TELEIOS

FP7-257662

Deliverable

D4.1

An implementation of a temporal and spatial extension of RDF and SPARQL on top of MonetDB - phase I

George Garbis, Konstantina Mpereta, Manos Karpathiotakis, Kostis Kyzirakos, Babis Nikolaou, Michael Sioutis, Stavros Vassos, Iris Miliaraki, Katerina Papadaki, Manolis Koubarakis

and

Consortium Members

February 29, 2012

Status: Final
Scheduled Delivery Date: 29 February, 2012

Executive Summary

The objectives of WP4 are to study query processing and optimization algorithms for the temporal/spatial extensions of RDF and SPARQL, called stRDF and stSPARQL respectively, that were proposed in WP2, and implement these algorithms on top of MonetDB. This deliverable is the first of a set of three deliverables that represent the WP4 contribution towards the design and implementation of a system offering the abilities to represent and query both definite and indefinite geospatial information. These three deliverables are as follows:

- D4.1 An implementation of a temporal and spatial extension of RDF and SPARQL on top of MonetDB - phase I (Prototype/Public, Month 18)
- D4.2 An implementation of a temporal and spatial extension of RDF and SPARQL on top of MonetDB - phase II (Prototype/Public, Month 30)
- D4.3 The evaluation of the developed implementation (Report/Public, Month 36)

In this report we present the implementation process followed for the development of the system Strabon. Strabon is a storage and query evaluation module for stRDF/stSPARQL, which is built on top of well-known RDF and relational DBMS technology for efficient data storage, indexing, optimization and query evaluation.

Document Information

Contract Number	FP7-257662	**Acronym**	TELEIOS
Full title	TELEIOS: Virtual Observatory Infrastructure for Earth Observation Data		
Project URL	`http://www.earthobservatory.eu/`		
EU Project Officer	Francesco Barbato		

Deliverable	**Number**	D4.1	**Name**	An implementation of a temporal and spatial extension of RDF and SPARQL on top of MonetDB - phase I			
Task	**Number**	T4.1	**Name**	Query processing and optimization for a temporal and spatial extension of RDF and SPARQL on top of MonetDB - phase I: definite geospatial information			
Work package	**Number**			WP4			
Date of delivery	**Contract**	M18 (Feb. 2012)		**Actual**	29 February 2012		
Status	Draft ☐ Final ☑						
Nature	Prototype ☑ Report ☐						
Distribution Type	Public ☑ Restricted ☐						
Responsible Partner	NKUA						
QA Partner	CWI						
Contact Person	Manolis Koubarakis						
	Email	koubarak@di.uoa.gr		**Phone**	+30 210 727 5213	**Fax**	+30 210 727 5214

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 FP7-257662. The Beneficiaries in this project are:

Partner	Acronym	Contact
National and Kapodistrian University of Athens Department of Informatics and Telecommunications	NKUA	Prof. Manolis Koubarakis National and Kapodistrian University of Athens Department of Informatics and Telecommunications Panepistimiopolis, Ilissia, GR-15784 Athens, Greece Email: (`koubarak@di.uoa.gr`) Tel: +30 210 7275213, Fax: +30 210 7275214
Fraunhofer Institute for Computer Graphic Research	Fraunhofer	MSc. Thorsten Reitz Fraunhofer Institute for Computer Graphic Research Fraunhofer Strasse 5, D-64283 Darmstadt, Germany Email: (`thorsten.reitz@igd.fraunhofer.de`) Tel: +49 6151 155 416, Fax: +49 6151 155 444
German Aerospace Center The Remote Sensing Technology Institute Photogrammetry and Image Analysis Department Image Analysis Team	DLR	Prof. Mihai Datcu German Aerospace Center The Remote Sensing Technology Institute Oberpfaffenhofen, D-82234 Wessling, Germany Email: (`mihai.datcu@dlr.de`) Tel: +49 8153 28 1388, Fax: +49 8153 28 1444
Stichting Centrum voor Wiskunde en Informatica Database Architecture Group	CWI	Prof. Martin Kersten Stichting Centrum voor Wiskunde en Informatica P.O. Box 94097, NL-1090 GB Amsterdam, Netherlands Email: (`martin.kersten@cwi.nl`) Tel: +31 20 5924066, Fax: +31 20 5924199
National Observatory of Athens Institute for Space Applications and Remote Sensing	NOA	Dr. Charis Kontoes National Observatory of Athens Institute for Space Applications and Remote Sensing Vas. Pavlou and I. Metaxa, GR 152 36 Athens, Greece Email: (`kontoes@space.noa.gr`) Tel: +30 210 8109186, Fax: +30 210 6138343
Advanced Computer Systems A.C.S S.p.A	ACS	Mr. Ugo Di Giammatteo Advanced Computer Systems A.C.S S.p.A Via Della Bufalotta 378, RM-00139 Rome, Italy Email: (`udig@acsys.it`) Tel: +39 06 87090944, Fax: +39 06 87201502

Contents

List of Figures

List of Tables

1. Introduction

The objectives of WP4 are to study query processing and optimization algorithms for the temporal/spatial extensions of RDF and SPARQL, called stRDF and stSPARQL respectively, that were proposed in WP2, and implement these algorithms on top of MonetDB.

In D2.1 we presented a new version of the data model sRDF and the query language sSPARQL proposed recently by our partner NKUA that extends the W3C standards RDF and SPARQL for representing and querying spatial data in the Semantic Web. In the new versions of sRDF and sSPARQL we opted for a more practical solution to the problem of representing geospatial data and used the OGC standards Well-known Text and GML instead of linear constraints. We also extended sRDF and sSPARQL to allow the representation and querying of qualitative spatial relations as they have traditionally been studied by the qualitative spatial reasoning community. In the proposed extension of sRDF, called $sRDF^i$, we used a new kind of literals to represent spatial regions about which the known information is incomplete or indefinite.

In D2.3 we extended the theoretical foundations of the data models stRDF and $stRDF^i$ and query language stSPARQL presented in Deliverable D2.1.

In this report we present the implementation process followed for the development of the system Strabon. Strabon is a storage and query evaluation module for stRDF/stSPARQL, which is built on top of well-known RDF and relational RDBMS technology for efficient data storage, indexing, optimization and query evaluation.

We initiated the development of Strabon in the context of FP7 project SemsorGrid4Env [ssg]. In this report we document the steps followed to port our implementation on top of MonetDB. Our motivation behind this port is to show how spatial and temporal filtering can also be improved by taking into account the distinctive characteristics of a column store. We also discuss the process we followed to enhance our system with additional functionality motivated by the use cases of TELEIOS. What is more, we present how Strabon can be used in a real world scenario. Finally, we give a short tutorial on how to compile and run Strabon. This report accompanies the code for our implementation which is available for download from the TELEIOS Mercurial repository[1]. In other words, the complete deliverable D4.1 is this report plus code.

The work presented in this deliverable deals only with definite geospatial information, i.e., geospatial objects for which exact geospatial co-ordinates are known. The model $stRDF^i$ that was presented in D2.3 deals with cases where the existing geospatial information in indefinite, incomplete or qualitative and what information we have available can be captured by constraints expressed in an appropriate first-order constraint language. Strabon will eventually be extended to support $stRDF^i$. This work will be reported in D4.2.

The experimental evaluation of the Strabon system will also start in the following months. The results will be presented in Deliverable 4.3 “The evaluation of the developed implementation”.

The organization of this report is as follows. In Chapter 2 we present the implementation of Strabon. In Chapter 3 we present how Strabon can be used in a real world scenario. In Chapter 4 we provide a short description of the Strabon code, and in Chapter 5 we present a brief tutorial on how to compile and run Strabon. Finally, in Chapter 6 we conclude this deliverable and discuss future work.

[1]`teleios-repo.acsys.it`

2. The System Strabon3

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 project TELEIOS. Strabon was created by our group in the context of the European FP7 project SemsorGrid4Env [ssg]. We provide a brief description of this initial version of Strabon that used PostGIS as a relational backend, as well as an extended description of the newest, 3rd version of our system, which has been ported on top of MonetDB and aims to take advantage of the scalability and performance promised by column store implementations. We also express in detail the enhancements in functionality that we performed on Strabon, being motivated from the requirements of the NOA use case. Last but not least, we describe the process followed in order to enable Strabon to support both PostGIS and MonetDB as a backend.

2.1 Strabon in SemsorGrid4Env

In the FP7 project SemsorGrid4Env, NKUA designed the data model stRDF that extends RDF with the ability to represent spatial and temporal data using linear constraints and the query language stSPARQL that extends SPARQL for querying stRDF data. NKUA also developed the system Strabon, a storage and query evaluation module for stRDF/stSPARQL. Strabon is built on top of the well-known RDF store Sesame[1] (version 2.2.4) and extends the components of Sesame to be able to manage thematic, spatial and temporal data that are stored in a PostgreSQL server that is “spatially enabled” with PostGIS. The Sesame architecture consists of the following layers (some of which are shown in Figure 2.1 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. The role of the SAIL API 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.

