
18     end
19     return R;
20 end procedure

1  event n.BC-RDFSReq (p^a) from m
2      R=BC-RDFS (p^a);
3      sendto m.BC-RDFSResp(R)
4  end event
```

always be evaluated locally since this predicate is an edb relation. For rules with two predicates in their body, we have to decide which predicate should be evaluated first. We select to first evaluate the predicate that can be processed locally to avoid extra communication costs. This can be done easily by selecting the appropriate adorned rule. Then, by using sideways information passing, more messages are sent to collect all inferences from other peers. For more details on how the algorithm works the reader might refer to [18].

4.3.2 Mapping dictionary

As URIs and literals may consist of long strings, we have implemented a mapping dictionary similar to centralized RDF stores [30, 6]. URIs and literals are mapped to integer identifiers and then, triple storage and query evaluation is performed using these identifiers. Since the DHT inherently provides the functionality of hash indexing, we have chosen to utilize this functionality to implement a distributed, scalable and fault-tolerant mapping dictionary.

When peer p receives a STORE request for a set of RDF(S) triples, it needs to invent a unique integer identifier id for each triple component c or retrieve the identifier of c if such an identifier has been assigned already (by p itself or any

other peer). Thus, p first examines its local database to see whether it already knows an identifier for c. If not, it sends a GET request with parameter the hash value of c to discover the id from the network. Then, for each triple component c that does not have an identifier already, peer p sends two PUT requests, one with key c and value id and one with key id and value c so that the mapping between c and p is stored in the DHT and become available to other peers. Finally, p replaces each triple component with its assigned identifier and stores the given set of triples in the network.

When a query arrives at peer p, all constants appearing in the query need to be translated to their identifiers. For each constant c, p runs a procedure similar to the one we sketched above to retrieve the dictionary identifier of c. After gathering all results, p replaces constants in the query by their identifiers and the query processing begins. In case any of the constants has no assigned identifier, the answer to the query is empty so no query processing needs to take place. After the query processing has finished and peer p has received the result set of the query, p needs to replace the identifiers included in the result set with their corresponding values. For each id found in the result set, p sends a GET request with parameter the hash value of id and replaces the identifiers with their initial values.

The uniqueness of identifiers in the above protocol could be ensured in various ways. We have chosen to use the following scheme which is fully distributed (thus scalable and fault tolerant) and does not require any kind of coordination between the peers that need to create dictionary identifiers. In our scheme, each peer keeps a local integer counter consisting of l bits which is initial set to 0. l is incremented by 1 everytime we want to generate a new identifier. Remember that each peer that joins the network is assigned a unique m-bit identifier by hashing its IP address. We create an n-bit identifier id for a triple component c by concatenating the m bits of the peer's unique identifier with the l bits of the current local counter. Depending on the network setting and the application requirements, we can determine an appropriate value for l so that each n-bit identifier is of reasonable space (e.g., 32 bits in our implementation).

4.4 Query Optimization

The goal of query optimization is to find an execution plan p_g that optimizes the performance of a database system with respect to a metric of interest. The main metrics of interest to us in this paper are the time required to answer a query (query response time), and the network bandwidth that is consumed during query evaluation. The optimization techniques we use are selectivity-based heuristics since the selectivity of the triple patterns can affect both of these metrics. Using standard terminology from relational systems, the *selectivity of a triple pattern* tp, denoted by $sel(tp)$, is the fraction of total number of triples in the network

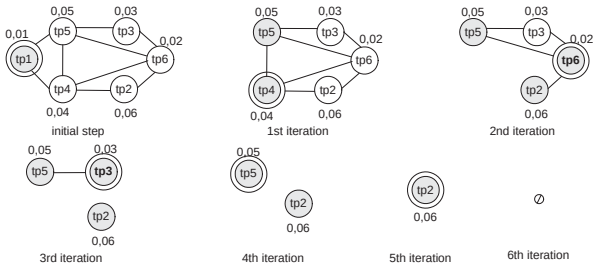


Figure 4.4: Query optimization example of LUBM Query 9

that match *tp*. We first describe the optimization algorithm which orders the triple patterns based on their selectivity and then present two selectivity-based heuristics.

