

SemSorGrid4Env

FP7-223913



Deliverable

D3.3v2

Implementation and deployment of the registry services - Phase II

Manos Karpathiotakis, Kostis Kyzirakos, Zoi Kaoudi, Vissarion Fisikopoulos, Iris Miliaraki, Manolis Koubarakis, Yannis Ioannidis, and Michael Hatzopoulos
National and Kapodistrian University of Athens

Vana Kalogeraki
Athens University of Economics and Business

February 24, 2011



Executive Summary

The objective of WP3 is to design, implement and deploy an open, dynamic and scalable registry for the SensorGrid4Env 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, services etc. In this deliverable we present the details of the centralized and distributed implementations of stSPARQL, nick-named Strabon and Atlas2, that form the basis of the SensorGrid4Env Semantic Registry implementation. We also give details of the source code of Strabon and Atlas2, provide some sample stRDF data and a set of stSPARQL queries that are supported by the current versions of our implementations. We will also describe the steps that are necessary to build, install and run Strabon and Atlas2. Finally, we will give a brief description of the testing strategy we followed.



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 occasionally 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 contains material from deliverable D3.2[30]
- Section 3 contains material from deliverable D3.3v1[32]
- Section 5 contains material from deliverable D3.3v1[32]



Document Information


Contract Number	FP7-223913	Acronym	SemSorGrid4Env
Full title	SemSorGrid4Env: Semantic Sensor Grids for Rapid Application Development for Environmental Management		
Project URL	www.sensorsgrid4env.eu		
Document URL	www.sensorsgrid4env.eu/intranet/servlet/download?ontology=Documentation+Ontology&concept=Deliverable&instanceSet=SemSorGrid4Env&instance=D3.3+v2&attribute=On-line+PDF+Version&value=D3.3v2-final.pdf		
EU Project Officer	Antonios Barbas		

Deliverable	Number	D3.3v2	Name	Implementation and deployment of the registry services - Phase II		
Task	Number	T3.3	Name	Implement and deploy the SensorGrid4Env registry services		
Work package	Number			WP3		
Date of delivery	Contract	28/2/2011	Actual	28/2/2011		
Code name	D3.3v2		Status	draft <input type="checkbox"/>	final <input checked="" type="checkbox"/>	
Nature	Prototype <input checked="" type="checkbox"/> Report <input type="checkbox"/> Specification <input type="checkbox"/> Tool <input type="checkbox"/> Other <input type="checkbox"/>					
Distribution Type	Public <input checked="" type="checkbox"/> Restricted <input type="checkbox"/> Consortium <input type="checkbox"/>					
Authoring Partner	National and Kapodistrian University of Athens					
QA Partner	University of Southampton					
Contact Person	Kostis Kyzirakos					
	Email	kkyzir@di.uoa.gr	Phone	+30 210 727 5159	Fax	
Abstract (for dissemination)	In this deliverable we present the centralized and distributed implementations of stSPARQL, nick-named Strabon and Atlas2, that form the basis of the SensorGrid4Env registry implementation. We will describe the source code of Strabon and Atlas2, provide some sample stRDF data and a set of stSPARQL queries that are supported by the current version of our implementations. We will also describe the steps that are necessary to build, install and run Strabon and Atlas2.					
Keywords	Registry, Strabon, stRDF, stSPARQL, Atlas2					

Version log/Date	Change	Author
0.1 / 7 January 2011	Contents	M.Koubarakis
0.8 / 17 February 2011	Final draft to be QAed	K. Kyzirakos, M. Karpathiotakis, Z.Kaoudi, M. Koubarakis, V. Fisikopoulos, I.Miliaraki, Y.Ioannidis, M.Hatzopoulos and V.Kalogeraki
0.9 / 24 February 2011	Response to QA comments by Alex Frazer	K. Kyzirakos, M. Karpathiotakis, Z.Kaoudi, M. Koubarakis
1 / 24 February 2011	Final Version - Response to QA verification comments by UPM	K. Kyzirakos, M. Karpathiotakis, Z.Kaoudi, M. Koubarakis

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 #@ asun@fi.upm.es #t +34-91 336-7439, #f +34-91 352-4819
The University of Manchester	UNIMAN 	Prof. Norman Paton Department of Computer Science The University of Manchester Oxford Road Manchester, M13 9PL, United Kingdom #@ npaton@cs.man.ac.uk #t +44-161-275-69 10, # +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 	Dr. Kirk Martinez University of Southampton Southampton SO17 1BJ United Kingdom #@ km@ecs.soton.ac.uk #t +44 23 80594491, #f +44 23 80592865
Deimos Space, S.L.U.	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 	Dr. 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

1	Introduction	1
2	Data Model and Query Language	3
2.1	The stRDF Data Model	3
2.1.1	Linear constraints	3
2.1.2	The sRDF data model	3
2.1.3	The stRDF Data Model	4
2.2	The Language stSPARQL	4
2.3	The variables NOW, PAST and FUTURE	7
2.4	Summary	8
3	The System Strabon	9
3.1	The Architecture of Strabon	9
3.2	Storing stRDF data	10
3.3	Evaluating stSPARQL queries	13
3.4	Summary	14
4	The system Atlas2	15
4.1	System architecture	15
4.2	Storing sRDF data	16
4.3	Evaluating sSPARQL queries	17
4.3.1	Query model	17
4.3.2	Query evaluation algorithm	18
4.4	Query optimization	21
4.5	Summary	21
5	The Semantic Registration and Discovery Service	22
5.1	The SemsorGrid4Env Web service architecture	22
5.2	Interfaces	23
5.3	Example orchestrations in the SemsorGrid4Env architecture	24
5.4	Summary	28
6	Source code	29
6.1	Strabon	29
6.1.1	Directory structure	29
6.1.2	External Dependencies	30
6.1.3	Application Programming Interface	30
6.2	Atlas2	30
6.2.1	Directory structure	31
6.2.2	Implementation details	31
6.2.3	Application Programming Interface	32
6.3	Semantic Registry	33
6.3.1	Directory structure	33
6.3.2	External Dependencies	33
6.3.3	Application Programming Interface	33
6.4	Summary	34
7	Installation and execution	35
7.1	Strabon	35
7.1.1	Installing PostgreSQL	35
7.1.2	Installing PostGIS	35
7.1.3	Creating a spatially enabled database	35
7.1.4	Compiling and running Strabon	36
7.2	Atlas2	37



7.2.1	Compiling and running Atlas2	38
7.3	Semantic Registry	40
7.3.1	Installing Apache Tomcat	40
7.3.2	Compiling and running the Semantic Registry	40
7.4	Summary	41
8	Testing Strategy	42
8.1	Testing in Strabon	42
8.2	Testing in Atlas2	42
8.3	Conclusions	43
9	Conclusions	44



List of Figures

3.1	The Strabon system architecture	10
3.2	Storing a spatial geometry in Strabon	11
3.3	Query evaluation	13
4.1	Atlas2 node architecture	16
4.2	Query evaluation in Atlas2	19
5.1	Service-oriented architecture	22
5.2	Implementing the Semantic Registration and Discovery Service using Strabon or Atlas2	23
5.3	Service Ontology	24



List of Tables

5.1	Namespaces used and corresponding prefixes	25
8.1	Coverage on unit test for primary modules	42



1. Introduction

The objective of WP3 is to design, implement and deploy an open, dynamic and scalable Semantic Registry for the SensorGrid4Env software architecture defined in WP1. The Semantic Registry that has been developed allows the description and discovery of Semantic Sensor Grid resources: sensors, sensor networks, data sources, services, ontologies etc.

In this report we present both a centralized and a distributed implementation of the SensorGrid4Env Semantic Registry that are the results of the technical work carried out in WP3. We start by reminding the reader about what we presented in earlier reports and then proceed to give details of the contributions of the present report.

In Deliverable D3.1 [31] we studied the problem of designing a data model and a query language for the registry of the SensorGrid4Env infrastructure and more generally for a registry in a Semantic Sensor Web or Grid infrastructure. 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 SensorGrid4Env Semantic 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 Deliverable D3.2 [30], we presented a semantics and an algebra for stSPARQL. We also presented Strabon, an implementation of stSPARQL that is the basis of the centralized SensorGrid4Env registry implementation. We also presented data structures and algorithms for efficient distributed query processing in the system Atlas¹ [25] and a detailed performance evaluation of these on Planet-lab². Atlas is a P2P system for the distributed storage of RDF data and the processing of SPARQL queries. Using the knowledge gained from the centralized implementation of Strabon, we would also develop a distributed implementation of stRDF and stSPARQL by extending Atlas.

In Deliverable D3.3v1 [32], we presented some implementation details of Strabon v0.2. We provided a sample dataset and a set of queries that were supported by the version of Strabon that was current at that time. In addition, we gave a short tutorial on how to compile and run Strabon, store stRDF files and evaluate stSPARQL queries.

In the present report D3.3v2, we discuss the details of the implementation of Strabon v0.8, as well as the details of an extension implementation of Atlas, called Atlas2, that supports the model stRDF and the query language stSPARQL. We also discuss how the Semantic Registry of the SensorGrid4Env infrastructure (also called Semantic Registration and Discovery Service) has been implemented on top of Strabon and Atlas2 giving rise to the two implementations that we have envisaged since the beginning of the project. We also give a short tutorial on how to compile and run both Strabon and Atlas2. This report accompanies the code for our implementations which is available for download from the SensorGrid4Env SVN server³. In other words, the complete deliverable D3.3v2 is this report plus code.

The performance of the two implementations (Strabon and Atlas2) presented in this deliverable will be evaluated experimentally in the final months of SensorGrid4Env and the results will be presented in Deliverable 3.4 “Evaluation of the registry services”

The organization of this report is as follows. In Chapter 2 we present the stRDF data model and the language stSPARQL. We include this presentation so that the report is self-contained. In Chapter 3 we present the implementation of Strabon. In Chapter 4 we present the implementation of Atlas2. In Chapter 5 we present the implementation of the Semantic Registry on top of Strabon and Atlas2. In Chapter 6 we provide a description of the source code of our two implementations.

¹<http://atlas.di.uoa.gr/>

²<http://www.planet-lab.org>

³<https://sensorgrid.techideas.net/repos/>



In chapter 7 we provide instructions on compiling and running the source code. In chapter 8 we give a brief description of our testing strategy. Finally, in Chapter 9 we conclude this deliverable and discuss future work.

In the context of this report, users will be referred to using *he* or *she* interchangeably.

2. Data Model and Query Language

In this chapter we present the data model stRDF and the query language stSPARQL. The contents of this chapter have been previously presented in [28, 30] and are included in this report so that it is self-contained.

2.1 The stRDF Data Model

To develop stRDF, we follow closely the ideas of constraint databases [22, 41] and especially the work on SQL [29]. 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.

2.1.1 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, dense and unbounded 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 $x_i + \dots + x_i$ (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 $\phi(x_1, \dots, x_k)$ of \mathcal{L} where x_1, \dots, x_k are variables such that $(a_1, \dots, a_k) \in S$ iff $\phi(a_1, \dots, a_k)$ is true in the structure \mathcal{Q} .*

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

2.1.2 The sRDF data model

We now define sRDF. As in theoretical treatments of RDF [39], 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, \dots that are pairwise-disjoint with I, B and L . The elements of each $C_k, k = 1, 2, \dots$ are the quantifier-free formulae of the first-order language \mathcal{L} with k free variables. We denote with C the infinite union $C_1 \cup C_2 \cup \dots$.

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 .



Example 1. *The following are sRDF triples :*

```
ex:s1 rdf:type, ex:Sensor . ex:s1 ex:has_location "x=10 and  
y=20"^^strdf:SemiLinearPointSet
```

The above triples define a sensor and its location 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 is e.g., `strdf:SemiLinearPointSet` (see Example 1 above) and can be defined using XML Schema. Alternatively, linear constraints can be expressed in RDF using MathML¹ and serialized as `rdf:XMLLiterals` as in [38]. [38] specifies a syntax and semantics for incorporating linear equations in OWL 2. We now move on to define stRDF.

2.1.3 The stRDF Data Model

We will now extend sRDF with time. Database researchers have differentiated among valid time, transaction time and user-defined time. Valid time is the time a piece of information is true in the real world. Transaction time is the time a transaction changing the current database state is committed, whereas user-defined time is a time that appears in the database and has user-defined semantics e.g., a person's date of birth.

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. [16] where a fourth component is added to each sRDF triple.

The *time structure* that we assume in stRDF is the set of rational numbers \mathbb{Q} (i.e., time is assumed to be linear, dense and unbounded). Temporal constraints are expressed by quantifier-free formulas of the language \mathcal{L} defined earlier, but their syntax is limited to elements of the set C_1 . *Atomic* temporal constraints are formulas of \mathcal{L} of the following form: $x \sim c$, where x is a variable, c is a rational number and \sim is $<$, \leq , \geq , $>$, $=$ or \neq . *Temporal constraints* are Boolean combinations of atomic temporal constraints using a *single* variable.

The following definition extends the concepts of triple and graph of sRDF so that thematic and spatial data with a temporal dimension can be represented.

Definition 3. *An stRDF quad is an sRDF triple (a, b, c) with a fourth component τ which is a temporal constraint. For quads, we will use the notation (a, b, c, τ) , where the temporal constraint τ defines the set of time points that the fact represented by the triple (a, b, c) is valid in the real world. An stRDF graph is a set of sRDF triples and stRDF quads.*

2.2 The Language stSPARQL

We present the syntax of stSPARQL by means of examples involving sensor networks. More examples of stSPARQL from a GIS perspective are given in [31].