Figure 2.1 presents the system architecture of the initial version of Strabon described. Strabon 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 the Parser and Evaluator of Sesame to handle stSPARQL queries and use its Optimizer and Transaction Manager as is.
- *Storage Manager*: This module is responsible for storing data in the underlying PostGIS database. The *Repository*, *SAIL* and *RDBMS* layers are modules of Sesame that were described above. We have extended the *RDBMS* layer to be able to store spatial literals in PostGIS.
- *PostGIS*: Strabon uses PostGIS to store stRDF data and to perform part of the query evaluation process.
- *Constraint Engine*: This module is responsible for processing the part of the stSPARQL queries that deals with geometries and the conversions that take place during data loading. We created this module because we wanted to allow 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 was introduced to translate spatial

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

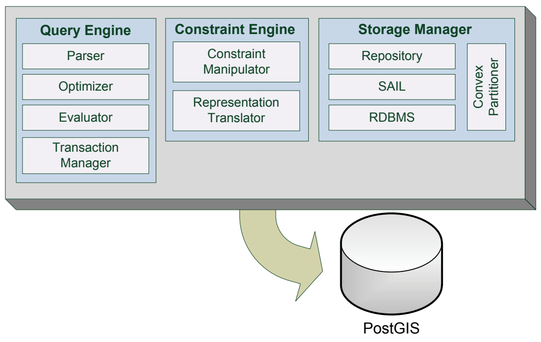

Figure 2.1: Strabon Architecture in the context of SemsorGrid4Env

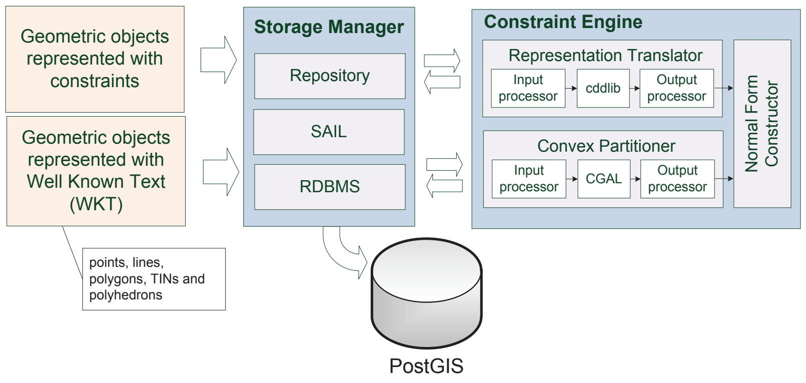
Geometric objects represented with constraints
Geometric objects represented with Well Known Text (WKT)
points, lines, polygons, TINs and polyhedrons
Storage Manager
Repository
SAIL
RDBMS
Constraint Engine
Representation Translator
Input processor
cddlib
Output processor
Convex Partitioner
Input processor
CGAL
Output processor
Normal Form Constructor
PostGIS

Figure 2.2: Storing a spatial geometry in the initial version of Strabon

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.

We now provide some additional details on the conversions that would take place during the storage of stRDF triples. 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 [Fuk] that is an implementation of the Double Description Method of Motzkin et al. [MRTT53] 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 Constuctor 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 PostGIS.

Our implementation also supports geometries expressed in the OGC Well-Known Text (WKT) specification. 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 [cga]. The Convex Partitioner uses CGAL's implementation of the simple approximation algorithm of Hertel and Mehlhorn [HM83] 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 PostGIS.

The implementation described above was heavily influenced by the implementation of Dédale [RSSG03]. For example, we used the same normal form for storing semi-linear sets, and we also catered for the storage of non-convex geometries imported from data sources that use the vector model.

2.2 The Architecture of Strabon3

Since the version of Strabon previously described was designed to implement the original version of the data model stRDF and the query language stSPARQL, we had made the implementation choice that linear constraints would be a first-class citizen in our system. Therefore, we had placed our efforts on achieving satisfactory performance primarily for the cases in which geospatial data were represented using constraints. After receiving feedback from the partners participating in SemsorGrid4Env and interacting with the EO community, we opted for a more practical solution to the problem of representing geospatial data and use the OGC standards Well-known Text and GML instead of linear constraints. Thus, we created a new version of Strabon fully utilizing widely-used standards for the representation of data. What is more, we decided to alter Strabon in order to allow it to become transparently integrated with newer versions of Sesame. To be more specific, the original version of Strabon was hard-coded on top of Sesame 2.2.4. On the contrary, the current version of Strabon is a Sesame SAIL layer that can be used in conjunction with newer versions of Sesame as well, thus taking advantage of newer features incorporated into Sesame (e.g., full SPARQL 1.1 support, bug fixes, etc.). Finally, Strabon is gradually evolving to fully utilize MonetDB as a relational backend. This port will enable us to take advantage of the scalability and performance promised by column store implementations.

This section presents an overview of the architecture of the current version of Strabon (Strabon 3). In [KKN+11] we discussed how the model RDF and the query language SPARQL were extended to represent and query spatial information using OGC standards. We implemented these extensions on top of Sesame, but in a different manner from the initial version of Strabon. Specifically, by exploiting the layered architecture and extensibility of Sesame, we added a transparent layer on top of Sesame extending it to handle spatial data, so that our implementation would easily integrate functionalities from new versions of Sesame, without intervening with the core implementation of

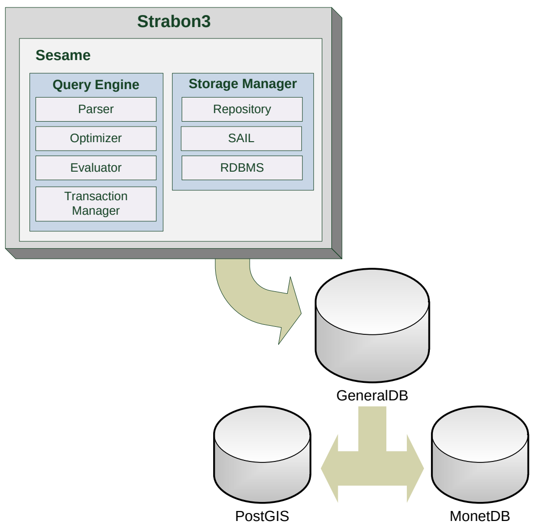

Figure 2.3: Strabon3 Architecture

Sesame. This feature gave us the opportunity to support additional functionality deriving from SPARQL 1.1, by the time recent versions of Sesame did so.

Both thematic and spatial data which are handled by this extension of Sesame are stored in a "spatially enabled" DBMS, i.e., a DBMS that offers storage, functions, and operators for spatial data. This version of Strabon has integrated two DBMS alternatives:

- **PostGIS**[2]**.** This module adds support for spatial objects and operations in PostgreSQL[3] to be used as a spatial database for geographic information systems (GIS).
- **MonetDB.** A spatial column-store DBMS which incorporates spatial capabilities through utilization of the GEOS[4] library.

The above backend alternatives are unified under an abstract layer, named GeneralDB, as figure 2.3 shows. We added GeneralDB to our storage stack so that any of the above DBMS could be used under a common interface (infrastructure) and the higher levels of Strabon could be built independently. Strabon follows the modular architecture of Sesame and comprises three modules:

Query Engine. The Query Engine of Strabon evaluates the stSPARQL query posed to the system. After a query is received, it is parsed and optimized. Then an execution plan is created and executed, and the results are returned to the client. The Evaluator and Optimizer of Sesame have been extended in order to handle stSPARQL queries. The parser and transaction manager were not modified, since the current version of stSPARQL is fully compatible with SPARQL 1.1.

Storage Manager. This module handles data storage in the GeneralDB RDBMS layer. We have extended the components of Sesame in a transparent fashion to be able to store the spatial literals we defined in Deliverable [KKN+11].

GeneralDB. Common infrastructure used for storing stRDF data and evaluating stSPARQL queries using either PostGIS or MonetDB.

[2]http://postgis.refractions.net/
[3]http://www.postgresql.org/
[4]http://trac.osgeo.org/geos/

2.2.1 Data Storage

This section presents the implementation details of the Data Storage module of Strabon. 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. Once the stRDF document is stored, it is then decomposed in triples and each triple is processed separately. Sesame adopts the *dictionary encoding* technique to store data; each RDF statement is encoded and identified by a unique integer, optimizing query evaluation. Sesame supports the following schemes for storing RDF data:

- **Monolithic**. According to this scheme, every triple is stored in a single triple table having the following schema:

  triple(subj_ id integer, pred_ id integer, obj_ id integer)

  The attributes *subj_ id, pred_ id,* and *obj_ id* are the unique encoding identifiers of the triple's subject, predicate, and object respectively.

- **Per-predicate**. Triples are stored in different tables based on their predicate. One table per predicate is maintained. For example, let us consider the following stRDF triple:

  subject predicate object

  The triple described above will be stored in a table with the following schema:

  predicate(subj_ id integer, obj_ id integer)

  The attributes *subj_ id* and *obj_ id* are the unique encoding identifiers of triple's subject and object respectively.

Under both those schemes, every RDF term is encoded by a unique positive integer to reduce space.

The mapping between the original RDF term and its encoding is stored in one of the predefined tables of Sesame. Each one of these tables is dedicated to storing mappings of a specific RDF resource. For example, a different table exists for the storage of URI values, another for blank nodes, etc. The combination of these tables forms a mapping dictionary used by Sesame.

Spatial literals are isolated and handled differently. Specifically, all spatial literals are also stored in an extra spatial table, named *geo_ values*, in a more practical format. Spatial literals are encoded using the same dictionary encoding techniques as well, and are also stored in the default scheme too (in the table containing the mappings of literals).

Schema of table *geo_ values*

Attribute	**Type**	**Comment**
id	integer	The unique encoding for the spatial literal.
strdfgeo	geometry	The spatial literal expressed in GeneralDB's geometry type

Each tuple in the *geo_ values* table has an *id* that is the unique encoding of the spatial literal based on the dictionary encoding process. The attribute *strdfgeo* is a spatial column whose data type is the GeneralDB-defined geometry type and is used to store the geometric object that is described by the spatial literal.