The optimization algorithm works as follows. Using the query graph representation, each node is assigned with the selectivity of the corresponding triple pattern. The algorithm firstly selects the query graph node n_0 with the minimum selectivity and adds it to the final execution plan. Then, it marks the nodes that are connected with n_0 and removes n_0 from the graph. The algorithm iteratively chooses the node n_{min} with the minimum selectivity, selecting only from the marked ones, adds it to the final execution plan, marks the nodes connected with n_{min} and removes n_{min} from the graph. The algorithm terminates when no nodes are left in the graph. Figure 4.4 depicts how the optimization algorithm would perform for the example query of Fig. 4.2 given some randomly chosen selectivities. At each iteration the selected triple pattern (i.e., the node with the minimum selectivity) is shown with a double circle. The algorithm ensures that a Cartesian product computation will never occur. This is crucial for the performance of a distributed system where intermediate results are transmitted through the network.

The above algorithm is a variation of the minimum selectivity algorithm [45]. This algorithm has also been used recently for the optimization of SPARQL BGP queries in a centralized environment by [46]. The difference of the above algorithm with the algorithms of [45] and [46] is that we use selectivities of single triple patterns and not selectivities of joined triple patterns (the extension to joins is the subject of future work).

A variation of the bound-is-easier heuristic. We consider a simple variation of the standard bound-is-easier heuristic of relational and datalog query processing [48] and assume that the more components of a triple pattern are bound the more selective it is. We further enrich this heuristic by considering the position of the bound components of a triple pattern if two triple patterns have the same number of bound components. We assume that subjects are more selective than objects which in turn are more selective than properties and make the following assumption regarding the selectivity ordering: $sel(\texttt{(s, p, o)}) <$

$sel(\texttt{(s, ?x, o)}) < sel(\texttt{(s, p, ?x)}) < sel(\texttt{(?x, p', o)}) < sel$ (`?x, rdf:type, o`) $< sel(\texttt{(s, ?x, ?y)}) < sel(\texttt{(?x, ?y, o)}) < sel(\texttt{(?x, p, ?y)})$, where $\texttt{p}' \neq$ `rdf:type`. Notice that some of these selectivity relationships are always true independently of the data while other are simply heuristics. For example, if we have two triple patterns of the form `(?x, p, o)` and the property `p` is equal to `rdf:type` to the second one, we heuristically assume the first one more selective. Finally, we add to this heuristic the fact that a triple pattern with a variable that can be projected out is preferable from a triple pattern of the same form with an answer variable. Based on this heuristic, we can assign weights to the nodes of the query graph accordingly and then run the optimization algorithm described above.

Histograms. Using the attribute value independence assumption [41], the selectivity of a triple pattern tp is computed using the formula: $sel(tp) = sel(s) * sel(p) * sel(o)$. In order to compute the selectivity of each triple pattern, the frequency distribution of each triple component is required. Since a structure with the exact frequency distribution of each such component would require excessive memory space, a commonly used method is *estimating* the frequency distribution by creating statistics using histograms. We compute the selectivity of the bound components of tp using histograms and we assume a selectivity of value 1.0 for the variables. We also assume selectivity 1.0 for the property `rdf:type`. As future work, we consider the implementation of multidimensional histograms.

The scheme we used for building histograms in the distributed setting is the following. Each peer is responsible for creating and maintaining statistics of its locally stored RDF(S) data and more precisely of the triple components which is responsible for. In order to create the statistics, peer p retrieves all triples from its local database. Triples of the form `(s,rdf:type,o)` are treated differently. Therefore, p creates four histograms: one for estimating the frequency of subjects, one for estimating the frequency of properties, and two for estimating the frequency of objects. The first one is used for keeping the frequency estimation of the objects whose property is `rdf:type` (i.e., objects are RDFS classes) and the second one for keeping the frequency estimation of the objects with different properties. We found experimentally that this differentiation between objects gives a more accurate estimation for the objects' frequency. The histograms we use are VOptimal histograms [36]. In addition, in order to estimate the selectivity of a component, the total number of triples indexed in the network is required. We have adopted a broadcast protocol for this reason. In [31], several solutions for distributed counting in peer-to-peer environments are outlined and a specific distributed structure is proposed. Studying such issues is out of the scope of the paper.