¹<http://www.w3.org/Math/>, last accessed February 20, 2010.



We will consider a dataset that describe static and moving sensors and use the CSIRO/SSN Ontology [34] to describe them . The main classes of interest in the SSN ontology are the class *Feature* that describes the observed domain, the class *Sensor* that describes the sensor, the class *SensorGrounding* that describes the physical characteristics and the location of the sensor and the class *Location* that is self explained. We extend the aforementioned ontology with the properties `strdf:hasGeometry` and `strdf:hasTrajectory` with range `strdf:SemiLinearPointSet`.

The stRDF description of a static sensor that measures temperature and has a certain location is the following (ssn is the namespace of the CSIRO/SSN ontology and ex an example ontology):

```
ex:sensor1 rdf:type ssn:Sensor .
ex:sensor1 ssn:measures ex:temperature .
ex:temperature ssn:type ssn:PhysicalQuality .
ex:sensor1 ssn:supports ex:grounding1 .
ex:grounding1 rdf:type ssn:SensorGrounding .
ex:grounding1 ssn:hasLocation ex:location1 .
ex:location1 rdf:type ssn:Location .
ex:location1 strdf:hasGeometry "x=10 and y=10"^^strdf:SemiLinearPointSet .
```

We choose to use the O&M-OWL ontology [19] to represent sensor observations. However, since we use stRDF to model space and time, we choose not to use the classes *Time*, *TimeInterval* and *TimeInstant* that come from OWL-Time and the classes *Geometry* and *Point* that come from GML. So our modeling is similar to the modeling in [19] but instead of relying on OWL-Time and GML we rely on the stRDF constructs. The stRDF representation of the sensor's observations is the following (om is the namespace of the O&M-OWL ontology):

```
ex:sensor1 rdf:type ex:TemperatureSensor .
ex:TemperatureSensor rdf:subClassOf om:Sensor .
ex:obs1 rdf:type om:Observation .
ex:obs1 om:procedure ex:sensor1 .
ex:obs1 om:observedProperty ex:temperature .
ex:temperature rdf:type om:Property .
ex:obs1 om:observationLocation ex:obslocation1 .
ex:obslocation1 rdf:type om:Location .
ex:obslocation1 strdf:hasGeometry "x=10 and y=10"^^strdf:SemiLinearPointSet .
ex:obs1 om:result ex:obs1Result .
ex:obs1Result rdf:type om:ResultData .
ex:obs1Result om:uom ex:Celcius .
ex:obs1Result om:value "27" "(10 <= t <= 11)"^^strdf:SemiLinearPointSet .
```

Notice the last quad that capture the spatiotemporal information.

Let us now present an example of modeling *moving sensors* in stRDF. Note that trajectories of moving sensors are easily represented in stRDF.

```
ex:sensor2 rdf:type ssn:Sensor .
ex:sensor2 ssn:measures ex:temperature .
ex:sensor2 ssn:supports ex:grounding2 .
ex:grounding2 rdf:type ssn:SensorGrounding .
ex:grounding2 ssn:hasLocation ex:location2 .
ex:location2 rdf:type ssn:Location .
ex:location2 strdf:hasTrajectory
  "(x=10t and y=5t and 0<=t<=5) or
  (x=10t and y=25 and 5<=t<=10)"^^strdf:SemiLinearPointSet .
```



Finally, we assume that we have the stRDF descriptions of some rural area where the sensors are deployed. The stRDF description of such an area called Brovallen is the following:

```
ex:areal rdf:type ex:RuralArea .
ex:areal ex:hasName "Brovallen" .
ex:areal strdf:hasGeometry
    "(-10x+13y<=-50 and y<=79 and y>=13 and
      x<=133) or (y<=13 and x<=133 and
      x+2y>=129)"^^strdf:SemiLinearPointSet .
```

Example 2. Spatial selection. *Find the URIs of the static sensors that are inside the rectangle $R(0,0,100,100)$*

```
select ?S
where {?S rdf:type ssn:Sensor .
      ?G rdf:type ssn:SensorGrounding .
      ?L rdf:type ssn:Location .
      ?S ssn:supports ?G .
      ?G ssn:haslocation ?L .
      ?L strdf:hasGeometry ?GEO .
      filter(?GEO inside "0<=x<=100 and 0<=y<=100")}
```

Let us now explain the new features of stSPARQL by referring to the above example. stSPARQL has a new kind of variables called *spatial variables*. Spatial variables can be used in basic graph patterns 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 terms include spatial constants (finite representations of semi-linear sets e.g., "0<=x<=10 and 0<=y<=10"), spatial variables and complex spatial terms (e.g., ?GEO INTER "x=10 and y=10" which denotes the intersection of the value of spatial variable ?GEO and the semi-linear set "x=10 and y=10"). 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 [9] can be used as predicates in a spatial filter expression e.g., filter(?GEO1 inside ?GEO2).

Example 3. Temporal selection. *Find the values of all observations that were valid at time 11 and the rural area they refer to.*

```
select ?V ?RA
where {?OBS rdf:type om:Observation .
      ?LOC rdf:type om:Location .
      ?R rdf:type om:ResultData .
      ?RA rdf:type ex:RuralArea .
      ?OBS om:observationLocation ?LOC .
      ?OBS om:result ?R .
      ?R om:value ?V ?T .
      ?LOC strdf:hasGeometry ?OBSLOC .
      ?RA strdf:hasGeometry ?RAGEO .
      filter(?T contains (t = 11) && ?RAGEO contains ?OBSLOC)}
```

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 variable in addition to spatial ones: *temporal variables*. 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 temporal filters, a new kind of filter that can be used in stSPARQL to constrain the valid time of triples.



The expressions that make up temporal filters are Boolean combinations of *interval predicates* that are used to compare temporal terms. A *temporal term* in stRDF is a temporal variable or a temporal constant (i.e., an element of the set C_1 e.g., "(t>=0 and t<=2) or (t>=5 and t<=7)"). We allow any of the thirteen interval relations identified by Allen in [2] to be used as the interval predicates e.g, *contains* in the above example .

Example 4. Intersection of an area with a trajectory. *Which areas of Brovallen were sensed by a moving sensor and when?*

```
select (?TR[1,2] INTER ?GEO) as ?SENSEDAREA ?GEO[3] as ?T1
where {?SN rdf:type ssn:Sensor .
      ?RA rdf:type ex:RuralArea.
      ?X rdf:type ssn:SensorGrounding .
      ?Y rdf:type ssn:Location.
      ?SN ssn:supports ?X .
      ?X ssn:hasLocation ?Y.
      ?Y strdf:hasTrajectory ?TR .
      ?RA ex:hasName "Brovallen".
      ?RA strdf:hasGeometry ?GEO .
      filter(?TR[1,2] overlap ?GEO)}
```

The above query demonstrates the projection of spatial terms. Projections of spatial terms (e.g., ?TR[1,2]) denote the projections of the corresponding point sets on the appropriate dimensions, and are written using the notation Variable "[" Dimension1 "," ... "," DimensionN "].

Example 5. Projection and spatial function application. *Find the URIs of the sensors that are north of Brovallen.*

```
select ?SN
where {?SN rdf:type ssn:Sensor .
      ?X rdf:type ssn:SensorGrounding .
      ?Y rdf:type ssn:Location .
      ?RA rdf:type ex:RuralArea .
      ?RA ex:hasName "Brovallen" .
      ?RA strdf:hasGeometry ?GEO .
      ?SN ssn:supports ?X .
      ?X ssn:hasLocation ?Y .
      ?Y strdf:hasGeometry ?SN_LOC .
      filter(MAX(?GEO[2])<MIN(?SN_LOC[2]))}
```

The above query demonstrates the projection of spatial terms and the application of metric spatial functions to spatial terms. We allow expressions like MAX(?GEO[2]) that return the maximum value of the unary term ?GEO[2].

2.3 The variables NOW, PAST and FUTURE

The temporal features of stRDF presented in Section 2.1.3 were introduced with the expectation that they will be used very often in the use cases of the project. In fact, only the concept of user-defined time was required by the use cases. In addition, the Flood use case required the introduction of *temporal variables* NOW, PAST and FUTURE and the creation of time instances and intervals that use these variables.

For example, a tide monitoring service could provide us with information that is current for a 12-hour window starting from the present time. The time interval capturing this window can be written as [NOW, NOW + 12 hours] using the temporal variable NOW. The following triples give a precise example from the Flood use case:



```
Demo:ModelledTideHeightService
  a Services:WebService;
  Services:hasDataset Demo:ModelledTideHeightDataset.
```

```
Demo:ModelledTideHeightDataset
  a Services:Dataset;
  rdfs:label "Modelled tide height Dataset";
  time:hasTemporalExtent "[NOW,NOW+12]"^^strdf:TemporalInterval.
```

Similarly, a user could query the Semantic Registry in order to locate Web services that could provide him with any information available for an 8-hour window starting at the time he executes his query. A possible query he could pose is the following:

```
SELECT ?SERVICE
WHERE { ?SERVICE rdf:type Services:WebService .
        ?SERVICE Services:hasDataset ?DATASET .
        ?DATASET time:hasTemporalExtent ?TIME .
        FILTER(?TIME contains
               "[NOW,NOW+8]"^^strdf:TemporalInterval).}
```

This query's result would be the service `Demo:ModelledTideHeightService` described in the above triples.

Let us now explain why it is reasonable to expect that the query evaluation will produce the above result. Since temporal variables cannot be directly stored in a traditional RDBMS, each one must first be bound to a real-world time instant. According to [7], whose approach we followed, the temporal variable `NOW` should be bound to the database observer's reference time. In our case, this "reference time" is always the time a query is posed. The time interval `[NOW, NOW + H]` is the interval that starts the time `T` the query is asked and ends `H` hours later. The hour granularity we assume has to do with the Flood use case, according to which potential users of the Semantic Registry are only interested in hourly deviations from the current time when using `NOW`-relative time instants. In general, `H` could be a value from the datatype `xdt:dayTimeDuration`.

So, the time interval `[NOW, NOW + 12]` is mapped to the real-world time window `[12:35:06, 12:35:06 + 12 hours]` if `12:35:06` is the time the query was asked.

Now that we have explained the exact meaning of this tuple after all its temporal variables have been bound, we can likewise deduce that the time interval inside the previous query's filter clause actually refers to the real-world time window `[12:35:06, 12:35:06 + 8 hours]`, which is clearly contained in the larger `[12:35:06, 12:35:06 + 12 hours]` window of our previous triple.

2.4 Summary

In this chapter we presented the data model `stRDF` and the query language `stSPARQL`. We also discussed the introduction of variables `NOW`, `PAST` and `FUTURE` that were required by the Flood use case.

3. The System Strabon

In this section we present Strabon, a storage and query evaluation module for stRDF/stSPARQL which is currently under development by our group in the context of the SemsorGrid4Env project [44]. In the first prototype of Strabon, we only supported the model sRDF (i.e., there is no support for time) and semi-linear sets of dimension at most 2 (i.e., only spatial data in 2 dimensions are supported). This was a realistic scenario that allowed us to capture all the spatial data of SemsorGrid4Env (e.g., spatial coverage for data sources exposing UK's national Spatial Data Infrastructure datasets, programmatic data and commercial product datasets). In this deliverable, we also present the temporal capabilities Strabon has been enhanced with.

The organization of this chapter is as follows. In Section 3.1 we present the conceptual architecture of Strabon. In Section 3.2 we thoroughly examine the process followed when stRDF data are to be stored, and finally in Section 3.3 we present the flow of operations that take place during query processing in our implementation.

3.1 The Architecture of Strabon

Strabon is built on top of the well-known RDF store Sesame [45] and extends Sesame's components to be able to manage thematic and spatial metadata that are stored in PostGIS. We chose to base our system on Sesame since its layered architecture allows implementations on top of a wide variety of repositories without changing any of Sesame's other components. In Strabon, the repository is the PostGIS DBMS that gives us many of the needed functionalities for spatial data¹. Sesame has the ability to stack layers and allows us to monitor the layers with the use of listeners. The listeners provide the ability to intercept the query evaluation process or the storing of a triple so that additional tasks are performed. The Sesame architecture consists of the following layers (some of which are shown in Figure 3.1 in the context of the Strabon architecture): Access APIs and Storage and Inference Layer (SAIL) APIs. The most important Access API is the Repository API that is used for querying and updating data. 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 provides RDF-specific methods to access RDF data that are translated to calls to the underlying database.

We would like to stress that although a new generation of RDF stores recently proposed by the database community [1, 46, 49, 35] has been shown to be more efficient than Sesame, the features of the software architecture of Sesame outlined above, its maturity as an open source RDF store, and its wide user base affected our decision to adopt it for the first implementation of Strabon. In the future, we plan to experiment with implementations which are based on more recently proposed RDF stores.

Figure 3.1 presents the system architecture of Strabon which consists of the following modules:

- *Query Engine*: This module is used to evaluate stSPARQL queries posed by clients. The Query Engine receives all the stSPARQL queries, parses them, optimizes them, prepares an execution plan and returns the appropriate response to the client. We have extended Sesame's Parser and Evaluator to handle stSPARQL queries and use Sesame's Optimizer and Transaction Manager as is. In Section 3.3 we describe how the Query Engine evaluates stSPARQL queries.

¹Ideally, our repository should have been a constraint database such as CSQL [29], Dédale [15], or MLPQ [40]. Unfortunately, CSQL has never been implemented, the Dédale code is not readily available as we learned from Philippe Rigaux, and MLPQ, although it is available from Peter Revesz, is not an open source system something that is a requirement for us in the context of project SemsorGrid4Env.