In Section 2.4.3, we present how we store geometries expressed in different CRS.

Due to its focus on atomic operations, Sesame lacks an efficient mechanism for bulk loading of large datasets, being inadequate for storing large RDF datasets. Thus, we have also implemented an stRDF loader which virtualizes a storing scheme identical to the one used by Sesame. This loader executes the following steps:

1. The loader creates the dictionary (including the mapping of every RDF term) in main memory.
2. If the monolithic storing scheme is followed, the loader flushes every row of the *triple* table to a single CSV file. In the case of the "per-predicate" scheme, a separate CSV file is created for every predicate and the loader flushes every predicate table to the respective CSV file.
3. The loader flushes the dictionary to CSV files.
4. The CSV files mentioned above are loaded and indexed into a PostGIS or MonetDB database.

2.2.2 Query Engine

The Query Engine of Strabon processes every query posed to the system. It comprises the following:

- **Parser**. The parser generates a query model that will be optimized later on.
- **Optimizer**. The Optimizer of Sesame consists of a set of heuristics that rearrange the order of the query's triple patterns so that the query will be evaluated more efficiently. We enhanced this optimizer with the ability to recognize potential spatial joins in the query plan. Prior to this modification, all joins would be performed based on thematic criteria only, and could lead to the presence of cartesian products in the final query model. We also extended the Evaluator of Sesame in order to enable it to receive the optimized model of the query and create a final query form to be evaluated by it that incorporates the extra features brought in by stSPARQL.
- **Transaction manager**. It is used as is currently.
- **Query Processor**. It executes the backbone of the query evaluation process.

The query evaluation process includes the following steps:

1. The parser generates a parse tree.
2. The parse tree is processed by several modules, building a query tree.
3. A query graph is constructed and traversed, taking into account both thematic and spatial joins, and creating a query tree with as few Cartesian products as possible.
4. The query's triple patterns are rearranged to optimize the query's evaluation, deviating from the default behavior of Sesame. This approach eliminates the Cartesian products in the final query plan taking into consideration the spatial operations involved in the query.

5. All stSPARQL extension functions present in the query are incorporated in the query tree. By default, Sesame performs the evaluation of extension functions after bindings for the variables present in the query have been retrieved. We altered this behavior for the following reasons:
   - Evaluating the whole query and then performing a post-processing step to eliminate all unwanted tuples introduces an overhead.
   - We wanted all spatial expressions present in the select clause of the query to be evaluated using the spatial functions of GeneralDB instead of relying on external libraries that would add unneeded post-processing cost.
6. The *geo_values* table is incorporated in the evaluation process. Before any spatial operator is mapped to SQL, it is declared that its arguments will be retrieved from *geo_values*. Values retrieved from this table will be joined thematically with the rest of the query via the id assigned to each spatial object during the dictionary encoding.
7. The query tree is passed to the Evaluator of Sesame, which has been enhanced to receive the modified query tree and create a final SQL query. The spatial operators of the original stSPARQL query are mapped to the equivalent built-in functions and operators of GeneralDB. The evaluator executes the produced SQL query in either PostGIS or MonetDB.
8. The results of this query are returned to the evaluator which performs any post-processing actions needed.
9. The final results are displayed. Strabon offers various available output formats depending on the user's needs. By default, spatial literals are returned in the form they were stored into and results are encoded using the SPARQL Query Results XML[5] Format recommendation. Another possible encoding is KML[6], a widely used standard in the mapping industry.

2.3 Migrating from PostgreSQL to MonetDB

MonetDB is a column-store DBMS that provides a modern and scalable solution without calling for substantial hardware investments. In this section, we present a minimal method for modifying MonetDB in order to replace PostgreSQL that is used by Strabon as a spatially-enabled relational backend.

For efficiency purposes, Strabon uses the Well-Known Binary format [OGC10b] to retrieve geometries from the underlying relational backend. MonetDB uses text for any kind of communication between the server and the clients, and the Well Known Text format is used for retrieving geometries. This approach caused a significant processing overhead since the geometries stored in MonetDB were exported in WKT, passed to Strabon using JDBC and then parsed in order to create geometric objects.

We created two functions that take a geometry as input and return a BLOB that contains the WKB representation of the geometry and vice versa. The SQL definition of these functions are:

```
CREATE FUNCTION AsBinary(g Geometry)
RETURNS BLOB external name geom."AsBinary";

CREATE FUNCTION GeomFromWKB(w BLOB)
RETURNS Geometry external name geom."GeomFromWKB";
```

[5] http://www.w3.org/TR/rdf-sparql-XMLres/
[6] http://www.opengeospatial.org/standards/kml

To perform the migration from PostgreSQL to MonetDB, we took into account the fact that Sesame defines an abstract layer that takes care of common SAIL tasks. Sesame provides an extension of this layer for using PostgreSQL as a relation backend. In the past, we modified this layer to make use of the spatial functions provided by PostGIS. We re-engineered this layer to avoid creating a separate implementation for each DBMS (PostGIS and MonetDB) and we defined an intermediate abstract layer, called GeneralDB. As a result, we extended the GeneralDB layer only for the parts that PostgreSQL and PostGIS differentiate over.

The main differences are related to the spatial part of the final SQL query that is posed to each DBMS. First, the spatial functions that are used had to change. Both systems have implemented functions complying to “OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option” OGC standard [OGC10a]. There are some minor differences in the naming or given arguments of spatial functions and some functions are not implemented in MonetDB, i.e., `ST_Cover(A:Geometry, B:Geometry): Boolean`. We implemented these functions by utilizing the `ST_Relate(A:Geometry, B:Geometry, intersectionPatternMatrix:String): Boolean` function that returns true if the geometric object `A` is “spatially related” to the geometric object `B` by testing for intersections between the interior, boundary and exterior of the two geometric objects as specified by the appropriate DE9-IM [CDFvO93] values in the `intersection Pattern Matrix` parameter. In addition, some other minor syntactic changes where needed. For example, MonetDB lacks of VACUUM and TRUNCATE commands. Thus, we had to deviate the behavior of Sesame not to call VACUUM. Another needed change came from the fact that the MonetDB JDBC demands the type of every dynamic variable to be clearly known at statement preparation. To tackle this issue, we added type casting for each dynamic parameter in order to be used in the SQL query posed in MonetDB.

We performed some preliminary experiments where we focus on the response time of queries with varying thematic and spatial selectivity. We generated datasets consisting of 10 and 100 million triples using the generator presented in [KKK+11]. This is a modified version of the generator used by the authors of [BNM10] which generates data according to a general version of the schema of Open Street Map presented in Figure 2.6. According to this schema, every feature is considered a node that has a spatial extent, e.g., the location of the node, and a number of tags describing the node. Each tag consists of a key - value pair of strings. By utilizing this schema we were able to adjust both the thematic and spatial selectivities of the queries used during evaluation. To be more precise, every generated node was tagged with key 1, every second node with key 2 etc, up to key 1024. As a result, if we selected all nodes tagged with key 1, we would select all available nodes, if we selected all nodes tagged with key 2, we would select half the nodes, etc. The locations of the generated nodes were placed uniformly on a 10.24 x 10.24 grid. More details on the methodology followed can be found in [KKK+11].

We instantiated the query template presented in [KKK+11] using appropriate values for managing the thematic and spatial selectivities. This resulted to queries like the following:

```
SELECT *
WHERE { ?tag geordf:key "1" .
        ?node geordf:hasTag ?tag ;
        strdf:hasGeography1 ?geo .
        FILTER (strdf:inside(?geo, "18.27 33.23,   18.2814146341463 33.23,
                18.2814146341463 33.2383317073171, 18.27 33.2383317073171,
                18.27 33.23"^^strdf:geometry))}
```

In Figures 2.4 and 2.5, we can see that PostgreSQL is very fast when a query selects a few spatial objects, whereas the performance of MonetDB is stable regardless of the number of spatial objects located inside the query region. This is due to the fact that MonetDB lacks support for spatial indexing. Traditional spatial databases evaluate spatial predicates in two steps: a *filtering* step and a *refinement* step. The *filtering* step selects those spatial objects whose minimum bounding