Note that the components used for creating the histograms are only the components for which peer p is responsible for. The storage scheme we use guarantees that every triple containing a component c will be located at the local database of the peer responsible for c and hence the number of occurrences of c in the

whole RDF(S) database can be found locally.

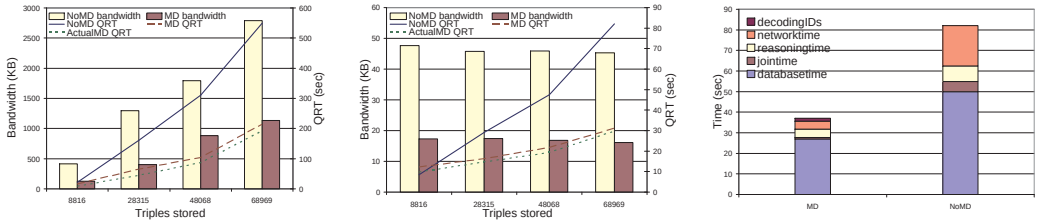
When peer p receives a QUERY request, it needs to retrieve the selectivity for each bound component c appearing in the triple patterns of the query. This is achieved by sending a message to the peer responsible for c specifying also the type t of the component (i.e., subject, property, object or class). The peer that receives this message retrieves the average frequency of c from the corresponding histogram (depending on the t value) and sends back the selectivity. Then, p computes the selectivities of all triple patterns, assigns them to the nodes of the query graph and runs the optimization algorithm.

Further considerations. The number of distinct property values found in an RDF(S) database is small in general. In addition, since our data are distributed in the network, the number of distinct properties which a peer is responsible for becomes even smaller. Therefore, we allow for maintaining the exact frequency distribution of properties. However, if the structure exceeds a fixed amount of memory, the system adaptively chooses to construct a histogram as described above. The same holds for the frequency of the class names. Finally, the histograms built do not take into account inferred data. Therefore, every time a query is evaluated and new inferences are discovered, the corresponding histograms are updated. Notice that the update of the histogram does not affect the performance of the query evaluation since it is performed at a peer that has already participated in the query evaluation and has already forwarded the query to a different peer.

4.5 Experimental Evaluation

In this section, we present an experimental evaluation of the algorithms presented in the paper, which have been implemented as an extension to our prototype system Atlas (`http://atlas.di.uoa.gr/`). We used as a testbed the PlanetLab network (`http://www.planet-lab.org/`) with 281 nodes across four continents that were available and lightly loaded at the time of the experiments. For our evaluation we used the LUBM benchmark [11] that provides datasets of arbitrary sizes and 14 queries. In the following, we only present results for queries with more than 4 triple patterns so that the benefits of the proposed optimizations can be clearly demonstrated. All measurements are averaged over 10 runs.

Query processing with a mapping dictionary. We first evaluate the query processing algorithm of Section 4.3.1 and demonstrate the importance of having a mapping dictionary. Measurements about the storage usage are omitted due to space limitations. We compare the query processing algorithm with and without the mapping dictionary functionality (MD and $NoMD$), using the *bound-is-easier* optimization in both cases. Figures 4.5(a) and 4.5(b) show two metrics concerning queries Q8 and Q4 respectively: bandwidth usage (left y-axis) and query response time (right y-axis). In the x-axis, we depict the number of



(a) Q8: Bandwidth and QRT (b) Q4: Bandwidth and QRT (c) Q4: Time allocation

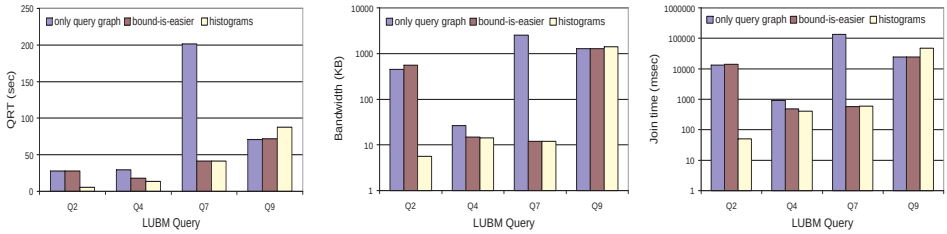
Figure 4.5: Query performance of a mapping dictionary

triples stored in the network. Due to the performance of the algorithm without the mapping dictionary, we show experiments with only up to 68,969 triples, which correspond to 10 departments of the LUBM University0. In the next section, we show experiments for the complete LUBM University0.