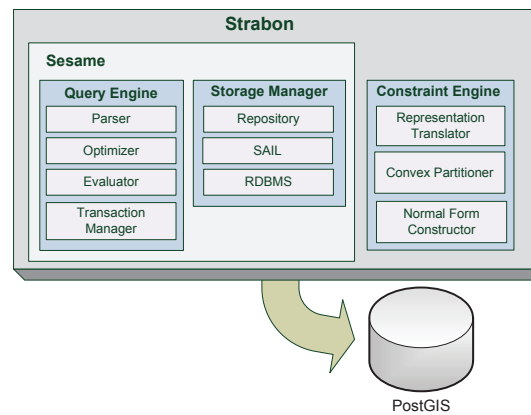


Figure 3.1: The Strabon system architecture

- *Storage Manager*: This module is responsible for storing data in the underlying PostGIS database. The *Repository*, *SAIL* and *RDBMS* layers are Sesame’s modules that were described above. We have extended the *RDBMS* layer to be able to store spatial literals in PostGIS. In Section 3.2 we provide more details about the Storage Manager.
- *PostGIS*: Strabon uses PostGIS to store stRDF data and to perform part of the query evaluation process as we will show in Section 3.3.
- *Constraint Engine*: This module is responsible for processing the part of the stSPARQL queries that deal with geometries and the conversions that take place during data loading. Since we want our implementation to cater for the case that input spatial objects are expressed in other spatial data models (e.g., using OGC standards), the *Representation Translator* component has been introduced to translate spatial objects between the constraint and other equivalent representations. Spatial objects that represent non-convex geometries are convexified prior to storage to take advantage of efficient computational geometry algorithms for convex geometries. This is the job of the *Convex Partitioner* module. The *Normal Form Constructor* takes the output of the Representation Translator or the Convex Partitioner and constructs a geometry in a normal form that we describe in Section 3.2 below.

3.2 Storing stRDF data

When a user wants to store stRDF data, she makes them available in the form of an stRDF document. The document is decomposed into stRDF triples and each triple is stored in the underlying repository as we explain below.

Sesame supports two storage schemes for pure RDF data. A “monolithic” scheme where all triples are stored in a giant *Triple* table, and a vertical partitioning scheme that stores triples using one table per predicate. In both cases, the data is stored using *dictionary encoding* i.e., each URI or literal is encoded by a unique positive integer to reduce space. The mapping between the original RDF term and its encoding is stored in a separate table.

In Strabon, the storage scheme of Sesame is extended with an extra table that stores detailed information about spatial literals. In addition, spatial literals are encoded using the same dictionary encoding techniques. If the “monolithic” storing scheme is used, all stRDF triples are stored in the *Triple* table. The schema of the *Triple* table is given below.

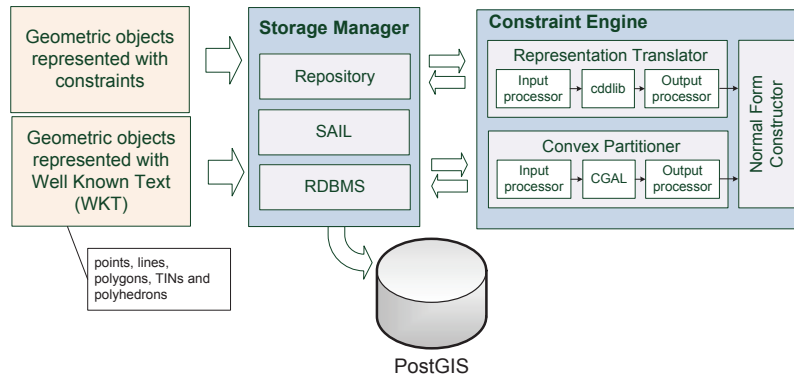


Figure 3.2: Storing a spatial geometry in Strabon

Schema of table *Triple*

Attribute	Type	Comment
<i>subj_id</i>	integer	The unique encoding for the subject.
<i>pred_id</i>	integer	The unique encoding for the predicate.
<i>obj_id</i>	integer	The unique encoding for the object.

Each tuple in the table *Triple* has a *subj_id* (resp. *pred_id*, *obj_id*) that is the unique encoding of the triple's subject (resp. predicate, object).

An additional table *SpatialLiteral* is used to store information about spatial literals. The schema of the table *SpatialLiteral* is given below.

Schema of table *SpatialLiteral*

Attribute	Type	Comment
<i>id</i>	integer	The unique encoding for the spatial literal.
<i>value</i>	varchar	The string representation of the spatial literal.
<i>strdfgeo</i>	geometry	The serialized object of the vector representation of a semi-linear point set in normal form.

Each tuple in the *SpatialLiteral* table has an *id* that is the unique encoding of the spatial literal. The string representation of the spatial geometry is stored in the *value* attribute. The attribute *strdfgeo* is a spatial column whose data type is the PostGIS-defined geometry type and is used to store the geometric object that is described by the spatial literal. The stored object is the vector representation of a semi-linear point set in normal form. In the case that the vertical partitioning scheme of Sesame is used, the same additions are done to it to facilitate the storage of stRDF data.

In the rest of this section we will describe in detail the conversions that take place during data loading and how we populate the table *SpatialLiteral*. Prior to storing geometries, we process the input geometries provided by the users so that each geometric object that we consequently store in PostGIS is in a *normal form* that satisfies the following requirements:

- Constraints have been transformed to the equivalent vector representation.
- The geometry is expressed as the union of convex components.
- There are neither redundant nor inconsistent geometries.
- The geometries are stored in a specific order to speed-up the conversion between the vector and constraint representation.



Let us now describe in detail how data are converted in normal form. When a user wants to store a spatial literal, the RDBMS layer of the Storage Manager initiates the procedure of converting the input geometry to normal form and storing it to PostGIS. When the input geometry is represented with constraints, the Storage Manager communicates with the Representation Translator to convert these constraints to disjunctive normal form (DNF) so that each geometry is expressed as a disjunction of conjunctions of constraints. Since each conjunction of constraints represents a convex spatial object, the DNF form is equivalent to a union of convex spatial objects. Subsequently, the Representation Translator converts the constraint representation of the geometry to the equivalent vector representation using Fukuda's `cddlib` library [10] that is an implementation of the Double Description Method of Motzkin et al. [48] for generating all vertices and extreme rays of a general convex polyhedron in R^d given a system of linear inequalities. Afterwards, the Normal Form Constructor of the Constraint Engine expresses the geometry in PostGIS' Enhanced Well Known Binary (EWKB) format, a superset of OGC's WKB format. For optimization purposes we store the vertices of each spatial object in a predefined order, so that we can later compute the constraint representation of the geometry with a simple scan over the geometry's vertices. Finally, the normalized geometry is stored in the `strdfgeo` attribute of the `SpatialLiteral` table described previously in this section.

Our implementation also supports geometries expressed in the OGC Well-Known Text (WKT) specification². In this case, the RDBMS layer of the Storage Manager communicates with the Constraint Engine, which normalizes the input geometry, but a different procedure is followed. Specifically, the Convex Partitioner simplifies the user input so that non-simple polygons are converted to a list of simple polygons. Each non-convex simple polygon is partitioned into convex sub-partitions using CGAL [47]. The Convex Partitioner uses CGAL's implementation of the simple approximation algorithm of Hertel and Mehlhorn [20] that requires $O(n)$ time and space to construct a decomposition of the initial polygon into no more than four times the optimal number of convex pieces. Finally, the Normal Form Constructor of the Constraint Engine expresses the geometry in EWKB, just like in the previous case, and the normalized geometry is stored in the `strdfgeo` attribute of the `SpatialLiteral` table.

The implementation described above has been heavily influenced by the implementation of Dédale [43]. For example, we use the same normal form for storing semi-linear sets, and we also cater for the storage of non-convex geometries imported from data sources that use the vector model.

Let us now explain how the tuples containing temporal information are handled. For each occurrence of a time interval, Strabon stores two additional tuples, each one providing information for the upper and lower bounds of the temporal extent of the corresponding dataset. Let us illustrate this with the use of an example. Suppose we need to store the following triples:

```
Demo:ModelledTideHeightDataset
  a Services:Dataset;
  rdfs:label "Modelled tide height Dataset";
  time:hasTemporalExtent "[NOW,NOW+12]"^^strdf:TemporalInterval.
```

The last stRDF triple will be converted into

```
Demo:ModelledTideHeightDataset time:hasTemporalExtent _:t1.
_:t1 time:Starts "0"^^<http://www.w3.org/2001/XMLSchema#integer>.
_:t1 time:Ends "12"^^<http://www.w3.org/2001/XMLSchema#integer>.
```

In this case we assume the reference time is 00:00. After this conversion, the resulting triples would be stored in Strabon.

²Although stRDF/stSPARQL is based on linear constraints, it is our intention that Strabon enables the processing of spatial data represented in common OGC formats, supported by popular DBMS or GIS. In fact, this need has arisen in SemsorGrid4Env where we have to import geometries expressed in WKT that described the spatial coverage of stored and stream data sources.

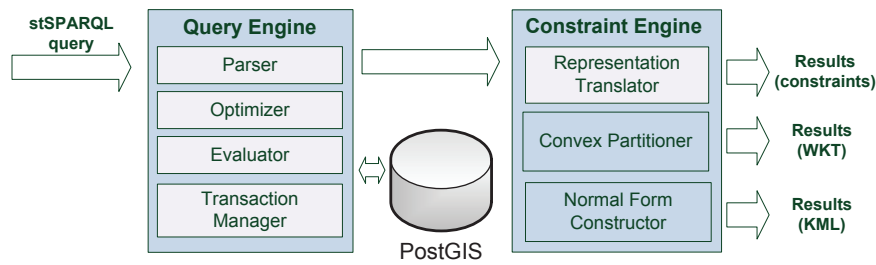


Figure 3.3: Query evaluation

3.3 Evaluating stSPARQL queries

Another important module of Strabon is the query engine. The design of the query engine is classical and is illustrated in Figure 3.3. It consists of a parser, an optimizer, a query processor (evaluator) and a transaction manager. The parser extends Sesame’s parser to parse stSPARQL queries.

The parser generates a query model³ that will be optimized later on by the optimizer. In the version of Strabon currently under development, the Sesame optimizer and transaction manager are used essentially as is. Later versions will deal with the new features of stSPARQL. Sesame’s optimizer consists of a set of heuristics that rearranges the order of the query’s triple patterns so that the query will be evaluated more efficiently. Sesame’s evaluator has been extended so that it takes as input the optimized model of the query and evaluates it in a streaming fashion by taking into account the new features introduced by stSPARQL.

An stSPARQL query is evaluated as follows: In an initial pre-processing step, we search for any occurrence of the temporal variables we mentioned in Section 2.3. Any occurrence we locate results in minor alterations in the query in order to utilize the conversion of “temporal triples” we described in Section 3.2.

For instance, the query

```

SELECT ?SERVICE
WHERE {
    ?SERVICE Services:hasDataset ?DATASET1.
    ?DATASET1 time:hasTemporalExtent ?TIME.
    FILTER(?TIME contains "[NOW,NOW+8]"^^strdf:TemporalInterval).
}
    
```

would be rewritten to

```

SELECT ?SERVICE
WHERE {
    ?SERVICE Services:hasDataset ?DATASET1.
    ?DATASET1 time:hasTemporalExtent ?TIME .
    ?TIME <http://strdf.di.uoa.gr/ontology#Starts> ?TIME.
    ?TIME <http://strdf.di.uoa.gr/ontology#Ends> ?TIME2.
    FILTER((?TIME1<="0"^^<http://www.w3.org/2001/XMLSchema#integer>
    && ?TIME2>="+8"^^<http://www.w3.org/2001/XMLSchema#integer>)) .
}
    
```

³“Query model” is the terminology of Sesame. Seasoned database researchers could understand this to be a query graph.



In the next step, in which most of the processing takes place, the stSPARQL query is transformed to an SQL query by the evaluator depending on the storage being used. For example, the spatial filters of the stSPARQL query are mapped to the equivalent built-in functions and operators of PostGIS. Afterwards, Sesame evaluates the SQL query via PostGIS. The results of this query are returned to the evaluator which performs some post-processing to the results. For example, it calculates the result of expressions that may exist in the SELECT clause of the stSPARQL query, such as the construction of new spatial terms, using the results that have been retrieved from PostGIS in the previous step. The last step of the query processing involves displaying the final results in the format specified by the user. In the default case, results are encoded according to the SPARQL Query Results XML Format recommendation⁴ and the constraint representation is used for spatial data. The user can also request that the spatial literals are encoded using OGC's Well Known Text representation. Finally, the user can also retrieve the results encoded in Keyhole Markup Language (KML)⁵ which is an OGC standard and is widely used in the mapping industry.

3.4 Summary

In this chapter we presented the implementation of Strabon, a Storage and Query Evaluation module for stRDF/stSPARQL.