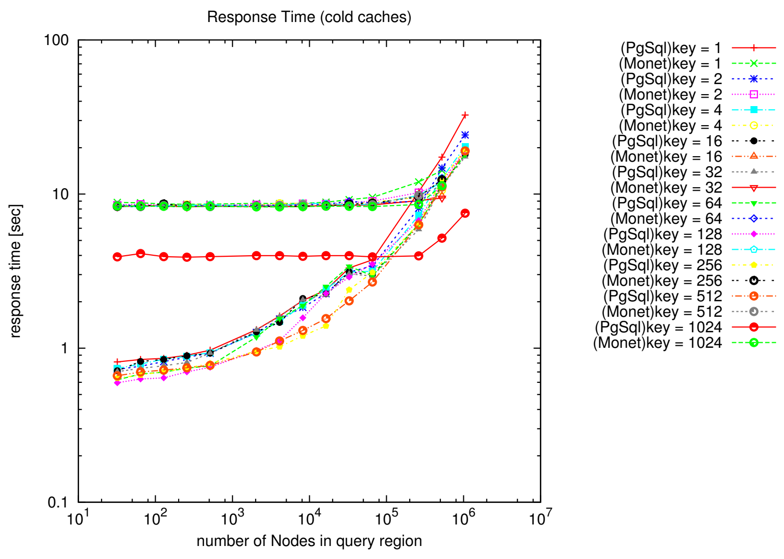

Figure 2.4: 10 million, cold caches

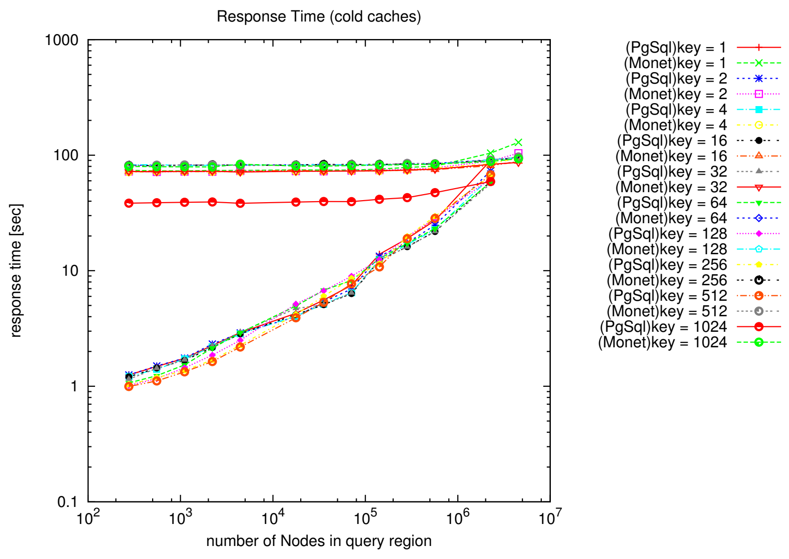

Figure 2.5: 100 million, cold caches

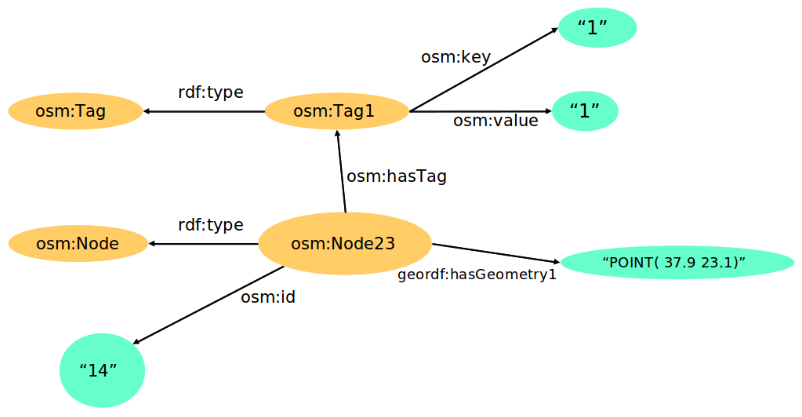

Figure 2.6: Primitive version of LGD schema

namespace	prefix
`http://www.earthobservatory.eu/ontologies/noaOntology.owl`	`noa`
`http://strdf.di.uoa.gr/ontology#`	`strdf`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#`	`rdf`

Table 2.1: Namespaces used and corresponding prefixes

box satisfy the spatial predicate. The result of the filtering step is a superset of the solution. In the *refinement* step, this superset is sequentially scanned and the spatial predicate is evaluated against the precise geometry of each spatial object. This two-step process allows the detection of spatial objects that will not satisfy the spatial predicate at a very low cost. PostGIS utilizes an R-tree-over-GiST spatial index for the filtering step, while it relies on the `GEOS` library for the refinement step. MonetDB relies entirely on the `GEOS` library for evaluating spatial predicates. This means that a full scan is performed on the `geo_values` table that contains all geometric objects and the spatial predicate is applied to all geometries. As a result, Strabon over MonetDB ran CPU-bound in all queries examined during this experiment.

Current work focuses on query processing and optimization for stRDF/stSPARQL on top of MonetDB. We are experimenting with the storage scheme that will be used for storing stRDF data in MonetDB. We are taking into account the distinctive column store architecture of MonetDB and experiment with storing multiple approximations of a geometry in separate columns since it does not incur additional load in the query evaluation process. We are also considering how to utilize techniques developed recently by CWI that continuously adapt the physical organization of the database as part of the query evaluation process in order to have a self-organizing implementation.

2.4 Enhancing Strabon Functionality

In this section we describe the process that we followed for enhancing Strabon with new functionality. This new functionality comprises support for updating RDF data, applying aggregation functions in spatial objects and storing spatial literals using various coordinate systems.

By examining in more detail the requirements set by the NOA Use Case and especially the requirements by TELEIOS UC2 SERVICEPROVIDER2 and SERVICEPROVIDER3 of [KPMm], the need to extend stSPARQL arose. Our aim was to express in stSPARQL logical rules that are used by NOA for refining the hotspot and burnt area products in a graceful way. This new functionality consists of support for updating RDF data, applying aggregation functions in spatial objects and storing spatial literals using various coordinate systems.

2.4.1 Updating stRDF Graphs

In this section we present some application examples that describe the motivation of adding this new update functionality. Afterwards, we present this new functionality and some implementation details about it.

Motivation

The necessity of adding update functionality to stSPARQL and subsequently Strabon is demonstrated hereafter with some examples. The prefixes and stRDF triples that are used in the following examples are shown in Table 2.1 and Example 1 respectively.

Example 1. *stRDF triples created using the NOA ontology*

```
noa:BA1 rdf:type noa:BurntArea ;
        noa:hasConfirmation noa:unknown;
        strdf:hasGeometry "POLYGON((20 20, 20 22, 22 22, 22 20, 20 20))"^^strdf:WKT .
noa:BA2 rdf:type noa:BurntArea ;
        noa:hasConfirmation noa:unknown;
        strdf:hasGeometry "POLYGON((23 23, 23 25, 25 25, 25 23, 23 23))"^^strdf:WKT .
noa:BA3 rdf:type noa:BurntArea .
        noa:hasConfirmation noa:unknown;
        strdf:hasGeometry "POLYGON((15 15, 15 17, 17 17, 17 15, 15 15))"^^strdf:WKT .

noa:H1 rdf:type noa:Hotspot;
        strdf:hasGeometry "POLYGON((19.5 19.5, 20 19.5, 20 20,
                          19.5 20, 19.5 19.5))"^^strdf:WKT.
noa:H2 rdf:type noa:Hotspot;
        strdf:hasGeometry "POLYGON((15 15, 15.5 15, 15.5 15.5,
                          15 15.5, 15 15))"^^strdf:WKT.

noa:UA1 rdf:type noa:UrbanArea;
        strdf:hasGeometry "POLYGON((21.5 18.5, 23.5 18.5, 23.5 21,
                          21.5 21, 21.5 18.5))"^^strdf:WKT.
noa:UA2 rdf:type noa:UrbanArea;
        strdf:hasGeometry "POLYGON((19 19, 21 19, 21 21, 19 21, 19 19))"^^strdf:WKT.
```

The above triples have been created using the ontology created for the NOA use case. They define three burnt areas, two hotspots and two urban areas. Each one corresponds to a specific area. Burnt areas additionally have a confirmation flag.

The NOA Use Case requirements included the need to update the data of an RDF graph. Prior to enhancing its functionality, Strabon offered only the following atomic update operations via an API:

1. Insert a statement.
2. Remove all statements that match a single triple pattern.

Let us now consider the stRDF graph in Example 1. This graph defines hotspots, burnt areas and urban areas where every hotspot and burnt or urban area is related to a geometry. Each burnt area also has a confirmation flag. A possible update query posed by NOA is **"Update the confirmation flag of a burnt area to confirmed if its geometry contains a hotspot"**. Without update functionality, the user would first pose a query (Example 2) to retrieve the identifiers of the burnt areas that need to be refined, and afterwards would delete the flags of the burnt areas retrieved in the previous step and insert the new ones by invoking the Strabon API mentioned above (Example 3).

Example 2. *Retrieve the URIs of the burnt areas that are not confirmed and have a geometry that contains the geometry of a hotspot.*

```
SELECT ?burntArea
WHERE {
  ?hotspot   rdf:type noa:Hotspot ;
```

```
           strdf:hasGeometry ?hGeo ;
  ?burntArea rdf:type noa:BurntArea ;
           strdf:hasGeometry ?bGeo ;
           noa:hasConfirmation ?bConfirmationFlag .
  FILTER ( strdf:contains(?bGeo, ?hGeo )) .
  FILTER ( ?bConfirmationFlag != noa:confirmed ) . }
```

This query retrieves the URIs of burnt areas that need to be refined. The refinement is demonstrated in Example 3.

Example 3. *Update confirmation flag of burnt areas retrieved from Example 2.*

```
for each burntArea
  Strabon.delete(BurntArea, noa:isConfirmed, null); //nulls can be seen as variables

for each burntArea
  Strabon.insert(burntArea, noa:isConfirmed, noa:confirmed);
```

This pseudocode deletes every triple whose subject is a given burnt area URI. A new triple, associating the previous URI with the property `noa:isConfirmed` *and the object* `noa:confirmed` *is then inserted. Note that null values in the delete function are handled as variables.*

Obviously, such a solution is not very flexible. Thus, we identified the need for Strabon to support a declarative update language for modifying RDF graphs. To do this, we relied on existing proposals for adding updates to SPARQL (SPARQL Update 1.1, currently a working draft of W3C) [GPP].

After adding support for SPARQL Update 1.1 to Strabon, the above update operation can be performed in a single step by posing the query in Example 4.