Bandwidth consumption. In both queries, we observe that MD achieves a significantly lower bandwidth consumption than $NoMD$. The size of the result set of Q8 grows with the number of triples stored and more messages are required for decoding the dictionary identifiers. Thus, the bandwidth usage of Q8 is also increasing with the number of triples as depicted in the graph. On the contrary, Q4 produces a constant number of results regardless of the number of triples stored which is determined from the first join using the bound-is-easier heuristic; hence, bandwidth consumption remains almost the same.

Query response time. The query response time (QRT) is shown with the lines in the graphs. We observe that the slope of the MD line (red dashed line) is smaller than the gradient of the $NoMD$ line (blue line), which means that the QRT of MD increases more smoothly than the QRT of $NoMD$ with the number of triples stored. The $ActualMD$ line (green dashed line) shows the query response time without the time needed for decoding the RDF components from the identifiers returned in the result set. Although the result set size of the queries affects the bandwidth consumption, it does not have an important impact on QRT. This is due to the fact that messages for decoding the dictionary identifiers are sent in parallel, so the load is distributed to almost every peer in the network. In both Figures 4.5(a) and 4.5(b) the QRT of $ActualMD$ is smaller than the QRT of MD by an almost constant factor. The only case where the decoding time deteriorated the QRT of MD, also depicted in Fig. 4.5(b), was at Q4 after having stored only 8,816 triples.

The total QRT for a certain query consists of the sum of the local processing time of each peer participating in the query processing plus the network time for transmitting the intermediate and the final results. The local processing time at each peer consists of the time for retrieving the triples matching a triple pattern from the local database, the time for joining intermediate results and the time needed for various local operations such as inference using the adorned rules of



(a) Query response time (b) Bandwidth usage (c) Join time

Figure 4.6: Query optimization performance

Table 4.1. In the case of MD, we should also take into consideration the time for decoding the identifiers at the end of the query evaluation. Figure 4.5(c) presents a stacked bar that shows these proportions of the total QRT for MD and $NoMD$ after having stored $68,969$ triples. We observe that for every time slice, MD outperforms $NoMD$. More significantly, we also observe that the most time-consuming operation in the query evaluation is retrieving triples from the peers' database. In our implementation, we have used the BerkeleyDB which is tightly integrated with the Bamboo DHT. From our experiments, we found that the time needed to retrieve a single triple from a peer's local database ranged from $0,1ms$ to $128ms$ for the different peers in Q4 and from $0,18ms$ to $2096ms$ in Q8 at the time of the experiments. As demonstrated in [4], slow nodes in PlanetLab can cause such problems. In a best case scenario, where all peers would need $0,1ms$ to retrieve one triple from their database (i.e., the smaller time we found in the experiment), the QRT of Q4 would be reduced by a factor of $2,3$ (i.e., from $37s$ to $16s$) while the QRT of Q8 would be reduced by $100s$. We plan to further investigate the behavior of our system's database and experiment with different database implementations or in-memory data structures for speeding up the retrieval of the triples.

Query optimization. In this section, we discuss how the performance of our system improves with respect to the chosen metrics with the use of the query optimizer. The metrics we use are the query response time, the bandwidth consumption and the time for joining the intermediate results. Motivated by the previous experiments we assume the use of a mapping dictionary. In this experiment, we have stored $103,314$ triples in the network which correspond to the complete LUBM University0. Figure 4.5 shows the performance of our system for several queries using the optimization techniques we described. In all graphs, the x-axis shows the LUBM queries while the y-axis depicts the metric of interest. The first bar in all graphs shows the performance of the query evaluation if the query graph is used to avoid Cartesian products but no other optimization is performed. The second and third bars depict the bound-is-easier and histograms heuristics respectively.

Figure 4.6(a) shows the QRT for different queries. In the graph, QRT also

contains the time required by the query optimizer for determining an execution plan. When using the bound-is-easier heuristic, this time is negligible, while when using the histograms the time ranged from $273ms$ to $451ms$ for the different queries. The different execution plans certainly can affect the join processing time and the bandwidth consumption of transmitting the different intermediate results. Since the peers that participate in the evaluation of a certain query are the same regardless of the execution plan and the triples that these peers will retrieve from their local database are the same, the total time spent for retrieving these triples and the RDFS reasoning performed is independent of the execution plan (this does not hold for a query with an empty result set). Figures 4.6(b) and 4.6(c) show on a logarithmic scale the total bandwidth consumption in $Kbytes$ and the total join processing time in ms.