⁴<http://www.w3.org/TR/rdf-sparql-XMLres/>

⁵<http://www.opengeospatial.org/standards/kml/>



4. The system Atlas2

Our group has been developing Atlas¹, a full-blown open source P2P system for the distributed processing of RDF(S) data stored on top of the Bamboo DHT² [42], since 2006 in the context of FP6 project OntoGrid. In SemSorGrid4Env, we continued the development of Atlas concentrating on the following new research themes:

- (i) We designed new methods for answering SPARQL queries involving RDFS ontologies. In OntoGrid, we had developed one bottom-up and one top-down method for answering SPARQL queries over RDFS ontologies [27]. In SemSorGrid4Env we developed one more bottom-up method based on magic sets [3] and carried out a detailed evaluation of all techniques in PlanetLab and a local cluster. These results will appear in a forthcoming journal paper [24].
- (ii) We selected the most efficient of the methods of (i) and integrated it with the conjunctive triple pattern query processing algorithm of Atlas which appears in [33, 23]. In this way, Atlas can answer SPARQL queries over RDF data but also RDFS ontologies. This new query evaluation algorithm of Atlas has been presented in Deliverable D3.2 (Chapter 4).
- (iii) We designed and implemented a query optimizer for Atlas which targets the minimization of the time required to answer a query and the network bandwidth consumed during query evaluation. The theoretical basis, implementation and evaluation of this optimizer have been presented originally in Deliverable D3.2 (Chapter 4) and more recently in a conference publication [26].

In this report, we introduce the extensions we have implemented in order to enable Atlas to store sRDF data and process sSPARQL queries. To do this successfully, we utilized the expertise acquired from the centralized implementation of Strabon. The resulting new version of Atlas is called *Atlas2*. In Atlas2, we chose to use geometries expressed in the OGC Well-Known Text (WKT) specification. This choice is reasonable given that the flood use case has data with geometries expressed in WKT. In addition, linear constraints in Strabon are internally transformed into geometries expressed in WKT and stored in PostGIS, so the storage methods of Strabon and Atlas2 are based on the same principles.

In this chapter, we first present the architecture of Atlas2 and then, we describe the algorithm used to store sRDF data and the query evaluation algorithm for sSPARQL.

4.1 System architecture

Atlas2, like Atlas, is 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., 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 a `LOOKUP(id_k)` operation which returns a pointer to the peer responsible for the identifier id_k . A simple interface of two methods is also provided in a DHT; `PUT(id_k, x)` and `GET(id_k)`. In Bamboo [42], 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 $O(\log N)$ hops, where N is the number of network peers.

¹<http://atlas.di.uoa.gr>

²<http://bamboo-dht.org/>

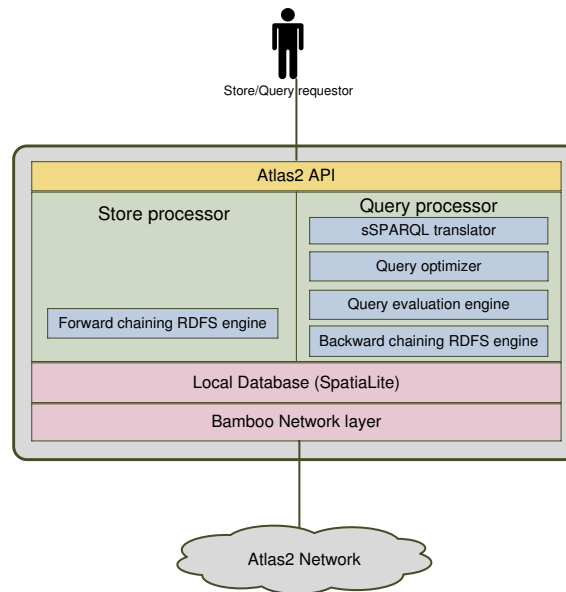


Figure 4.1: Atlas2 node architecture

A higher level view of each node's architecture as implemented in Atlas2 is shown in Figure 4.1. Any node can accept either a request for storing sRDF data in the network (STORE request) or a request for evaluating a sSPARQL query (QUERY request). We distinguish the following components of each Atlas2 node:

- the *Bamboo network layer*, which is responsible for routing and handling network messages,
- the *local database*, which is where the triples are stored locally at each node,
- the *store processor*, which is responsible for indexing and storing triples at the appropriate nodes in the network,
- the *SPARQL translator*, which is responsible for parsing SPARQL queries and transforming them to the equivalent internal representation handled by the *query processor*,
- the *query processor*, which is responsible for the distributed evaluation of sSPARQL queries,
- the *RDFS reasoning engine*, which is responsible for computing the appropriate inferences in order to give the answer to the queries taking into account the RDFS entailment rules, and
- the *query optimizer*, which is responsible for generating optimal plans to be evaluated distributed by the query processor.

4.2 Storing sRDF data

When a node wants to store sRDF data, it submits it in the form of an RDF document. The document can be in either RDF/XML or N-TRIPLE format. This document is decomposed into a collection of sRDF triples. sRDF triples have the form of (**subject**,**predicate**,**object**), where **subject** can be an IRI or a blank node, **predicate** is an IRI, and **object** can be an IRI, blank node, non-spatial literal or a spatial literal expressed in WKT.

Firstly, let us briefly explain how sRDF triples are stored in the network after a STORE request. As in Atlas, we use the triple indexing algorithm originally presented in [4] where each triple is



indexed in the DHT *three times*. The hash values of the subject, predicate and object of a triple are used to compute the identifiers that will indicate the nodes responsible for storing the triple. However, when a triple contains a spatial literal we only index the triple based on its subject and predicate³. 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 to obtain identifiers and send the corresponding triples to the peer that will store them using a single message. This storage scheme actually creates three indexes on a giant triples table which is distributed in the network. In our earlier work [33], 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 [36, 50, 18] where several indexes are maintained for faster retrieval. The use of these more exhaustive indexing schemes is the subject of future work.

In the latest version of Atlas presented in [26], we had adopted SQLite⁴ as the local database of each node. SQLite is a software library which can be integrated in an application program and used as a light-weight relational DBMS⁵. In order to be able to support spatial operations in Atlas2, we moved to SpatiaLite⁶, an extension of SQLite that enables us to support spatial data in a way conforming with OpenGIS specifications.

Triples are stored in a SpatiaLite database locally at each node. When a node joins the network for the first time, a database relation with three columns is built. The three columns correspond to the subject, predicate and object of the triples stored. When a triple arrives at a node, it is stored as a tuple in this relation. All objects are stored in the same column regardless of whether they are spatial literals or not.

As URIs and literals may consist of long strings, we have implemented a mapping dictionary similarly to Strabon and other centralized RDF stores [36, 6]. URIs and literals are mapped to integer identifiers and then, triple storage and query evaluation is performed using these identifiers. We do not use dictionary encoding for the spatial literals. Details on the implementation of the mapping dictionary on top of Bamboo were introduced in Deliverable D3.2 and [26].

4.3 Evaluating sSPARQL queries

In this section, we present the subset of sSPARQL that is supported by Atlas2 and describe the query evaluation algorithm.

4.3.1 Query model

Atlas2 supports the subset of sSPARQL which covers basic graph pattern queries with filter conditions which include *spatial operators*. Let us first define this subset of sSPARQL.

Definition 4. *Let U , L and V denote the pairwise disjoint sets of URIs, literals (spatial or non-spatial) and variables respectively. A triple pattern is a tuple (s, p, o) from $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$.*

The syntax of a sSPARQL query supported by Atlas2 is the following:

³To be able to index the triple on the DHT using an object which is a spatial literal would require the DHT to be based on a spatial data structure such as an R-Tree [17]. Unfortunately, this is not possible with Bamboo on which Atlas is based.

⁴<http://www.sqlite.org/>

⁵In previous versions of Atlas, the Berkeley DB database [21] was used which is included in the Bamboo implementation and used by Bamboo for various database tasks. However, we found that this implementation is inefficient and have moved to a lightweight relational database instead.

⁶<http://www.gaia-gis.it/spatialite/>



```

SELECT ?x1, ..., ?xk
WHERE {
  s1 p1 o1 .
  s2 p2 o2 .
  ...
  sn pn on .
  FILTER ((op11 op1 op12) && (op21 op2 op22) && ... && (opl1 opl opl2))
}

```

where $?x_1, \dots, ?x_k \in V$ are variables and (s_i, p_i, o_i) is a triple pattern as defined above. Variables $?x_1, \dots, ?x_k$ are called *answer variables* and each variable $?x_i$ appears in at least one triple pattern. For each constraint i ($1 \leq i \leq l$) in the filter condition, the first operand op_{i1} is a spatial variable that appears in at least one triple pattern, op_i is one of the spatial operators $\{\text{ANYINTERACT}, \text{DISJOINT}, \text{EQUALS}, \text{TOUCH}, \text{OVERLAP}, \text{COVERS}, \text{COVERED BY}, \text{CONTAINS}\}$ of the sSRARQL query language, and the second operand op_{i2} can be either a spatial literal in WKT format or a spatial variable.

Now let us present two examples of such queries.

Example 6. Spatial selection. *Find the URIs of the static sensors that overlap with the rectangle $R(0,0,100,100)$.*

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX
ssn: <http://www.w3.org/2005/Incubator/ssn/ontology#> SELECT
DISTINCT ?S WHERE {
  ?S rdf:type ssn:Sensor .
  ?G rdf:type ssn:SensorGrounding .
  ?L rdf:type ssn:Location .
  ?S ssn:supports ?G .
  ?G ssn:haslocation ?L .
  ?L ssn:hasGeometry ?GEO .
  FILTER(?GEO OVERLAP POLYGON((0 0, 0 100, 100 100, 100 0, 0 0))strdf:WKT)
}

```

Example 7. Spatial join. *Find the services of those GeoJSON Web services whose dataset geometries equals with the geometry of Solent.*

```

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX so:
<http://www.semsorgrid4env.eu/ontologies/ServiceOntology.owl#>
PREFIX coastDef:
<http://www.semsorgrid4env.eu/ontologies/CoastalDefences.owl#>
PREFIX addReg:
<http://www.ordnancesurvey.co.uk/ontology/v1/AdditionalRegions#>
SELECT DISTINCT ?ENDPOINT WHERE {
  ?S rdf:type so:WebService.
  ?S so:hasServiceType so:GeoJSON.
  ?S so:hasDataset ?D .
  ?D so:coversRegion ?R .
  ?R so:hasSpatialExtent ?G1.
  addReg:Solent so:hasSpatialExtent ?G2.
  FILTER(?G1 EQUALS ?G2).
}

```

4.3.2 Query evaluation algorithm

We have extended the query evaluation algorithm of Atlas, as it was presented in Deliverable D3.2 and in [26], to support filter conditions with spatial operators. Each triple pattern of a basic graph pattern query is evaluated at a (potentially) different node. Then, the results are joined with the intermediate results from previous triple patterns. To evaluate a triple pattern, a key from the

Algorithm 1: QC

```

1 event  $n$ .QEval( $id$ ,  $tp$ ,  $restTPs$ ,  $interRes$ ,  $vars$ ,  $constraints$ ,  $retIP$ )
2    $IR := MATCH(tp, constraints)$ ;
3   if  $interRes = \{\}$  then  $interRes' := IR$ ;
4   else  $interRes' := JOIN(IR, interRes, constraints)$ ;
5   if  $interRes' = \{\}$  then
6     sendto  $retIP$ .queryResp( $\{\}$ );
7     return;
8   end
9   if  $restTPs = \{\}$  then //all triple patterns have been evaluated
10    answer := project  $interRes'$  on  $vars$ ;
11    sendto  $retIP$ .queryResp(answer);
12    return;
13  end
14  project out unnecessary vars from  $interRes'$ ;
15   $tp' = restTPs.removeFirst()$ ;
16   $key' := FINDKEY(tp')$ ;
17   $id' := HASH(key')$ ;
18  sendto  $id'$ .QEval( $id'$ ,  $tp'$ ,  $restTPs$ ,  $interRes'$ ,  $vars$ ,  $retIP$ ,  $constraints$ );
19 end event

```

Figure 4.2: Query evaluation in Atlas2

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 nodes. We will call the node that is able to evaluate a triple pattern responsible node for the triple pattern. At each node, the intermediate results are inserted in a temporary relation which is joined with the bindings of the triple pattern’s variables which are retrieved from the matching triples stored at the local database. In case where a triple pattern is constrained by a filter with a spatial operator, the appropriate constraint is incorporated into the corresponding SQL query. The expertise acquired from the Strabon implementation was very crucial for developing this part of the algorithm.

The node that receives a query request chooses the triple pattern that should be evaluated first and sends a `QEval` message to the node that will start the query evaluation⁷. Figure 4.2 shows the pseudocode when such a message arrives at a node n . Keyword `event` is used for handling messages also indicating the peer where the handler is executed. Keyword `sendto` prefixed by an identifier declares that the message should be sent to the peer which is responsible for this identifier. In this case, a `LOOKUP` operation is performed first to discover the node responsible for this identifier and then the message is sent directly to this node.

First, node n evaluates the triple pattern tp using local function `MATCH` and forms a temporary relation IR by posing a selection query to its triple relation. If there are spatial filters that involve triple pattern tp , these constraints of these spatial filters are also taken into account in the corresponding SQL query. If relation $interRes$, which holds the intermediate results so far, is empty, node n is the first peer of the query evaluation and assigns IR to $interRes'$. Otherwise, n assigns to $interRes'$ the natural join of IR and $interRes$ using local function `JOIN`. In case list $constraints$ contains a expression with a spatial operator which enforces a spatial join between triple pattern tp and the intermediate results $interRes$, it performs a spatial join between relations IR and $interRes$ instead. If the result of the join is an empty relation, n returns an empty set to the node that posed the query (node with IP address $retIP$) and query evaluation terminates. Otherwise, query evaluation continues and node n picks the next triple pattern tp' from $restTPs$ to be evaluated. If $restTPs$ is empty, node n is the last node in the query evaluation, computes the projection of $interRes'$ on the answer variables $vars$ and sends the answer to the node with IP address $retIP$. Otherwise, query evaluation continues and n projects out from $interRes'$ variables that neither appear in $vars$ nor in the rest of the triple patterns. Then, n sends a new `QEval`

⁷In [26] we study optimization techniques in order to choose a good triple pattern ordering.



message to the next node based on the next triple pattern tp ⁸.

Let us now demonstrate the query evaluation algorithm using the query of Example 7 which contains a spatial join. Assume that we have the following sRDF triples which have been stored in the network according to our indexing scheme described in Section 4.2.