Example 4. *Update confirmation flag of every burnt area that is not confirmed and its geometry contains a hotspot.*

```
DELETE { ?burntArea noa:isConfirmed ?bConfirmationFlag }
INSERT { ?burntArea noa:isConfirmed noa:confirmed }
WHERE {
    ?hotspot   rdf:type noa:Hotspot ;
               strdf:hasGeometry ?hGeo .
    ?burntArea rdf:type noa:BurntArea ;
               strdf:hasGeometry ?bGeo ;
               noa:confirmationFlag ?bConfirmationFlag .
    FILTER ( strdf:Contains(?bGeo, ?hGeo) ) .
    FILTER ( ?bConfirmationFlag!= noa:confirmed) . }
```

This query makes use of the update extensions that took place in stSPARQL and conducts the refinement operation that is described above.

Updates in stSPARQL

The W3C SPARQL Working Group has started a discussion about updating RDF graphs, using SPARQL, since 2009. This discussion has led to the latest W3C Working Draft SPARQL 1.1 Update [GPP] that describes SPARQL-Update. SPARQL-Update is an update language for RDF graphs that uses a syntax derived from SPARQL. Sesame has supported SPARQL-Update since version 2.5, while Strabon is currently based on Sesame 2.6.3. Since Sesame already supports RDF updates, we only had to extend this functionality to handle geometry values in a special way.

The main facilities that are provided by SPARQL Update are :

1. Insertion of new triples into an RDF graph.
2. Deletion of triples from an RDF graph.
3. Performing a group of update operations as a single action.
4. Creation of a new RDF graph in a Graph Store.
5. Deletion of an RDF graph from a Graph Store.

In this document we focus only on performing a group of update operations as a single action, which is the most important and complicated. This is conducted with the MODIFY operation of SPARQL Update. According to [GPP] the syntax of this operation is the following:

```
# UPDATE outline syntax : general form:
MODIFY [ <uri> ]*
DELETE { template }
INSERT { template }
[ WHERE { pattern } ]
```

The intuitive semantics of this operation are the following:

1. Evaluate the WHERE clause once.
2. Use results of WHERE clause to instantiate the DELETE template.
3. Delete resulting triples.
4. Use results of WHERE clause to instantiate the INSERT template.
5. Insert resulting triples.

In Strabon, the WHERE clause is evaluated with the same mechanism as a simple stSPARQL query, using the Query Engine, and an iterator of bindings is returned. Afterwards, for each binding the respective triples are derived according to the delete and insert templates. First, triples derived from the delete template are deleted by the Storage Manager. The Storage Manager slightly distinguishes its behavior in the case of spatial literals by executing specific SQL scripts for this case. Finally, triples derived from the insert template are inserted using exactly the same mechanism as in storing triples.

If the "monolithic" scheme is used, the respective tuple is deleted from the table `triples`. Otherwise, the Storage Manager locates the table that stores the triples containing the current triple's predicate and deletes the respective tuple. As far as `dictionary encoding` is concerned, the table `hash_values` keeps a hash of every stored resource or literal. When some triples are deleted the table `hash_values` is not edited but the number of deleted triples is stored in a counter. Each time the number of deleted triples gets greater than the size of `hash_values` then all tables that comprise the dictionary are scanned and values that are not still present in KB are deleted.

2.4.2 Aggregation in stRDF Graphs

In this section we present some application examples that describe the motivation of extending stSPARQL and Strabon with aggregate functionality. Afterwards, we present this new functionality and some implementation details about it.

Motivation

Another extension of stSPARQL that needed to be carried out to provide support for the operations that arose in the NOA use case is the addition of spatial aggregate functions. Based on the W3C SPARQL 1.1 recommendation, stSPARQL was extended with this type of functionality. The examples below defend the usefulness of such functions.

A refinement rule used by NOA that requires an aggregate function is **"Update burnt area products by keeping the part of their polygon that does not lie in urban areas"**.

In the previous version of Strabon, that did not support spatial aggregate functions, the user had to pose an stSPARQL query to discover the hotspots that overlap with an urban area and then perform an additional processing step to remove all the parts of the geometry of each burnt area that lie in an urban area. Then, the graph had to be updated as we showed in the previous examples.

The main functionality that is required by this rule is the ability to compute the union of a set of geometries. Note that this functionality already exists in spatially extended relational databases, such as PostGIS. While conforming to the SPARQL 1.1 proposal, which defines aggregate functions for simple datatypes (e.g., integers, strings), we extended stSPARQL to support aggregations on spatial objects. After these enhancements, the refinement described above can be performed in a single step by posing the query in Example 5.

Example 5. *Update the geometry of every burnt area so that only areas that do not lie in urban areas are kept.*

```
DELETE {?burntArea strdf:hasGeometry ?bGeo}
INSERT {?burntArea strdf:hasGeometry ?cleanBurntArea}
WHERE {
  SELECT ?burntArea ?bGeo strdf:Difference(?bGeo, strdf:Union(?uGeo)) as ?cleanBurntArea
  WHERE {
    ?burntArea   rdf:type noa:BurntArea ;
               strdf:hasGeometry ?bGeo .
    ?urbanArea rdf:type noa:UrbanArea ;
               strdf:hasGeometry ?uGeo .
    FILTER ( strdf:anyInteract(?uGeo,?bGeo)) }
  GROUP BY ?burntArea ?bGeo }
```

Spatial Aggregates in stSPARQL

Strabon is constantly extended with additional functionality. The priority that is assigned to each potential extension is directly connected to the requirements set by the two use cases of TELEIOS. In this case, a number of the queries that NOA intends to use for the refinement of their hotspot products require applying a spatial operator over a group of solutions. Thus, incorporating spatial aggregates in stSPARQL was a task of high priority.

The approach we followed in order to enhance stSPARQL, and subsequently Strabon, with spatial capabilities is fully compliant with the approach followed by the current working draft on SPARQL 1.1 regarding aggregates[7]. Therefore, the current version of stSPARQL supports both `GROUP BY` and `HAVING` expressions. If the `GROUP BY` clause is specified, the output is divided into groups that match on one or more values. If the `HAVING` clause is present, it eliminates groups that do not satisfy the given condition.

The traditional aggregate functions such as `AVG` do not utilize the spatial dimension of the stRDF literals, and can't be applied to geometric objects. Therefore, we defined the following spatial aggregate functions:

- strdf:Union(A:Geometry):Geometry, returns a geometric object that is the union of the set of input geometries.
- strdf:Extent(A:Geometry):Geometry, returns a geometric object that is the minimum bounding box of the set of input geometries.

These functions can be used in the `GROUP BY` and `HAVING` clauses of an stSPARQL query. What is more, similarly to the functions defined in [KKN+11], and according to the SPARQL 1.1 working draft, they can be used in the `SELECT` clause of the query.

The spatial aggregation capabilities of stSPARQL is demonstrated with a series of examples. Table 2.1 shows the prefixes used for the namespaces that are utilized in the stRDF triples of the following examples.

Example 6. *stRDF triples created using the NOA ontology.*

```
noa:BA1 rdf:type noa:BurntArea ;
        strdf:hasGeometry "POLYGON((20 20, 20 22,
                          22 22, 22 20, 20 20))"^^strdf:WKT .
noa:BA2 rdf:type noa:BurntArea ;
        strdf:hasGeometry "POLYGON((23 18, 24 19,
                          23 19, 23 18))"^^strdf:WKT .
noa:BA3 rdf:type noa:BurntArea ;
        strdf:hasGeometry "POLYGON((20 15, 21 15,
                      21 16, 20 15))"^^strdf:WKT .

noa:UA1 rdf:type noa:UrbanArea;
        strdf:hasGeometry "POLYGON((21.5 18.5, 23.5 18.5,
                          23.5 21, 21.5 21, 21.5 18.5))"^^strdf:WKT.
noa:UA2 rdf:type noa:UrbanArea;
        strdf:hasGeometry "POLYGON((19 19, 21 19,
                          21 21, 19 21, 19 19))"^^strdf:WKT.
```

The above triples have been created using the ontology created for the NOA use case. They define three burnt area products and two urban areas. They also include an approximation of each area's geometry. Please note that according to our approach, if no URI is present to denote the coordinate reference system a geometry is encoded in, then by default it is assumed that the coordinates of said geometry are given in the World Geodetic System 1984 (WGS84). A visualization of these geometric objects is provided in Figure 2.7.