Let us now explain the results that we see in Fig. 4.5 for each query. Q2 is a query that yields an empty result set caused by joining its last two triple patterns (triple patterns 5 and 6). For that reason, the query graph and the bound-is-easier heuristic, which choose the same execution plan where the join of these triple patterns is computed last, perform identically. Their performance is poor compared to using the histograms where a very selective triple pattern is chosen first and the join that yields an empty result is executed third.

Q4 is a star-shape query with its triple patterns sharing the same subject variable. The bound-is-easier and the histograms heuristics choose the same ordering for the first two triple patterns which actually determine the size of the final result set. Therefore the histograms heuristic performs slightly better. In Q7, all metrics are reduced significantly when using either heuristic since both generate the same execution plan. Finally, Q9 is a tricky query which contains a cycle join of its last three triple patterns. The combination of all triple patterns is the one that yields a small result set. So, even if the query optimizer took into consideration the selectivities of the joined triple patterns, it would still fail to find the optimal execution plan. It is by chance that the query graph and the bound-is-easier choose a better execution plan than the histograms heuristic.

4.6 Related Work

[5, 1, 29] consider RDF query processing on top of structured overlay networks, such as DHTs. [5] and [29] consider no RDFS reasoning while [1] provides semantic interoperability through schema mappings. In [18], we described how both forward with backward reasoning can be implemented on top of DHTs and presented an analytical and experimental evaluation. BabelPeers [3] is a DHT-based system that supports a forward chaining approach for RDFS reasoning. However, having experimented with forward chaining in [18], we believe that this approach cannot perform well in a real distributed environment since forward chaining not only has excessive space requirements but also cannot reach a fixpoint at a reason-

able amount of time. MARVIN [32] is another DHT-based system that performs RDF(S) reasoning using forward chaining. In contrast to the dynamic and uncontrollable environment we used for our experiments, MARVIN runs on DAS-3 (Distributed ASCI Supercomputer `http://www.cs.vu.nl/das3/`) and manages to scale to a considerable number of triples. Using an analytical model, the authors show that a fixpoint will be reached eventually but the time needed still remains questionable. Apart from this, the system uses an in-memory implementation at each peer which speeds up inferencing significantly.

Query optimization for SPARQL has been addressed only recently. In [46], the authors present a selectivity-based framework for optimizing SPARQL BGP queries. The optimization algorithm proposed is based on the minimum selectivity heuristic of [45]. The join selectivity is precomputed by executing SPARQL queries on every pair of properties that is related through the RDFS schema and keeping the size of the result sets. In the absence of an RDFS schema all combinations of distinct properties should be considered which becomes very expensive. In RDF-3X [30], the authors propose two kinds of statistics for the selectivity estimation of the joins: specialized histograms which can handle both triple patterns and joins, and the computation of frequent join paths in the RDF graphs. An interesting work that deals with query optimization in a distributed environment, although not a DHT, is presented in [47]. The authors of [47] address the problem of query optimization by defining a cost model and adopting randomized algorithms from the database literature for join ordering [45]. The join selectivity of two triple patterns is assumed to be estimated but no method is clearly proposed.

4.7 Summary

We presented some extensions to the system Atlas that we undertook recently so that we can have a more mature state-of-the-art implementation that could then be integrated with the centralized implementation of Chapter 3 to provide a distributed implementation of the SemsorGrid4Env registry. The new features in Atlas we presented in this chapter are:

- Being able to answer queries over RDF(S) databases; not just RDF databases as in our previous work [29, 20, 19].

- Using a dictionary encoding as in recent work on centralized RDF stores. In this respect, we are the first to use and evaluate this technique in a distributed implementation such as Atlas.

- Utilizing query optimization techniques based on histograms and various heuristics [48, 36].

Al the extensions of Atlas that we presented in this chapter have been full evaluated experimentally on PlanetLab.