```

ex:ModelledTideHeightService  rdf:type so:WebService .
ex:ModelledWaveHeightService  rdf:type so:WebService .
ex:ModelledWaveHeightService  so:hasDataset
ex:ModelledWaveHeightDataset . ex:ModelledWaveHeightService
so:hasServiceType so:GeoJSON . ex:ModelledWaveHeightService
so:hasEndpointReference ex:endpoint1 . ex:ModelledWaveHeightDataset
so:coversRegion ex:SolentModelledArea . ex:SolentModelledArea
so:hasSpatialExtent
      POLYGON((0 0, 0 100, 100 100, 100 0, 0 0))^strdf:WKT .
addReg:Solent so:hasSpatialExtent
      POLYGON((0 0, 0 100, 100 100, 100 0, 0 0))^strdf:WKT .

```

The node that receives the query (let it be x) sends a `QEval` message to the node responsible for evaluating the first triple pattern (node n_1). The parameters of the message are the following:

- id is the hash value of the key `so:WebService`,
- tp is the triple pattern `(?S, rdf:type, so:WebService)`,
- $restTPs$ contains the triple patterns


```

(?S, so:hasServiceType, so:GeoJSON),
(?S, so:hasDataset ?D),
(?D, so:coversRegion, ?R),
(?R, so:hasSpatialExtent, ?G1),
(addReg:Solent, so:hasSpatialExtent ?G2),

```
- $interRes$ is an empty set,
- $vars$ contains variable `?S`,
- $constraints$ contains the constraint `(?G1 equals ?G2)`, and
- $retIP$ is the IP address of node x .

Node n_1 finds the bindings of triple pattern tp by posing a selection query to its local database relation and assigns them to relation lR :

```

S
-----
ex:ModelledTideHeightService ex:ModelledWaveHeightService

```

In this case, two triple patterns matched with the triple pattern tp . Since $interRes$ is empty, it also assigns $interRes'$ with the bindings of lR . Then, it composes a new `QEval` message with tp' the second triple pattern of the query `(?S, so:hasServiceType, so:GeoJSON)`, id' the hash value of key `so:GeoJSON` and sends it to the node responsible for key `so:GeoJSON` (node n_2). Node n_2 computes the intermediate results $interRes'$ by joining the previous intermediate results $interRes$ with the bindings found locally for the second triple pattern. The new intermediate results $interRes'$ are now equal to:

⁸We assume that each triple pattern has at least one bound component. The case where all three components of a triple pattern are variables requires a slightly different implementation which we do not discuss here.

5. The Semantic Registration and Discovery Service

In this chapter we present the Semantic Registry of the SensorGrid4Env infrastructure (also called the Semantic Registration and Discovery Service). We provide a brief description of SensorGrid4Env's Web service architecture. What is more, we mention the interfaces the Semantic Registry consists of, giving details on how either Strabon or Atlas2 can be used to implement the Semantic Registry of the SensorGrid4Env architecture. We also provide an example exposing the SensorGrid4Env software platform functionality.

5.1 The SensorGrid4Env Web service architecture

The SensorGrid4Env service-oriented architecture [12] shown in Figure 5.2 has three major classes of services that can be characterized as an *Application Tier*, a *Middleware Tier* and a *Data Tier* [13]. For the purposes of that project, a service-oriented architecture [12] was specified and formed the basis for its middleware. The main factor that led to this implementation choice is the type of the data sources we had to deal with. Specifically, both sensor networks and existing databases are of distributed nature.

A service-oriented architecture prescribes three different roles that a software component which is part of the architecture can play. A *service provider* exposes its functionality in the form of services and publishes the description of a service (i.e., *metadata* about the service) into a *service registry*. Then, a *service client* can lookup these descriptions in the service registry to discover the services that meet its requirements. Once an appropriate service is found, the client binds to the provider to consume the service. A high level view of this interaction pattern taken from [37, 8] is shown in Figure 5.1.

The Application Tier comprises services that provide domain specific functionality to consumers and applications. For example, an application may enable a user to visualize a layer on a map, displaying temperature or tide height in a selected area. The Middleware Tier consists of two main services: the *Semantic Registry* (also called the *Semantic Registration and Discovery service*), a service that can be accessed by prospective clients who want to manage thematic and spatial metadata about sensors, sensor networks, data sources, etc. and the *Semantic Integration service* that virtualizes multiple data sources with the use of semantic technologies [5]. The Data Tier comprises services that provide access to *streaming* data sources e.g., access to real-time sensor data, and *stored* data sources e.g., sensed data stored in a relational database. Let us now give more details about the Semantic Registry.

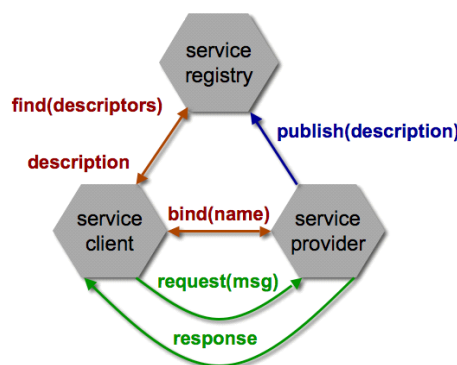


Figure 5.1: Service-oriented architecture

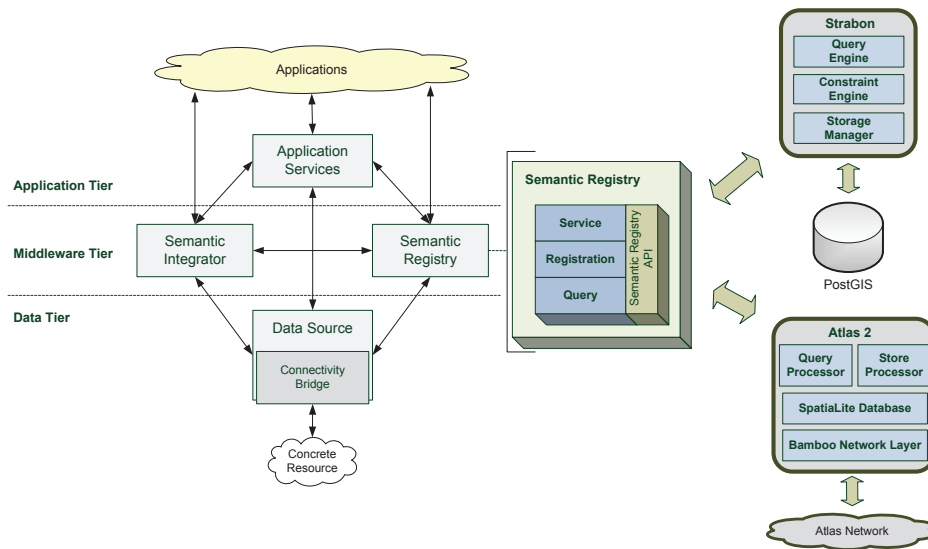


Figure 5.2: Implementing the Semantic Registration and Discovery Service using Strabon or Atlas2

The *Semantic Registration and Discovery Service* implements the *Semantic Registry* of the SensorGrid4Env project. It can be built on top of either the centralized (i.e., Strabon) or the distributed (i.e., Atlas2) implementation of stSPARQL. In SensorGrid4Env, these two implementations virtualize a *metadata store* and the means to populate, query and maintain metadata about available services.

5.2 Interfaces

The Semantic Registry enables the registration of Semantic Sensor Web resources, such as sensors, sensor networks, streaming data sources with real-time data and stored data sources with historical data. These resources can then be discovered using thematic and spatial criteria, and subsequently be used in other services or applications like mashups.

Figure 5.2 depicts how each one of our implementations can be used to realize the Semantic Registry of the SensorGrid4Env architecture. Strabon lies between clients i.e., metadata providers or metadata consumers, and PostGIS, that stores the metadata provided by metadata providers. Atlas2 follows the same principle, with the only difference being the fact that it uses SpatialLite as a back-end instead of PostGIS. Metadata providers intend to make public their resources by registering them with the registry, while metadata consumers aim to discover specific data resources based on criteria specific to them. The components of the SensorGrid4Env architecture play the roles of metadata providers or consumers. A data source is a metadata provider that may produce metadata about the stream or stored data exposed via its interfaces. The Semantic Integrator may be either a metadata consumer or a metadata provider. It may query the registry to discover data sources with specific characteristics, semantically integrate them in a new virtual data source and then store the description of the new virtual data source to the registry. Applications may also be metadata providers or metadata consumers. Applications may use the registry to discover data sources that meet their needs, or publish data sources that reside outside the context of SensorGrid4Env, such as OGC’s SWE services.

The list of interfaces exposed by the Semantic Registry that are mandatory for its operation are depicted in Figure 5.2. Specifically the service consists of the following interfaces:

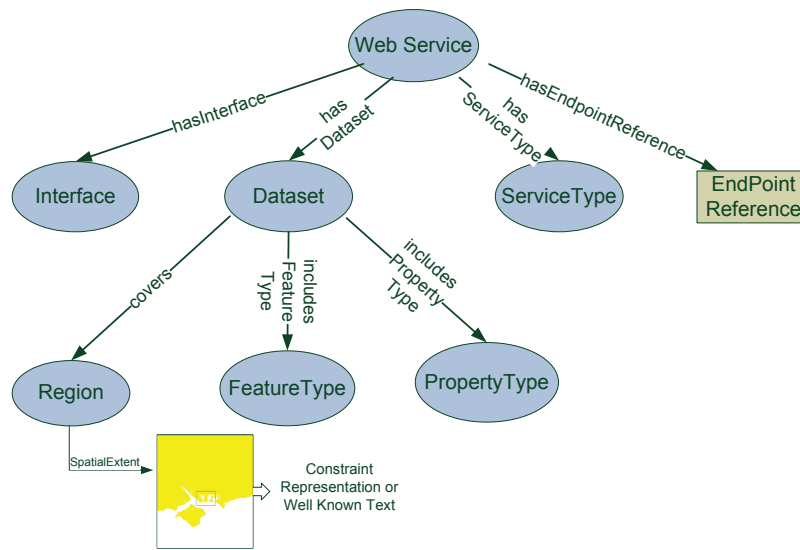


Figure 5.3: Service Ontology

- *Service Interface*. This interface enables clients to “make well-informed and well-formed interactions” [14] with the service. For example, a client may access the Service Interface to be informed about the interfaces that are offered by the registry and the declarative query languages supported by the registry.
- *Registration Interface*. This interface enables a metadata provider to register an RDF(S) document about SSW resources (e.g. sensors, sensor networks, data sources, etc.). If a metadata provider wants to store an stRDF description of a data resource, he must use an appropriate message from the ones defined in the specification of *WS-DAI-RDF(S)-Query*¹. The *Web Services Data Access and Integration Interface* (WS-DAI)² is a flexible framework defined by the Global Grid Forum, that ensures that a user dealing with grid or Web applications does not become exposed to the complexity of underlying data storage mechanisms. WS-DAI-RDF(S)-Query is a realization of this interface, that is specialized for accessing resources containing RDF(S) data.
- *Query Interface*. The *WS-DAI-RDF(S)-Query* specification has been adopted for this interface as well. The Query Interface provides operations allowing a metadata consumer to query the contents of the registry in order to discover relevant resources using stSPARQL. The consumer’s query is encapsulated inside a `SPARQLExecute` message, which is defined in the WS-DAI-RDF(S) realization of WS-DAI.

5.3 Example orchestrations in the SemsorGrid4Env architecture

Let us now give examples of the SemsorGrid4Env software platform functionality, focusing on the interactions of various components with the Semantic Registry for registering and discovering data sources.

First and foremost, it should be mentioned that the property documents used for the description of the data resources are automatically “semantically enriched” by using a library developed for this purpose before being stored to the Semantic Registry. The semantic information the property document is enhanced with describes the content of the dataset.

¹<http://forge.ogf.org/sf/go/doc14074>

²<http://www.ogf.org/documents/GFD.74.pdf>



namespace	prefix
http://www.sensorgrid4env.eu/ontologies/ServiceOntology.owl	Services
http://www.ordnancesurvey.co.uk/ontology/v1/AdministrativeGeography	Administrative Geography
http://www.sensorgrid4env.eu/ontologies/CoastalDefences.owl	CoastalDefences
http://sweet.jpl.nasa.gov/2.0/timeExtent.owl	time
http://www.ordnancesurvey.co.uk/ontology/v1/AdditionalRegions	Additional Regions
http://sweet.jpl.nasa.gov/2.0/space.owl	space
http://www.sensorgrid4env.eu/ontologies/sampleInstances.rdf	Demo

Table 5.1: Namespaces used and corresponding prefixes

This information is modeled according to numerous ontologies that have been developed in the context of SensorGrid4Env to create semantically enriched property documents that contain thematic and spatial metadata about the exposed data sources. For example, it is possible to use stRDF in order to describe its geographical coverage, the features of interest it includes, etc. These ontologies include, among others, a sensor network and observation ontology, a domain-specific ontology describing the concept of flood and an ontology describing the content of a dataset exposed by a service (e.g., the dataset's spatial coverage, its structure, etc) [11]. A small part of the first of these ontologies is depicted in Fig. 5.3.

Table 5.1 also shows the prefixes used for the various namespaces that are included in our ontologies.

At this time we will make our example more concrete. Suppose that a user wants to expose **sensed data** of tide height. The property document he registers in Strabon (or alternatively in Atlas2) includes the following semantic information:

```

Demo:ModelledTideHeightService
  a Services:WebService;
  rdfs:label "Modelled tide height".
  Services:hasEndpointReference
    "http://www.channelcoast.org/services/spatial/wms?
    SERVICE=WMS&VERSION=1.1.1&
    REQUEST=GetMap&LAYERS=tide_height";
  Services:hasInterface Demo:ModelledTideHeightInterface;
  Services:hasServiceType Services:WMS;
  Services:hasDataset Demo:ModelledTideHeightDataset;

Demo:ModelledTideHeightInterface
  a Services:Interface;
  rdfs:label "Modelled tide height Interface".