Example 7. *Retrieve the burnt areas that intersect with more than one urban area.*

[7]http://www.w3.org/TR/sparql11-query/#aggregates

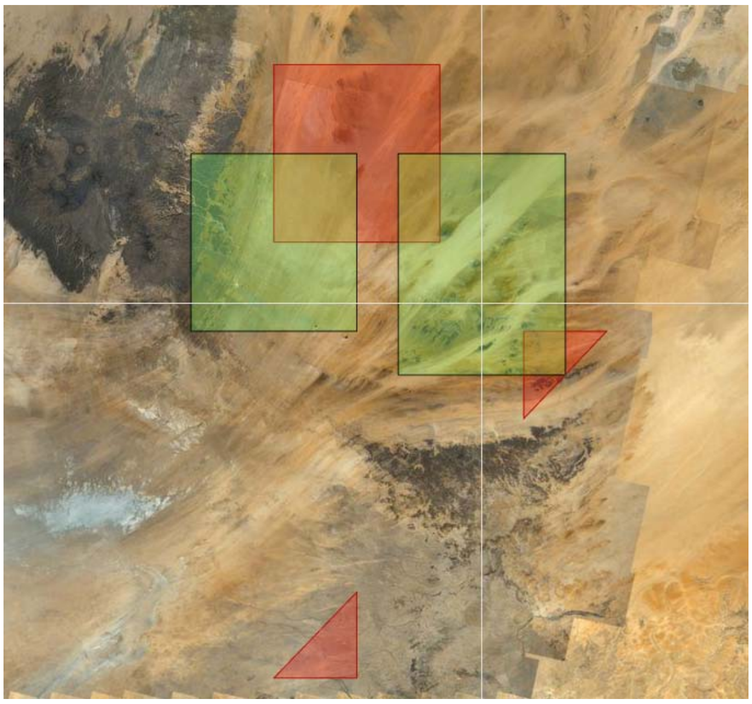

Figure 2.7: Burnt and Urban areas modeled according to the NOA ontology

```
SELECT ?burntArea
WHERE {  ?burntArea rdf:type noa:BurntArea;
    strdf:hasGeometry ?baGeo.
 ?urbanArea rdf:type noa:UrbanArea;
    strdf:hasGeometry ?uaGeo.
 FILTER(strdf:anyInteract(?baGeo,?uaGeo)) }
GROUP BY ?burntArea
HAVING (COUNT(?uaGeo) > 1 )
```

The result of the query is displayed below:

?burntArea
noa:BA1

The query above tests whether a burnt area intersects with an urban one. If this is the case, groupings are made depending on the burnt area, and any burnt area that does not intersect multiple urban areas is filtered out. As we can see, in this query only one of the three burnt areas available in the dataset is returned. The other two have been filtered out since they either intersect only one urban area, or are disjoint with each urban area.

Example 8. *Retrieve the parts of fire mapping products that do not lie in urban areas.*

```
SELECT ?burntArea
(strdf:difference(?baGeo, strdf:union(?uaGeo)) AS ?ruralPiece)
WHERE
{ ?burntArea rdf:type noa:BurntArea;
     strdf:hasGeometry ?baGeo.
```

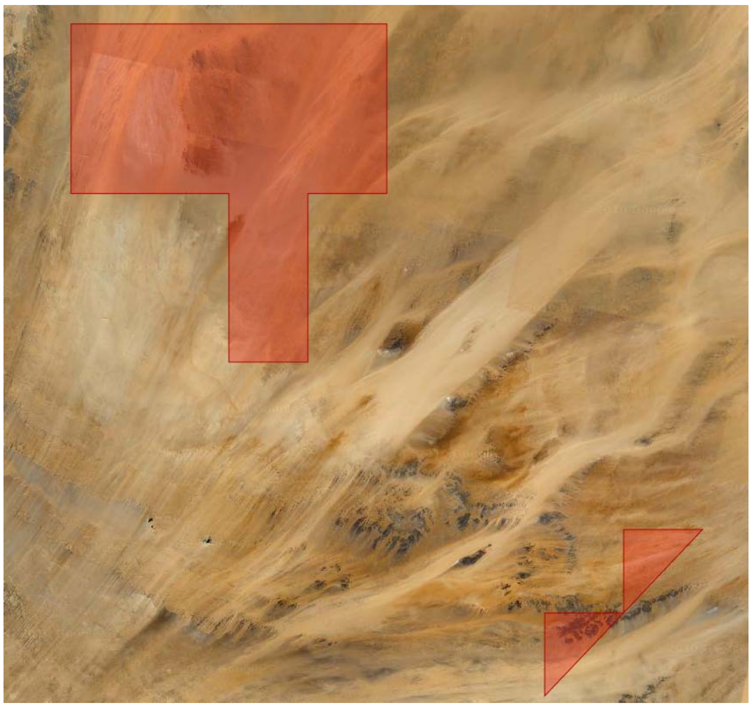
Figure 2.8: Result of query from Example 8

```
  ?urbanArea rdf:type noa:UrbanArea;
     strdf:hasGeometry ?uaGeo.
  FILTER(strdf:anyInteract(?baGeo,?uaGeo))
}
GROUP BY ?burntArea
```

The result of the query is displayed below:

?burntArea	?ruralPiece
noa:BA1	POLYGON ((20 21, 20 22, 22 22, 22 21, 21.5 21, 21.5 20, 21 20, 21 21, 20 21))
noa:BA2	MULTIPOLYGON (((23.5 18.5, 23 18, 23 18.5, 23.5 18.5)), ((23.5 19, 24 19, 23.5 18.5, 23.5 19)))

Queries similar to the one above provided the main motivation for this functionality extension in stSPARQL. The query above tests whether a burnt area intersects with an urban one. If this is the case, groupings are made depending on said burnt area. The geometries of the urban areas corresponding to each burnt product are unified, and their difference from the original burnt product is calculated and returned to the user. A visualization of these geometric objects that were computed on the fly can be seen in Figure 2.8.

Example 9. *Retrieve the fire mapping products that intersect with urban areas that are spread among an area that is over 8 square degrees.*

```
SELECT ?burntArea
WHERE { ?burntArea rdf:type noa:BurntArea;
```

```
                  strdf:hasGeometry ?baGeo.
        ?urbanArea rdf:type noa:UrbanArea;
                  strdf:hasGeometry ?uaGeo;
        FILTER(strdf:anyInteract(?baGeo,?uaGeo)). }
GROUP BY ?burntArea
HAVING (strdf:area(strdf:extent(?uaGeo)) > 8)
```

The result of the query is displayed below:

?burntArea
noa:BA1

The `strdf:Extent` *function is not as computationally expensive as* `strdf:Union`*, and is therefore used in cases where exact precision in results is not of major importance. The query above tests whether a burnt area intersects with an urban one. If this is the case, groupings are made depending on said burnt area. The burnt areas that have not affected "major" urban areas are filtered out of the final result.*

2.4.3 Coordinate Reference Systems in stRDF Graphs

In this section we present our motivation towards enhancing Strabon with the ability to handle geometric objects expressed in a variety of coordinate reference systems. We also provide information regarding the implementation choices we made to implement this feature in Strabon.

Motivation

Different organizations tend to expose the geospatial data they produce in a variety of coordinate reference systems. For example, NOA publish their hotspot products using the Greek Geodetic Reference System. The Greek Geodetic Reference System (GGRS87) is a projection system commonly used in Greece. We wanted Strabon to permit the evaluation of queries involving spatial objects expressed in different coordinate reference systems. Such an example deriving from the NOA use case is the combination of the NOA hotspot products with road information obtained from LinkedGeoData, which is published using the World Geodetic System (WGS84). By incorporating this functionality, our system is compliant with the guidelines of the Linked Data initiative, since datasets that have been made available by different organizations can now be linked based on their spatial relationships.

Coordinate Reference Systems in Strabon

Strabon currently supports storing spatial literals using various coordinate systems by transforming them to a single CRS (e.g., WGS84) that is defined by the user during the configuration phase. Any geometric object to be stored is transformed to a CRS that is uniformly used. The identifier of the CRS in which the geometry was originally encoded in during data loading is stored as well.

Storing geometric objects using a single CRS has the advantage that any operation that involves geometries encoded in different CRS, can be performed in a single step. Otherwise, the geometries should be converted to a common CRS prior to the operation execution. This approach incurs an overhead during data loading and requires the geometries which are bound to variables that appear in the select clause of a query to be transformed to the original CRS before returning them to the user.

An alternative storing scheme would be to store in different tables the spatial literals that are encoded in different CRS. The schema of such tables is *geo_values_crs(id integer, value geometry)* where *crs* is a unique identifier for the CRS. This scheme can be seen as a "per-CRS" storing scheme for spatial literals. Storing spatial literals according to this scheme is faster since the geometries do not need to be transformed prior to storage, has less storage overhead since we do not have to store explicitly the *srid* for each geometric object and results in the creation of multiple small tables storing the spatial literals, instead of a single table that contains all spatial literals. According to this scheme, a spatial selection query that defines a rectangle expressed in WGS84 and requests all geometries contained by this rectangle, is evaluated directly in the *geo_values_wgs84* table, while the geometries of every other table should be converted in WGS84 prior to the evaluation of the spatial filter. The final result of this query consists of the union of the intermediate results produced by the evaluation of the spatial filter in each table storing spatial literals. Spatial joins are more complex since their evaluation would require evaluating the spatial predicate between all pairs of tables storing spatial literals.

We performed an experiment where we focus on the two aforementioned storing schemes for spatial literals. We generated a dataset that consists of 10 million triples using the generator presented in [KKK+11]. We modified the dataset so that the geometries are encoded in 2,3,4 or 5 different CRS and we stored them using the two storing schemes. We instantiated the query template presented in [KKK+11] using the value `"1"` for `PARAMETER_A` and appropriate values for `PARAMETER_B` so that 10, 10^2, 10^3, etc. spatial nodes are selected. As a result, queries like the following were used:

```
SELECT *
WHERE { ?tag geordf:key "1" .
        ?node geordf:hasTag ?tag ;
        strdf:hasGeography1 ?geo .
        FILTER (strdf:inside(?geo, "18.27 33.23,   18.2814146341463 33.23,
                18.2814146341463 33.2383317073171, 18.27 33.2383317073171,
                18.27 33.23"^^strdf:geometry))}
```

We measured the response times for the case of cold and warm caches, but we present only the case of cold caches since they exhibit similar behavior. The response time for each dataset and storing scheme is presented in Figure 2.9.