Chapter 5

Conclusions and Future Work

This deliverable is the second deliverable of WP3 which will design, implement and deploy an open, dynamic and scalable registry for the SemsorGrid4Env software architecture defined in WP1.

In this deliverable, we presented a semantics and an algebra for stSPARQL in the spirit of the algebra for SPARQL given in [33]. This will be the algebraic language that will drive the stSPARQL implementation.

We also presented Strabon, the implementation of stSPARQL that is currently been developed in WP3 by extending the Sesame RDF store. Strabon will serve as the basis of the SemsorGrid4Env registry implementation. This implementation is currently centralized and it will be later on extended to a distributed one.

Finally, we presented data structures and algorithms for efficient distributed query processing in the system Atlas and a detailed performance evaluation of these on Planetlab. This work has resulted in a more mature implementation of Atlas which will be integrated with Strabon to provide a distributed implementation of the SemsorGrid4Env registry.

Our future work in WP3 will concentrate on the following:

- Extending Strabon to handle temporal information.
- Implementing the Semantic Registration and Discovery Service and integrating Strabon with the rest components of SemsorGrid4Env.
- Provide detailed performance evaluation in the context of the software architecture of WP1 and the use cases of the project. In addition, the performance of distributed query processing algorithms that will be developed will be evaluated experimentally on PlanetLab.

Bibliography

[1] K. Aberer, P. Cudre-Mauroux, M Hauswirth, and T. V. Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *ISWC 2004*.

[2] James F Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] D. Battre, A. Hoing, F. Heine, and O. Kao. On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores. In *DBISP2P 2006*.

[4] Sean Rhea Byung-Gon, Sean Rhea, Byung gon Chun, John Kubiatowicz, and Scott Shenker. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *WORLDS 2005*.

[5] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW 2004*.

[6] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB 2005*.

[7] Zhan Cui, Anthony G. Cohn, and David A. Randell. Qualitative and Topological Relationships in Spatial Databases. In David J. Abel and Beng Chin Ooi, editors, *Advances in Spatial Databases*, volume 692 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 1993.

[8] Max J. Egenhofer. Toward the Semantic Geospatial Web. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 1–4, New York, NY, USA, 2002. ACM Press.

[9] Max J. Egenhofer and Robert D. Franzosa. Point-Set Topological Spatial Relations. In *Proceedings of International Journal of Geographical Information Systems*, volume 5, pages 161–174, 1991.

[10] Alasdair J G Gray, Ixent Galpin, Alvaro A A Fernandes, , Norman W. Paton, Kevin Page, Jason Sadler, Manolis Koubarakis, Kostis Kyzirakos,

Jean-Paul Calbimonte, Oscar Corcho, Raul Garcia, Victor M Diaz, and Israel Liebana. SemsorGrid4Env architecture - phase i. Deliverable D3.1 Version 1.0, SemSorGrid4Env, August 2009.

[11] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[12] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2007.

[13] Claudio Gutierrez, Carlos Hurtado, and Ro Vaisman. Temporal RDF. In *European Conference on the Semantic Web (ECSW05)*, pages 93–107, 2005.

[14] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *Third Latin American Web Congress*, 2005.

[15] P. Hayes. RDF Semantics. W3C Recommendation, February 2004.

[16] Carlos A. Hurtado and Alejandro A. Vaisman. Reasoning with Temporal Constraints in RDF. In *Principles and Practice of Semantic Web Reasoning*, pages 164–178. Springer, 2006.

[17] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 299–313, 1990.

[18] Z. Kaoudi, I. Miliaraki, and M. Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *ISWC 2008*.

[19] Z. Kaoudi, I. Miliaraki, M. Magiridou, A. Papadakis-Pesaresi, E. Liarou, S. Idreos, , S. Skiadopoulos, and M. Koubarakis. Deployment of Ontology Services and Semantic Grid Services on top of Self-organized P2P Networks. Deliverable D4.2v1, Ontogrid project, February 2005.

[20] Z. Kaoudi, I. Miliaraki, S. Skiadopoulos, M. Magiridou, E. Liarou, S. Idreos, and M. Koubarakis. Specification and Design of Ontology Services and Semantic Grid Services on top of Self-organized P2P Networks. Deliverable D4.1, Ontogrid project, September 2005.