Demo:ModelledTideHeightDataset
  a Services:Dataset;
  rdfs:label "Modelled tide height Dataset".
  time:hasTemporalExtent "[NOW,NOW+12]"^^strdf:TemporalInterval;
  Services:coversRegion AdditionalRegions:SolentModelledArea;
  Services:includesFeatureType CoastalDefences:PhysicalMetocean;
  Services:includesPropertyType CoastalDefences:TideHeight;

```



```
AdditionalRegions:SolentModelledArea
  a space:Region.
  Services:hasSpatialExtent
    "POLYGON((-3 49, -1 49, -1 57, -3 57, -3 49))"^^strdf:WKT;
```

The same user also wants to register **predicted values** of wave height in the area of the Solent using a Stored Data Service. The metadata included in this service's property document are:

```
Demo:ModelledWaveHeightService
  a Services:WebService;
  rdfs:label "Modelled wave height".
  Services:hasEndpointReference
    "http://www.channelcoast.org/services/spatial/wms?
    SERVICE=WMS&VERSION=1.1.1
    &REQUEST=GetMap&LAYERS=wave_height";
  Services:hasInterface Demo:ModelledWaveHeightInterface;
  Services:hasServiceType Services:WMS;
  Services:hasDataset Demo:ModelledWaveHeightDataset;

Demo:ModelledWaveHeightInterface
  a Services:Interface;
  rdfs:label "Modelled wave height Interface".

Demo:ModelledWaveHeightDataset
  a Services:Dataset;
  rdfs:label "Modelled wave height Dataset".
  time:hasTemporalExtent "[NOW,NOW+12]"^^strdf:TemporalInterval;
  Services:coversRegion AdditionalRegions:SolentModelledArea;
  Services:includesFeatureType CoastalDefences:PhysicalMetocean;
  Services:includesPropertyType CoastalDefences:WaveHeight;

AdditionalRegions:SolentModelledArea
  a space:Region.
  Services:hasSpatialExtent
    "POLYGON((-4 52, -2 52, -2 54, -4 54, -4 52))"^^strdf:WKT;
```

Another user wants to register a service providing various information about the flood defence infrastructures of England in general. The thematic, spatial and temporal metadata related to this service are the following:

```
Demo:NFCDDService Services:hasDataset Demo:NFCDDDdataset;
  a Services:WebService;
  rdfs:label "NFCDD".
  Services:hasEndpointReference
    "http://www.channelcoast.org/services/spatial/wms?
    SERVICE=WFS&VERSION=1.1.1
    &REQUEST=GetFeature&TYPENAME=nfcdd";
  Services:hasInterface Demo:NFCDDInterface;
  Services:hasServiceType Services:WFS;

Demo:NFCDDInterface
  a Services:Interface;
```



```
rdfs:label "NFCDD Interface".
```

```
Demo:NFCDDDataset
  a Services:Dataset;
  rdfs:label "NFCDD Dataset".
  Services:coversRegion AdministrativeGeography:england,
    AdministrativeGeography:wales;
  Services:includesFeatureType CoastalDefences:Defence;
  Services:includesPropertyType CoastalDefences:AssetCondition,
    CoastalDefences:AssetHeight;
  time:hasTemporalExtent "NOW";
```

```
AdministrativeGeography:england
  a space:Region.
  Services:hasSpatialExtent
    "POLYGON((-7.3 58, -2.6 59, 2.3 52, 0 50,
      -2 51, -3.6 54, -7 55, -7.3 58))"^^strdf:WKT;
```

```
AdministrativeGeography:wales
  a space:Region.
  Services:hasSpatialExtent
    "POLYGON((-5 53, -3.5 53, -2 51, -5 51, -5 53))"^^strdf:WKT;
```

These users have used the ontologies mentioned above to create the semantically enriched property documents. Afterwards, the users invoke the registration interface of the Semantic Registry to register the semantically enriched property documents.

Let us suppose now that an end-user is looking for a Web Feature Service (its endpoint actually) that observes assets' condition, and whose coverage intersects the coverage of a Web Map Service observing tide height and of another Web Map Service observing wave height. He may also want the area in which these services actually intersect to be returned in the results, in the format of Well-Known Text. The user may utilize a mashup application that transforms the user's request in an stSPARQL query with the appropriate thematic and spatial restrictions. A form of this stSPARQL query could be the following:

```
SELECT DISTINCT ?ENDPOINT toWKT(?GEO1 INTER ?GEO2 INTER ?GEO3)
AS Geometry WHERE {
  ?SERVICE1 rdf:type Services:WebService.
  ?SERVICE1 Services:hasEndpointReference ?ENDPOINT .
  ?SERVICE1 Services:hasServiceType Services:WFS.
  ?SERVICE1 Services:hasDataset ?DATASET1.
  ?DATASET1 Services:coversRegion ?REGION1.
  ?DATASET1 Services:includesPropertyType CoastalDefences:AssetCondition.
  ?REGION1 Services:hasSpatialExtent ?GEO1.
  ?SERVICE2 rdf:type Services:WebService.
  ?SERVICE2 Services:hasServiceType Services:WMS.
  ?SERVICE2 Services:hasDataset ?DATASET2.
  ?DATASET2 Services:includesPropertyType CoastalDefences:TideHeight.
  ?DATASET2 Services:coversRegion ?REGION2.
  ?REGION2 Services:hasSpatialExtent ?GEO2.
  ?SERVICE3 rdf:type Services:WebService.
  ?SERVICE3 Services:hasServiceType Services:WMS.
  ?SERVICE3 Services:hasDataset ?DATASET3.
  ?DATASET3 Services:includesPropertyType CoastalDefences:WaveHeight.
  ?DATASET3 Services:coversRegion ?REGION3.
  ?REGION3 Services:hasSpatialExtent ?GEO3.
```



```
FILTER(?GEO2 INTER ?GEO3 anyinteract ?GEO1)  
}
```

This query is transmitted through the Semantic Registry to Strabon (or alternatively to Atlas2) encapsulated inside a `SparqlExecute` request, where it is evaluated. The result of this query is returned to the mashup application and is presented to the user.

Similarly, a domain expert may be interested in exposing a source targeted at more specialized audiences. Specifically, she may want to associate the wave height with the height of the defence infrastructures in the area of the Solent, so that potential hazards (due to overflows) for the vulnerable defence structures may be discovered. In order to accommodate these specialized needs, she can use the Semantic Integrator to create a “virtual” data source that unifies the data source concerning wave height and the data source related to English defence infrastructures. Afterwards, the new data source is registered with the registry in the same way, and can be discovered in a way identical to the scenario above.

5.4 Summary

In this chapter we presented the Semantic Registration and Discovery Service. We mentioned the interfaces it comprises, and we also provided an example exposing the SemSorGrid4Env software platform functionality.



6. Source code

In this chapter we will describe the directory structure of the Strabon and Atlas2 prototypes, as well as the Semantic Registry. We will also briefly mention any external dependencies these three implementations have. We will then present the application programming interfaces that can be used in order to store stRDF files and evaluate stSPARQL queries in Strabon and Atlas2. Finally, we will mention which functions can be used in order to perform the above actions through the Semantic Registry.

6.1 Strabon

In this section, we describe the necessary source code details for Strabon.

6.1.1 Directory structure

The source code of Strabon is available for download from the SemsorGrid4Env SVN server¹. What we have made available for download is a version of Sesame enhanced with Strabon. Following its download, the inner directories of the downloaded folder will have the following structure:

- compliance/: Classes used for testing
- core/: Folder containing the java source files. Its inner directories are the following:
 - model: Contains interfaces and implementation for all basic RDF entities
 - query: Contains the implementation of the structures utilized during Query Processing
 - queryalgebra: Contains the implementation of the Query Evaluation module
 - queryparser: Contains the implementation of Sesame's parser, extended in order to be able to parse stSPARQL
 - queryresultio: Contains the implementation of various classes that deal with the query results' output form
 - repository: Contains the implementation of the Repository API, which is the central access point for Sesame repositories
 - rio: Contains the implementation of a set of parsers and writers for various RDF file formats
 - sail: Contains the implementation of the Storage And Inference Layer (Sail) API, which is a low level System API for RDF stores and inferencers
 - console: Contains the implementation of the command-line application used for interacting with Sesame
 - http: Contains the numerous Java classes implementing a protocol for accessing Sesame repositories over HTTP
 - src: Contains instructions for the build of the javadoc and the jar file distributable
 - runtime
- testsuites/: Folder containing the majority of the testing classes.
- pom.xml/: XML file that contains information about the project and configuration details used by Maven to build Strabon

¹<https://ssg4env.techideas.net/repos/Strabon>



6.1.2 External Dependencies

Strabon has dependencies on the following Java libraries, which are referenced in its POM file:

- PostgreSQL JDBC
- PostGIS
- Maven (along with many Maven plugins, e.g. surefire)
- GeoAPI 2.3
- GeoTools
- Java Topology Suite (JTS)
- JUnit
- Orbital
- Java Advanced Imaging API
- JAXB: Java Architecture for XML Binding
- Xerces
- Apache Commons
- SLF4J: Simple Logging Facade for Java

6.1.3 Application Programming Interface

The classes that are used to display Strabon's functionality are `StoreOp` and `QueryOp`. They both contain main functions that use methods located in `Strabon.java`.

- **StoreOp**: The main function in this class establishes a connection with a repository. Afterwards, it stores in the repository the triples contained in the file or url specified by the user. The function used for this purpose is `storeInRepo(Object src, String format, RepositoryConnectionWrapper wrapper)` and is implemented in `Strabon.java`. The first argument of this function is either a URL or a File. As for the formats that are supported for input stRDF files, they include N3 format (triples), NTRIPLES format and RDF/XML format. If the path provided is wrong, an exception is thrown. Otherwise, the stRDF file is stored in the underlying RDBMS.
- **QueryOp**: A connection with the repository is established by `QueryOp` as well. This class can be used in order to pose queries on previously stored data. As for the function used for this operation, it is implemented in `Strabon.java`, along with `storeInRepo`. Specifically, the function is `query(String queryString, String resultsFormat, SailRepositoryConnection con)`. In addition to the query's string representation, the user can specify the format in which he wants his results displayed. The supported formats are JSON, RDF/XML, DAWG and KML representation.

6.2 Atlas2

In this section, we describe the necessary source code details for Atlas2.



6.2.1 Directory structure

The source code of Atlas2 is available for download from the SVN server². After downloading Atlas2, the inner directories of the main folder will have the following structure:

- src/: the java source files
- test/: the junit tests
- lib/: the required libraries for the compilation of the project
- ext/: the required jar files
- bin/: the location of the generated class files
- build.xml: the build file to use with the Java-based build tool Apache Ant to compile and run Atlas2

6.2.2 Implementation details

Atlas2 is written using the Java programming language and therefore is platform independent. As we have already mentioned, Atlas2 is implemented on top of the Bamboo DHT. In this section, we will present implementation details for the Atlas2 system along with the structure of Bamboo from which we inherited the programming style. The programming architecture used in designing Bamboo is SEDA [51], which stands for the Staged, Event-Driven Architecture. When using SEDA, the functional units of a program are represented as separate stages. These stages communicate using Java objects called events. The main advantage of SEDA is that the interfaces between the different stages are only defined by the events they receive and send. The stages that initialize an Atlas2 node are specified in a configuration file written in XML. An example configuration file for running an Atlas2 node is the following:

```
<sandstorm>
  <global>
    <initargs>
      node_id localhost:5009
    </initargs>
  </global>
  <stages>
    <Network>
      class bamboo.network.Network
      <initargs> </initargs>
    </Network>
    <Router>
      class bamboo.router.Router
      <initargs>
        gateway count 1
        leaf_set_size 8
      </initargs>
    </Router>
    <Dht>
      class bamboo.dht.Dht
      <initargs>
        gateway count 1
        gateway 0 localhost:5009
      </initargs>
    </Dht>
    <AtlasStorageManager>
      class eu.ist.ontogrid.ontokit.Atlas.Stages.AtlasStorageManager
      <initargs>
        homedir /tmp/sm-blocks-0
      </initargs>
    </AtlasStorageManager>
  </stages>
</sandstorm>
```

²<https://ssg4env.techideas.net/repos/Atlas2>