We notice that the response time for the scheme that uses a single table for storing the spatial literals is slightly modified depending on the number of CRS present in the dataset. This behavior is unaffected by the spatial selectivity of the query. This observation reveals that the time required for transforming the geometric objects back to their original CRS, is not significant compared to the total query run time. On the contrary, the response time for the alternate storing scheme is significantly influenced by the number of CRS present in the dataset. We notice that as we increase the number of CRS present in the dataset, the system's performance deteriorates consistently. This occurs because the time required for merging the filtered spatial geometries of each table substantially influences the response time, compared to the query run time. For these reasons, we chose to store geometric objects using a single CRS.

For functions that take as arguments spatial literals encoded in different CRS, we follow the guidelines provided by GeoSPARQL and perform all calculations in the spatial reference system of the first argument.

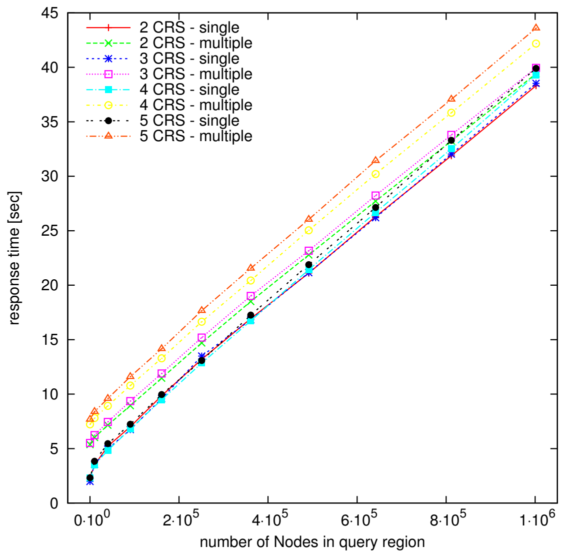

Figure 2.9: Response time for different storing schemes

2.5 Summary

In this chapter we presented the latest version of the system Strabon. First, we described the architecture of Strabon focusing on how data are stored in the backend RDBMS and how stSPARQL queries are processed to produce relevant SQL queries. Afterwards, we discussed about migrating from PostgreSQL/PostGIS to MonetDB and finally, we described some new functionality that were added in Strabon, namely, update operation, aggregate functions on spatial objects and support of various Coordinate Reference Systems.

3. Using Strabon in a real world scenario

In this chapter we present how Strabon can be used in a real world scenario. We demonstrate how to use Strabon to build a semantics-based EO Web portal inspired by the NOA use case of TELEIOS [KPMm]. Thus, it is oriented towards fire monitoring applications.

In this chapter, we briefly describe the ontologies that we developed for representing data that are being used or produced by the processing chain of NOA and the content of the produced products. We also provide example queries that can be used for data discovery. Afterwards, we present datasets exposed as Linked Open Data that are being used in a mockup fire monitoring application.

In this chapter, we use the namespaces defined in Table 3.1.

3.1 Creating and Populating Ontologies for the NOA Use Case

In the context of TELEIOS, we have developed three ontologies for annotating raw data that are being used in the NOA processing chain and the produced hotspot products. These ontologies are the following:

- **Image Annotation Ontology:** A domain ontology that models EO data collections with respect to use cases.
- **NOA Ontology:** An ontology that models raw data and products that are being consumed or produced by NOA.
- **Corine Ontology:** An ontology that models the CORINE Land Cover nomenclature.

The purpose of these ontologies is to model collections of data that are being used or produced by NOA. Data are separated in two main categories: **raw data** and **products**. Raw data include files with satellite measurements before any processing has taken place, while products are the result of processing on raw data.

In this section we present the aforementioned ontologies and give examples on how these ontologies are populated. First we present the NOA ontology that is used for modeling raw data and products that are being consumed or produced by NOA. Then, we present the ImageAnnotation Ontology that models EO data collections and finally, we present the Corine Land Cover Ontology that models the CORINE Land Cover nomenclature.

3.1.1 NOA Ontology

The NOA Ontology covers useful properties of the data collections that are being used or produced by the NOA processing chain. In the NOA Ontology, data are distinguished in three major categories:

namespace	prefix
`http://strdf.di.uoa.gr/ontology#`	`strdf`
`http://www.w3.org/1999/02/22-rdf-syntax-ns#`	`rdf`
`http://linkedgeodata.org/triplify/`	`lgd`
`http://linkedgeodata.org/ontology/`	`lgdo`
`http://www.w3.org/2000/01/rdf-schema#`	`rdfs`
`http://linkedgeodata.org/property/`	`lgdp`
`http://linkedgeodata.org/triplify/way10973689/`	`lgdw_-way10973689`
`http://www.georss.org/georss/`	`georss`
`http://www.openlinksw.com/schemas/virtrdf#`	`virtrdf`
`http://www.w3.org/2003/01/geo/wgs84_pos#`	`geo`
`http://www.geonames.org/ontology#`	`gn`
`http://teleios.di.uoa.gr/ontologies/noaOntology.owl#`	`noa`
`http://www.w3.org/2002/07/owl#`	`owl`
`http://dbpedia.org/resource/`	`dbpedia`
`http://teleios.di.uoa.gr/ontologies/imageAnnotationOntology.owl#`	`imAn`
`http://sweet.jpl.nasa.gov/2.0/data.owl#`	`data`

Table 3.1: Namespaces used and corresponding prefixes

- **Raw Data:** Instances of this class describe files with raw data.
- **Hotspot:** Instances of this class describe hotspots that were detected by processing in raw data.
- **Shape File:** An ESRI shapefile is created from all hotspots detected by a specific acquisition. Instances of this class describe these ESRI shapefiles.

Other classes that are defined in the NOA Ontology are:

- **Coastline:** Instances of this class describe areas that define the coastline of Greece.
- **Confirmation Value:** This class contains only three instances `noa:confirmed`, `noa:false_-alarm` and `noa:unknown`. These instances are assigned to hotspots and indicate that the hotspot has been confirmed to be a fire or a false alarm or that no confirmation has yet taken place.
- **Organization:** This class contains instances that represent EO Organizations (e.g., NOA, FIRMS, EFFIS).

We now give some examples of stRDF triples that describe EO products.

Example 10. *stRDF triples that represent a raw file with name* `H-000-MSG1-MSG1_RSS-IR_039-000007-201008211900-C_`.

```
noa:example_HMSG1_RSS_IR_039_s7_100821_1900_rd rdf:type noa:RawData;
  noa:belongToCollection imAn:MSG1.METEOSAT8.RAWDATA;
  noa:isDerivedFromSatellite "METEOSAT8"^^xsd:string;
  noa:isDerivedFromSensor "MSG1_RSS"^^xsd:string;
  noa:hasSegment "7"^^xsd:integer;
  noa:hasAqcuisitionTime "2010-08-21T19:00:00"^^xsd:dateTime;
  noa:isFromChannel "IR_039"^^xsd:string;
  noa:hasFilename "H-000-MSG1-MSG1_RSS-IR_039-000007-201008211900-C_"^^xsd:string.
```

Each raw file is annotated with metadata that are extracted from its filename. In the NOA use case, three raw datasets are used: MSG/SEVIRI archives, GMES ERCS FMM-1 archives and GMES ERCS FMM-2 archives. Here we show how to annotate a raw data file of MSG/SEVIRI archives. The filename of a raw data file of this dataset has the following format:

```
H-000-<sensor>-<sensor>-<channel>-<segment>-<datetime>-C_
```

The file with name `H-000-MSG1-MSG1_RSS-IR_039-000007-201008211900-C_` *is produced by a measurement taken from MSG1_RSS sensor, which is attached in METEOSAT8 satellite. The sensor channel that was used is IR_039 and acquired data cover the geographical segment 7. Finally, the sensing date is "2010-08-24" and the time is "19:00".*

Example 11. *stRDF triples that represent an ESRI shapefile with name* `HMSG1_RSS_IR_039_s7_100821_1900.shp`.

```
noa:example_HMSG1_RSS_IR_039_s7_100821_1900_shp a noa:ShpFile ;
  noa:isProducedBy noa:Noa ;
  noa:belongToCollection imAn:MSG1Collection ;
  noa:hasAqcuisitionTime "2010-08-21T19:00:00"^^xsd:dateTime ;
  noa:isDerivedFromSensor "MSG1_RSS"^^xsd:string ;
  noa:isDerivedFromSatellite "METEOSAT8"^^xsd:string ;
  noa:producedFromProcessingChain "Cloud-Mask-v1"^^xsd:string ;
  noa:hasFilename "HMSG1_RSS_IR_039_s7_100821_1900.shp"^^xsd:string.
```

The ESRI shapefiles that depict a group of hotspots are named according to the format below:

```
H<sensor>_<channel>_<segment_<date>_<time>.shp
```

Consequently, the shapefile that is produced by the raw data presented in Example 10, is named `HMSG1_RSS_IR_039_s7_100821_1900.shp`.

Example 12. *stRDF triples that represent a hotspot of the shapefile of Example 11.*

```
noa:example_HMSG1_RSS_IR_039_s7_100821_1900_0 a noa:Hotspot;
  noa:hasConfirmation noa:unknown;
  isProducedBy noa:noa;
  hasGeometry "POLYGON((24.42 38.06,24.47 38.06,24.47 38.02,
               24.42 38.02,24.42 38.06))"^^strdf:WKT;
  isDerivedFromSatellite "METEOSAT8"^^xsd:string;
  isDerivedFromSensor "MSG1_RSS"^^xsd:string;
  producedFromProcessingChain "Cloud-Mask-v1";
  hasConfidence "1.0"^^xsd:double;
  hasAqcuisitionTime "2010-08-21T19:00:00"^^xsd:dateTime.
```

Each hotspot that is contained in a shapefile should be annotated separately. For each hotspot, we represent its distinct geometry and confidence. In addition, we provide information such as the acquisition time that is derived from the corresponding shapefile.