[21] Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Semantic Grid Resource Discovery in Atlas. *Knowledge and Data Management in Grids*, 2006. Talia Domenico and Bilas Angelos and Dikaiakos Marios D. (editors), Springer.

[22] Dave Kolas. Supporting Spatial Semantics with SPARQL. In *Terra Cognita Workshop (Terra Cognita 2008)*, Karlsruhe, Germany, 2008.

[23] Dave Kolas and Troy Self. Spatially Augmented Knowledgebase. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007)*, Berlin, Heidelberg.

[24] Bart Kuijpers. Linear versus Polynomial Constraint Databases. In Shashi Shekhar and Hui Xiong, editors, *Encyclopedia of GIS*, pages 612–615. Springer, 2008.

[25] Gabriel Kuper, Leonid Libkin, and Jan Paredaens. *Constraint Databases*. Springer Verlag, 2000.

[26] Gabriel Kuper, Sridhar Ramaswamy, Kyuseok Shim, and Jianwen Su. A Constraint-based Spatial Extension to SQL. In *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, 1998.

[27] Kostis Kyzirakos, Manolis Koubarakis, and Zoi Kaoudi. Data models and languages for registries in SemsorGrid4Env. Deliverable D3.1 Version 1.0, SemSorGrid4Env, August 2009.

[28] Robert Laurini and Derek Thompson. *Fundamentals of spatial informations systems*. The A.P.I.C. Series no. 37 - Academic Press, 1992.

[29] Erietta Liarou, Stratos Idreos, and Manolis Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In *Proceedings of 5th the International Semantic Web Conference (ISWC 2006)*, Athens, GA, USA, November 2006.

[30] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. In *VLDB'08*.

[31] N. Ntarmos, P. Triantafillou, and G. Weikum. Distributed Hash Sketches: Scalable, Efficient, and Accurate Cardinality Estimation for Distributed Multisets. *ACM Transactions on Computer Systems*, 2009.

[32] Eyal Oren, Spyros Kotoulas, George Anadiotis, Ronny Siebes, Annette ten Teije, and Frank van Harmelen. MARVIN: A platform for large-scale analysis of Semantic Web data. In *WebSci'09*.

[33] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. In *Proceedings of the 5th International Semantic Web Conference (ISWC 2006)*, pages 30–43, 2006.

[34] Matthew Perry. *A Framework to Support Spatial, Temporal and Thematic Analytics over Semantic Web Data*. PhD thesis, Wright State University, 2008.

[35] Agnes Voisard Philippe Rigaux, Michel Scholl. *Spatial Databases*. Morgan Kaufmann Publishers, 2001.

[36] Viswanath Poosala, Yannis Ioannidis, Peter Haas, and Eugene Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *ACM SIGMOD 1996*.

[37] E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF.

[38] Peter Z. Revesz. *Introduction to Constraint Databases*. Springer, 2002.

[39] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference 2004*.

[40] Philippe Rigaux, Michel Scholl, Luc Segoufin, and Stéphane Grumbach. Building a constraint-based spatial database system: model, languages, and implementation. *Information Systems*, 28(6):563–595, 2003.

[41] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price.

[42] A. Sheth, C. Henson, and S. S. Sahoo. Semantic Sensor Web. *Internet Computing, IEEE*, 12(4):78–83, 2008.

[43] R. Singh, A. Turner, M. Maron, and A. Doyle. GeoRSS: Geographically Encoded Objects for RSS Feeds. `http://georss.org/gml`, 2008.

[44] R. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 236–246, 1985.

[45] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *The VLDB Journal*, 1997.

[46] Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL Basic Graph Pattern Optimization using Selectivity Estimation. In *WWW 2008*.

[47] Heiner Stuckenschmidt, Richard Vdovjak, Jeen Broekstra, Geert jan Houben, Tu Eindhoven, and Aduna Amersfoort. Towards Distributed Processing of RDF Path Queries. *Int. J. Web Engineering and Techonology*, 2005.

[48] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I and II*. Computer Science Press, 1988.

[49] Luc Vandeurzen, Marc Gyssens, and Dirk Van Gucht. On the expressiveness of linear-constraint query languages for spatial databases. *Theoretical Computer Science*, 254(1-2):423–463, 2001.

[50] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB 2008*.