```
</initargs>
</AtlasStorageManager>
<BulkStore>
  class eu.ist.ontogrid.ontokit.Atlas.Stages.BulkStore
  <initargs> </initargs>
</BulkStore>
<QueryExecutor> class eu.ist.ontogrid.ontokit.Atlas.Stages.Query
  <initargs>
    dictionary true
  </initargs>
</QueryExecutor>
<RDFS> class eu.ist.ontogrid.ontokit.Atlas.Stages.RDFS
  <initargs>backward_chaining true </initargs>
</RDFS>
<Optimizer> class eu.ist.ontogrid.ontokit.Atlas.Stages.Optimizer
  <initargs>
    optimization true
    type_of_optimization static
  </initargs>
</Optimizer>
```

In the following, we describe the main stages of Atlas2:

- **BulkStore:** This stage provides Atlas2 with the functionality of storing documents containing RDF statements. It accepts store requests and creates the equivalent put operations for storing this data in the network. The data is stored according to the storing protocol described in Section 4.2.
- **QueryExecutor:** This stage provides Atlas2 with the functionality of querying the RDF data stored in Atlas2 by submitting SPARQL queries. It accepts query requests and evaluates them by following the query protocol described in Section 4.3.
- **AtlasStorageManager:** This stage is responsible for communicating with the node's local database. A JDBC driver for SpatiaLite is created and every operation to the node's database is performed through this stage.
- **RDFS:** This state is responsible for carrying out every necessary action in order to enable RDFS reasoning. The reasoning process is performed in a backward chaining fashion.
- **Optimizer:** This stage is the query optimizer we have implemented in Atlas2 as we presented in Deliverable D3.2. The optimizer is responsible for creating query plans that will improve the system's performance in terms of query response time and bandwidth consumption.

6.2.3 Application Programming Interface

We have created an API for storing sRDF and RDF data in Atlas2 and posing sSPARQL and SPARQL queries to an Atlas2 node. The following functions are provided by the Atlas2 API:

- **store(urlorfile,lang):** stores a file in the Atlas2 network which is in a lang format. The file urlorfile should be given either by a URL or should be at a local path found from the Atlas2 node. The format of the file should be one of RDF/XML, NTRIPLE, N3, TURTLE. If the file is successfully stored in the network, a boolean value equal to true is returned to the user.
- **query(string):** poses an sSPARQL query to the Atlas2 network. If the query has not a correct syntax or is not supported by the system, an error message is returned. Otherwise, the answer to the query is returned to the user.



6.3 Semantic Registry

In this section, we describe the necessary source code details for the Semantic Registry.

6.3.1 Directory structure

The source code of the Semantic Registry is available for download from the SensorGrid4Env SVN server³. Following its download, the inner directories of the downloaded folder will have the following structure:

- `src/`: the java source files
- `build.xml`: the build file to use with the Java-based build tool Apache Ant to compile and run the Semantic Registry
- `target/`: the location of the generated class files
- `WebContent/`: the location of the elements of our project necessary to create a Web application
- `WEB-INF/`: the location of necessary configuration information for the Semantic Registry
- `wsdl/`: the location of the XML file describing the Semantic Registration and Discovery Service
- `pom.xml/`: XML file that contains information about the project and configuration details used by Maven to build the Semantic Registry

6.3.2 External Dependencies

The Semantic Registry has dependencies on the following Java libraries, which are referenced in its POM file:

- Strabon
- Apache CXF: A Web Services framework

6.3.3 Application Programming Interface

The classes that are used to display Strabon's functionality are `StoreOp` and `QueryOp`. They both contain main functions that use methods located in `Strabon.java`.

- `RegistrationPTImpl`: The role of this class is to implement the *Registration Interface* exposed by the Semantic Registration and Discovery Service.
- `SPARQLAccessPTImpl`: The role of this class is to implement the *Query Interface* exposed by the Semantic Registration and Discovery Service.
- `Register`: The role of this class is to invoke the *Registration Interface* exposed by the Semantic Registration and Discovery Service. It is used to enable a metadata provider to register an RDF(S) document about SSW resources (e.g. sensors, sensor networks, data sources, etc.).
- `Query`: The role of this class is to invoke the *Query Interface* exposed by the Semantic Registration and Discovery Service. It is used to query the contents of the registry in order to discover relevant resources using stSPARQL.

³<https://sbg4env.techideas.net/repos/Registry>



6.4 Summary

In this chapter we provided a brief description of the source code of our centralized and distributed implementations. We also gave a description of Semantic Registry's source code. We described the directory structure of each code package, listed each one's external dependencies, and enumerated the methods of their APIs.



7. Installation and execution

In this chapter we will provide instructions on how to compile Strabon and Atlas2, as well as how to install any programs they require. We will also include small tutorials on how to run Strabon and Atlas2, i.e., store stRDF files and evaluate stSPARQL queries. Finally, we will describe how to compile and deploy the Semantic Registry.

7.1 Strabon

In this section we give general directions on how to successfully run Strabon. Firstly, in order to run Strabon, you will need to install the relational database PostgreSQL, install the PostGIS spatial extension of PostgreSQL and create a spatially enabled database where Strabon will store stRDF files. Then you may download and build Strabon from source.

7.1.1 Installing PostgreSQL

In order to download the latest stable version of PostgreSQL the following steps must be followed:

- Download PostgreSQL 8.4.x from <http://www.postgresql.org/download/>
- Install PostgreSQL, specifying a password for the default user with administrative rights (postgres) and a port to which your server will listen (default is 5432)

7.1.2 Installing PostGIS

Afterwards, PostGIS needs to be installed, to add support for geographic objects to the PostgreSQL object-relational database. We are currently using PostGIS v1.5 in order to accommodate our needs. The stable version of PostGIS v1.5 is expected to be published in January, 2010. Until then, the following steps must be followed in order to install the development snapshot on a Windows environment:

- Download PostGIS 1.5 for PostgreSQL 8.4 from <http://postgis.refractions.net/download/>.
- Install PostGIS on top of your PostgreSQL installation
- An alternative to these two steps would be installing PostGIS from PostgreSQL's built-in Application Stack Builder

7.1.3 Creating a spatially enabled database

The next step is to create a "spatially enabled" database, where Strabon will store stRDF descriptions. In order to create this database the following steps must be followed:

- Run pgAdmin III (it was installed along with PostgreSQL)
- Connect to your database server
- From menu Edit, select "New Object → New Database"
- Name your database "sesame_store" and use "template_postgis" as its template



7.1.4 Compiling and running Strabon

In this section, we describe how to compile, use and run Strabon. After you have downloaded and installed PostgreSQL and PostGIS, you have to download Strabon from our repositories and compile it using `ant` and `maven`.

Checking out the code from SVN

The following steps need to be followed in order to download Strabon's source files:

- Install an `svn` client such as the one located here <http://www.open.collab.net/products/subversion/>
- Open a console
- Create a folder in which you want the source code to be stored, and change your current directory to this
- Type `svn co https://sensorsgrid.techideas.net/repos/Strabon/trunk`

Once you download Strabon from the SVN server, you should follow the following steps in order to compile and run Strabon:

- Download and install Apache ant from <http://ant.apache.org/>.
- Download and install Apache maven from <http://maven.apache.org/>.
- Edit your local Maven's configuration file (possibly `/.m2/settings.xml`) and add two servers with the id `ssg4e.internal` and `ssg4e.snapshot`. For example, your `/.m2/settings.xml` should be like the following:

```
<?xml version="1.0" encoding="UTF-8"?> <settings
xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <servers>
    <server>
      <id>ssg4env.internal</id>
      <username>your-techideas-username</username>
      <password>your-techideas-password</password>
    </server>
    <server>
      <id>ssg4env.snapshot</id>
      <username>your-techideas-username</username>
      <password>your-techideas-password</password>
    </server>
  </servers>
</settings>
```

- Change your directory to the `trunk/core` folder and type `ant package` in order to compile the project

The result of these actions is the creation of a zipped file named

`openrdf-sesame-2.2.4-sdk.zip` inside the `core/target` directory, containing our implementation. After unzipping this file and changing your directory to the `lib` folder it contained, you should follow the following steps in order to store an `stRDF` file and evaluate an `stSPARQL` query:



Storing an stRDF file

Once you have successfully built Strabon you can store an stRDF file by executing the following command:

```
java -cp $(for file in `ls -1 *.jar`; do myVar=$myVar./$file:""; done;
echo $myVar;) eu.ist.sensorgrid4env.strabon.StoreOp <HOST> <PORT>
<DATABASE> <USERNAME> <PASSWORD> <STRDF_SRC> <FORMAT>
```

where

- <HOST>: is the postgres database host to connect to
- <PORT>: is the port to connect to on the database host
- <DATABASE>: is the spatially enabled postgres database that Strabon will use as a backend
- <USERNAME>: is the username to use when connecting to the database
- <PASSWORD>: is the password to use when connecting to the database
- <STRDF_SRC>: is the stRDF file or the URL to be stored
- <FORMAT>: is the format of the file/URL to be stored (available types: NTRIPLES, N3, RDFXML)

Evaluating an stSPARQL query

Once you have successfully built Strabon you can evaluate an stSPARQL query by executing the following command:

```
java -cp $(for file in `ls -1 *.jar`; do myVar=$myVar./$file:""; done;
echo $myVar;) eu.ist.sensorgrid4env.strabon.QueryOp <HOST> <PORT>
<DATABASE> <USERNAME> <PASSWORD> <STSPARQL_QUERY> <FORMAT>
```

where

- <HOST>: is the postgres database host to connect to
- <PORT>: is the port to connect to on the database host
- <DATABASE>: is the spatially enabled postgres database that Strabon will use as a backend
- <USERNAME>: is the username to use when connecting to the database
- <PASSWORD>: is the password to use when connecting to the database
- <STSPARQL_QUERY>: is the stSPARQL query to evaluate
- [<FORMAT>]: is the format of your results (JSON/DAWG/XML)

7.2 Atlas2

In this section, we describe how to compile, use and run Atlas2.



7.2.1 Compiling and running Atlas2

Atlas2 can run in both Windows and Linux. In this section, we describe how to run Atlas2 either using Apache Ant¹ or using Eclipse IDE². If you are working in Windows, you should copy all `ddl` files found under `lib/win` directory into `C:/WINDOWS/system32`. If you are working in Linux, you should set the variable `LD_LIBRARY_PATH` to directory `lib/linux`.

Instructions to run Atlas2 using Apache Ant

Here are the instructions to compile Atlas2, run some Atlas2 nodes, store an sRDF file and pose an sSPARQL query.

Compiling Atlas2

1. Run `ant` to compile Atlas2.

Running Atlas2

1. Edit `build.xml` to include the correct arguments to the targets `run-atlas-node-*`. The arguments needed are five:
`<node_ip>` `<node_port>` `<gateway_ip>` `<gateway_port>` `<directory_path>`
where `<node_ip>` and `<node_port>` is the IP address and port number of the node that will join the network, `<gateway_ip>` and `<gateway_port>` is the IP and port number of a known node already in the network and `<directory_path>` is the directory where the local data are stored. The first time you run an Atlas2 node the node and gateway are the same.
2. Run `ant run-atlas-node-1` to start the first Atlas2 node.
3. Run `ant run-atlas-node-*` to start other Atlas2 nodes.

Storing an sRDF file

1. Edit `build.xml` to include the correct arguments to the target `Store`. The arguments needed are four:
`<node_ip>` `<node_port+1>` `<urlorfile>` `<lang>`
where `<node_ip>` `<node_port>` is the IP and port number of any node already in the network, `<urlorfile>` is the URL of the file to be stored, and `<lang>` is the format of the file. Predefined values for `lang` are `RDF/XML`, `N-TRIPLE`, `TURTLE` (or `TTL`) and `N3`. If the Atlas2 node runs locally you can also insert the local path of a file in `<urlorfile>`.
2. Run `ant Store` to store an sRDF file to the Atlas2 network.

Querying sSPARQL

1. Edit `build.xml` to include the correct arguments to the target `Query`. The arguments needed are three:
`<node_ip>` `<node_port+1>` `<query>`
where: `<node_ip>` `<node_port>` is the IP and port number of any node already in the network and `<query>` is the sSPARQL query
2. Run `ant Query` to pose an sSPARQL query.

¹<http://ant.apache.org/>

²<http://www.eclipse.org/>



Instructions to run Atlas2 using Eclipse

In this section, we describe how to compile, run and use Atlas2 using Eclipse.

Compiling Atlas2

If you have downloaded Atlas2 using Subclipse³ (the project that adds Subversion support to the Eclipse IDE) then the project should be automatically compiled successfully. In any other case, in order to compile Atlas2 using Eclipse the following steps should be followed:

1. Create a new Java Project.
2. Copy the `src` and `ext` directories into the project.
3. Configure the Build Path by adding all the jar files that are found in the `ext` directory.

Running Atlas2

1. Create a new Java application with main class `eu.ist.ontogrid.ontokit.Atlas.Atlas` and add the following arguments:
`<node_ip> <node_port> <gateway_ip> <gateway_port> <directory_path>`
where `<node_ip>` and `<node_port>` is the IP address and port number of the node that will join the network, `<gateway_ip>` and `<gateway_port>` is the IP and port number of a known node already in the network and `<directory_path>` is the directory where the local data are stored. The first time you run an Atlas2 node and gateway are the same.
2. Repeat step 1 for as many nodes you want to run.

Storing an sRDF file

1. Create a new Java Application with main class `eu.ist.ontogrid.ontokit.Atlas.xmlrpcapi.BulkStore`
2. Add the following arguments: `<node_ip> <node_port+1> <urlorfile> <lang>`
where `<node_ip>` `<node_port>` is the IP and port number of any node already in the network, `<urlorfile>` is the URL of the file to be stored, and `<lang>` is the format of the file. Predefined values for `lang` are `RDF/XML`, `N-TRIPLE`, `TURTLE` (or `TTL`) and `N3`. If the Atlas2 node runs locally you can also insert the local path of a file in `<urlorfile>`.

Querying sSPARQL

1. Create a new Java Application with main class `eu.ist.ontogrid.ontokit.Atlas.xmlrpcapi.Query`
2. Add the following arguments: `<node_ip> <node_port+1> <query>`
where: `<node_ip>` `<node_port>` is the IP and port number of any node already in the network and `<query>` is the sSPARQL query.

³<http://subclipse.tigris.org/>



7.3 Semantic Registry

In this section we give general directions on how to successfully compile and deploy the Semantic Registry. A servlet container such as Apache Tomcat will be needed for the Semantic Registry to be deployed in.

7.3.1 Installing Apache Tomcat

In order to download and install the latest stable version of Apache Tomcat the following steps must be followed:

- Download Tomcat 6.x.x from <http://tomcat.apache.org/>
- Install Tomcat
- Add your credentials in the tomcat-users.xml file located in the folder *TOMCAT_HOME/conf*

7.3.2 Compiling and running the Semantic Registry

In this section, we describe how to compile and deploy the Semantic Registry. After you have downloaded Tomcat, you have to download the Semantic Registry's code from our repositories and compile it using maven.

Checking out the code from SVN

The following steps need to be followed in order to download the Semantic Registry's source files:

- Install an svn client such as the one located here <http://www.open.collab.net/products/subversion/>
- Open a console
- Create a folder in which you want the source code to be stored, and change your current directory to this
- Type `svn co https://semsorgrid.techideas.net/repos/Registry/trunk`

Once you download the Semantic Registry from the SVN server, you should follow the following steps in order to compile it:

- Download and install Apache maven from <http://maven.apache.org/>.
- Edit your local Maven's configuration file (possibly `/.m2/settings.xml`) and add two servers with the id `ssg4e.internal` and `ssg4e.snapshot`. For example, your `/.m2/settings.xml` should be like the following:



```
<?xml version="1.0" encoding="UTF-8"?> <settings
xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <servers>
    <server>
      <id>ssg4env.internal</id>
      <username>your-techideas-username</username>
      <password>your-techideas-password</password>
    </server>
    <server>
      <id>ssg4env.snapshot</id>
      <username>your-techideas-username</username>
      <password>your-techideas-password</password>
    </server>
  </servers>
</settings>
```

- Enter the `trunk/WebContent/WEB-INF` directory so that it contains the credentials needed for you to connect to your local PostGIS installation
- Return to the `trunk` directory and then type `mvn package` in order to compile the project. The result of these actions is the creation of a `.war` file inside the `target` directory.
- Place the `.war` file that was created inside the `webapps` folder of your Tomcat installations
- Start your Tomcat server
- Open a browser and go to `localhost:<port>/manager/html` where `port` is the one you specified during Tomcat's installation (the default port is 8080)
- Click on the Semantic Registry in order to view its Graphical User Interface

7.4 Summary

In this chapter we provided instructions on installing our two different implementations. What is more, we described the execution process of each one of the systems for the convenience of potential users. We also gave instructions on how to compile and deploy the Semantic Registry.

8. Testing Strategy

In this chapter we give a brief presentation of the progress we have made in testing our code, following the evaluation strategy specified by the project. Our effort has mainly been focused on unit testing. For these purposes, we utilized the JUnit unit testing framework. In the next sections, we will present details on the tests that were used to check the functionality of Strabon and Atlas2.

8.1 Testing in Strabon

As we have already mentioned in 3.1 Strabon is built on top of the well-known RDF store Sesame. Sesame's source code already included a number of unit tests ensuring its proper function. We added numerous tests of our own in order to check the proper function of our additions. Table 8.1 shows the unit test coverage for the packages on which we made the most extensive changes and additions. The `strabon` package includes the majority of the enhancements we made on Sesame. The `query` package includes the implementation of the structures utilized by Sesame during Query Processing. The `model` package contains the interfaces and implementation for all basic RDF entities. Finally the `sail` package contains the implementation of the SAIL API, a low level System API for RDF stores and inferencers.

Module	Coverage
strabon-structures	43%
query	27%
query-impl	21%
query-parser-sparql-ast	40%
query-algebra-evaluation	2%
query-algebra-evaluation-iterator	3%
query-algebra-evaluation-util	55%
query-resultio-text	87%
query-resultio-sparqlxml	75%
query-resultio-binary	73%
query-resultio-text	87%
model-datatypes	24%
model-impl	21%
model-util	32%
model-vocabulary	26%
sail-rdbms-schema	5%

Table 8.1: Coverage on unit test for primary modules

8.2 Testing in Atlas2

As we have already discussed in Section 6.2, Atlas2 is implemented on top of the Bamboo DHT from which it has inherited its programming style. Hence Atlas2 is written in Java using SEDA [51] (Staged Event-Driven Architecture). The main modules of Atlas2 are represented as separate stages connected by queues. These stages communicate using Java objects called events. We run JUnit tests only for testing the sSPARQL parser in Atlas2 and for testing the results of the queries from an Atlas client. These JUnit tests can be found under the `test/` directory.



8.3 Conclusions

In this section we mentioned the progress we have made on testing our two implementations, Strabon and Atlas2. We also provided some statistics displaying the amount of code we have successfully tested.



9. Conclusions

In this report, we discussed the details of the implementation of Strabon v0.8, as well as the details of a new version of Atlas, called Atlas2, that supports the model stRDF and the query language stSPARQL. We also discussed how the Semantic Registry of the SemsorGrid4Env infrastructure can be implemented on top of Strabon or Atlas2. We included a presentation of the stRDF data model and the language stSPARQL, mainly for completeness reasons. We also gave a short tutorial on how to compile and run both our centralized and distributed implementations Strabon and Atlas2. Finally, we gave a brief description of our testing strategy for our implementations. This report accompanies the code for our implementations which is available for download from the SemsorGrid4Env SVN server¹.

In the forthcoming deliverable named “Evaluation of the registry services” (Deliverable 3.4) we will present the results of the experimental evaluation of our two implementations presented in this deliverable.

¹<https://sensorsgrid.techideas.net/repos/>



Bibliography

- [1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
- [2] James F Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.
- [3] C. Beeri and R. Ramakrishnan. On the Power of Magic. In *PODS '87: Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 269–284, New York, NY, USA, 1987. ACM.
- [4] M. Cai, M. R. Frank, B. Yan, and R. M. MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 2(2):109–130, December 2004.
- [5] Jean-Paul Calbimonte, Oscar Corcho, and Alasdair J. G. Gray. Enabling Ontology-based Access to Streaming Data Sources. In *ISWC 2010 (to appear)*, 2010.
- [6] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st international conference on Very large data bases (VLDB)*, pages 1216–1227, 2005.
- [7] James Clifford, Curtis E. Dyreson, Tomás Isakowitz, Christian S. Jensen, and Richard T. Snodgrass. On the semantics of "now" in databases. *ACM Trans. Database Syst.*, 22(2):171–214, 1997.
- [8] The SemsorGrid4Env consortium. Semantic Sensor Grids for Rapid Application Development for Environmental Management (White Paper). Unpublished manuscript.
- [9] Zhan Cui, Anthony G. Cohn, and David A. Randell. Qualitative and Topological Relationships in Spatial Databases. In *Advances in Spatial Databases*, 1993.
- [10] Komei Fukuda. cddlib library. http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html.
- [11] Raul Garcia-Castro, Gabriel Rucabado-Rucabado, Chris Hill, Agustin Izquierdo, and Oscar Corcho. Sensor Network Ontology Suite. Deliverable D4.3v1, SemSorGrid4Env, 2009.
- [12] 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 Castro, Jesus E Gabaldon, and Juan Jose Aparicio. Semsorgrid4env architecture - phase ii. Deliverable D3.2 Version 1.0, SemSorGrid4Env, December 2010.
- [13] 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.
- [14] 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 Libana. SemSorGrid4Env Architecture Phase I. Deliverable D1.3v1, SemSorGrid4Env, 2009.
- [15] Stéphane Grumbach, Philippe Rigaux, and Luc Segoufin. The DEDALE System for Complex Spatial Queries. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data (SIGMOD '98)*, pages 213–224, New York, NY, USA, 1998. ACM.
- [16] Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, 19(2):207–218, 2007.



- [17] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of Annual Meeting SIGMOD'84*, 1984.
- [18] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *Third Latin American Web Congress*, Washington, DC, USA, 2005.
- [19] Cory Henson, Josh Pschorr, Amit Sheth, and Krishnaprasad Thirunarayan. SemSOS: Semantic Sensor Observation Service. In *CTS*, 2009.
- [20] Stefan Hertel and Kurt Mehlhorn. Fast triangulation of simple polygons. In *Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 207–218. Springer Berlin / Heidelberg, 1983.
- [21] BerkeleyDB: <http://sleepycat.com/>.
- [22] 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.
- [23] Z. Kaoudi, M. Koubarakis, K. Kyzirakos, M. Magiridou, I. Miliaraki, and A. Papadakis-Pesaresi. Publishing, Discovering and Updating Semantic Grid Resources using DHTs. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, Heraklion, Crete, Greece, June 2007.
- [24] Zoi Kaoudi and Manolis Koubarakis. Distributed RDFS Reasoning over Structured Overlay Networks. Journal in preparation.
- [25] Zoi Kaoudi, Manolis Koubarakis, Kostis Kyzirakos, Iris Miliaraki, Matoula Magiridou, and Antonios Papadakis-Pesaresi. Atlas: Storing, Updating and Querying RDF(S) Data on Top of DHTs. *Journal of Web Semantics*, 2010.
- [26] Zoi Kaoudi, Kostis Kyzirakos, and Manolis Koubarakis. SPARQL Query Optimization on Top of DHTs. In *Proceedings of the 9th International Conference on The Semantic Web (ISWC 2010)*, Shanghai, China, 2010.
- [27] Zoi Kaoudi, Iris Miliaraki, and Manolis Koubarakis. RDFS Reasoning and Query Answering on Top of DHTs. In *Proceedings of the 7th International Conference on The Semantic Web (ISWC 2008)*, Karlsruhe, Germany, 2008.
- [28] Manolis Koubarakis and Kostis Kyzirakos. Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In *Proceedings of the 7th Extended Semantic Web Conference, ESWC*, pages 425–439, 2010.
- [29] 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.
- [30] Kostis Kyzirakos, Zoi Kaoudi, Manolis Karpathiotakis, and Manolis Koubarakis. Distributed data structures and algorithms for a semantic sensor grid registry. Deliverable D3.2 Version 1.0, SemSorGrid4Env, August 2009.
- [31] Kostis Kyzirakos, Manolis Koubarakis, and Zoi Kaoudi. Data models and languages for registries in SemsorGrid4Env. Deliverable D3.1 Version 1.0, SemSorGrid4Env, March 2009.
- [32] Kostis Kyzirakos, Manolis Koubarakis, and Manos Karpathiotakis. Implementation and deployment of the registry services phase i. Deliverable D3.3v1, SemSorGrid4Env, 2009.
- [33] E. Liarou, S. Idreos, and M. 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.
- [34] Holger Neuhaus and Michael Compton. The Semantic Sensor Network Ontology: A Generic Language to Describe Sensor Assets. In *AGILE 2009 Pre-Conference Workshop Challenges in Geospatial Data Harmonisation*, 2009.



- [35] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *VLDB*, 1(1), 2008.
- [36] Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. In *Proceedings of 34th International Conference on Very Large Data Bases (VLDB 2008)*, volume 1, pages 647–659, Auckland, New Zealand, 2008.
- [37] M. P. Papazoglou. Service-oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003)*, pages 3–12, 2003.
- [38] Bijan Parsia and Uli Sattler. OWL 2 Web Ontology Language, Data Range Extension: Linear Equations. W3C Working Group Note, October 2009. , <http://www.w3.org/TR/2009/NOTE-owl2-dr-linear-20091027/>, last accessed February 20, 2010.
- [39] 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.
- [40] Peter Revesz, Rui Chen, Pradip Kanjamala, Yiming Li, Yuguo Liu, and Yonghui Wang. The MLPQ/GIS constraint database system. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 29(2), 2000.
- [41] Peter Z. Revesz. *Introduction to Constraint Databases*. Springer, 2002.
- [42] S. Rhea, D. Geels, T. Roscoe, and J. Kubiawicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference 2004*, 2004.
- [43] Philippe Rigaux, Michel Scholl, Luc Segoufin, and Stephane Grumbach. Building a constraint-based spatial database system: model, languages, and implementation. *Information Systems*, 28(6):563–595, 2003.
- [44] Semantic Sensor Grids for Rapid Application Development for Environmental Management (SensorGrid4Env). <http://www.sensorsgrid4env.eu>.
- [45] Sesame RDF framework. <http://www.openrdf.org/>.
- [46] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *VLDB*, 1(2), 2008.
- [47] Computational Geometry Algorithms Library CGAL. <http://www.cgal.org>.
- [48] G.L. Thompson T.S. Motzkin, H. Raïfa and R.M. Thrall. The double description method. In H.W. Kuhn and A.W. Tucker, editors, *Contributions to theory of games*, volume 2. Princeton University Press, 1953.
- [49] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *VLDB*, 1(1), 2008.
- [50] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of 34th International Conference on Very Large Data Bases (VLDB 2008)*, volume 1, pages 647–659, Auckland, New Zealand, 2008.
- [51] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.