3.1.2 Corine Land Cover Ontology

This ontology aims to model the CORINE Land Cover (CLC) nomenclature[1]. It has two main classes `clc:Area` and `clc:LandUse`. The class `clc:LandUse` is the top class of CLC taxonomy and contains classes that model the full hierarchy of land uses. The class `clc:Area` represents every area with a specific land use. The main properties of an instance of class `clc:Area` are the following.

- `strdf:hasGeometry`
- `clc:hasLandUse`

Property `noa:hasGeometry` indicates where an area lies, while `clc:hasLandUse` indicates the land use of an area connecting it with an instance of `clc:LandUse`. The stRDF description of an area that lies in a coniferous forest is included in Example 13.

Example 13. *stRDF triples that represent an area covered by coniferous forest.*

```
clc:Area_45 a clc:Area;
    noa:hasGeometry "POLYGON((22.07 40.62, ..., 22.07 40.62))"^^strdf:WKT;
    clc:hasLandUse clc:coniferousForest.
```

The above triples have been created using the NOA and CLC ontologies. They define an area covered by coniferous vegetation. Please note that `clc:coniferousForest` *is an instance of class* `clc:ConiferousForest` *which takes part in the taxonomy of LandUse.*

3.1.3 ImageAnnotation Ontology

The Image Annotation Ontology is an extension of the Earth Observation Lightweight Ontology (EOLO), described in [Pod]. EOLO aims at connecting the world of EO observation scientists with the world of data management technicians. It defines use cases with respect to observed objects of interest and their physical properties. Data collections are also defined in the same manner. Therefore, there is a matching between an application and data collections that are needed for this application. In this section, we describe how the NOA use case and respective datasets can be described using an extension of the EOLO ImageAnnotation Ontology. The main classes of the EOLO ontology are:

- **Phenomenon:** Natural or anthropogenic phenomena that can be studied.
- **ApplicationDomain:** Axioms that include several restrictions on the Collection properties.
- **Collection:** Data with similar characteristics.
- **ObservedObject:** Objects that can be observed (e.g., atmosphere, land surface).
- **PhysicalProperty:** Physical properties that can be measured (e.g., temperature, pressure).
- **measureRelation:** Pairs of observed object and a physical property. This class is used to represent the n-ary relation of property measures between the classes `data:Collection`, `data:ObservedObject` and `PhysicalProperty`. To increase readability of the ontology we created an equivalent class named `imAn:PropertyOfObject`.

[1] `http://www.eea.europa.eu/publications/COR0-landcover`

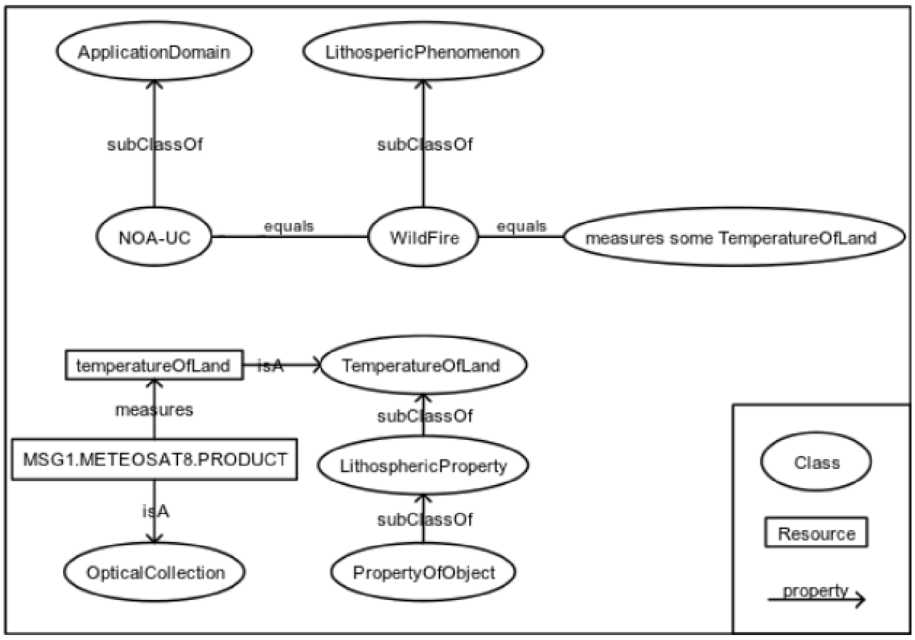

Figure 3.1: ImageAnnotation Ontology

To shed more light on the modeling EOLO follows, we present how the NOA use case can be modeled in EOLO. The class `imAn:NOA-UC` is added as a subclass of `eolo:ApplicationDomain` and represents the NOA Use Case. Since NOA is interested in observing wild fires, the class `imAn:WildFire` is added as a subclass of class `eolo:Phenomenon` and is set to be equivalent to `imAn:NOA-UC`, to restrict NOA Use Case only in wild fire phenomena. To observe wild fires, one needs data about land temperature. The property `eolo:measures` connects an `eolo:Collection` instance with an `imAn:PropertyOfObject` instance so that it models the collection of data being measured. If we set the class `imAn:WildFire` to be equal to the axiom "`eolo:measures some eoloTemperatureOfLand`" then every data collection like `imAn:MSG1Collection` will be an instance of class `imAn:WildFire`. The Image Ontology is shown in Figure 3.1 and a data collection described in Example 14.

Example 14. *stRDF triples that represent a data collection.*

```
iman:MSG1Collection rdf:type eolo:OpticalCollection;
                    eolo:hasGeoExtents eolo:Regional ;
                    eolo:hasProcessingLevel eolo:L1 ;
                    eolo:hasResolution eolo:LowResolution ;
                    eolo:hasTemporalExtents eolo:Instant ;
                    rdfs:label "MSG1Collection" ;
                    eolo:measures eolo:temperatureOfLand_ind .
```

The EOLO ontology defines properties that can be used for the qualitative characterization of data. In the example above, the collection `MSG1Collection` *contains current measurements with a regional spatial extent. It provides data with low resolution that are produced by processing level 1, which means that they are the product of some basic processing.*

3.2 Linked Oped Data for Fire Monitoring

In the Fire Monitoring application we are interested in a technological solution that integrates multiple, heterogeneous data sources in order to produce complete thematic maps. In order to increase the value of the final map product we interconnect EO data with Linked Open Data which is a new, rapidly developing research area of Semantic Web that studies how one can make RDF data available on the Web and interconnect them with other data with the aim of increasing its value for everybody [BHBL09]. The resulting "Web of data" has recently started being populated with geospatial data. Two representative examples of such efforts are LinkedGeoData[2] (LGD) and GeoNames[3] that are used in TELEIOS to enhance products related to fire monitoring. In the following sections, the datasets utilized in TELEIOS in the context of the fire monitoring application are described.

3.2.1 LinkedGeoData

LinkedGeoData is an effort aiming at enriching available geographic information published as Linked Data. LinkedGeoData is primarily focused on publishing OpenStreetMap (OSM)[4] data as Linked Data.

The data model of OpenStreetMap consists of three main categories:

Nodes: Point locations with latitude/longitude values.

Ways: Ordered sequences of nodes.

Relations: Groupings of nodes and ways.

The respective ontology, LinkedGeoData, is derived mainly from OSM tags, i.e., attribute-value annotations of nodes, ways, and relations, counting up to 500 classes, 50 object properties and about 15,000 datatype properties. Points are used by OSM to represent places, cities, etc. and are defined by a longitude/latitude pair expressed in the WGS84 coordinate reference system. Furthermore, an effort was made to match DBpedia resources with LinkedGeoData, taking into account the common classes between the two ontologies, for which the `owl:sameAs` link was used to connect them. Currently, the knowledge base of LGD comprises approximately 2 billion triples.

For the mockup fire monitoring application, we isolated data concerning Greece which led to approximately 840,000 triples. In this section we present some sample data derived from the LGD dataset.

Example 15. *RDF representation of the Parthenon according to LinkedGeoData.*

```
lgd:way10973689 rdf:type         lgdo:Tourism, lgdo:Building, lgdo:HistoricRuins ;
                lgdp:int_name    "Parthenon" .
                lgdo:hasNodes    lgdw_way10973689:nodes .
                georss:polygon   "37.9715909 23.7262015 37.9712993 23.7262856 37.9714414
                                 23.7270791 37.971733 23.7269951 37.9715909 23.7262015" .

lgdw_way10973689:nodes rdf:type rdf:Seq .
lgdw_way10973689:nodes rdf:_1   lgd:node97810425 ;
                       rdf:_2   lgd:node97810428 ;
                       rdf:_3   lgd:node97810431 ;
                       rdf:_4   lgd:node97810434 ;
                       rdf:_5   lgd:node97810425 .

lgd:node97810434       geo:geometry "POINT(23.727 37.9717)"^^virtrdf:Geometry .
```

[2] http://linkedgeodata.org/
[3] http://www.geonames.org/
[4] http://www.openstreetmap.org/

```
lgd:node97810425       geo:geometry "POINT(23.7262 37.9716)"^^virtrdf:Geometry .
lgd:node97810428       geo:geometry "POINT(23.7263 37.9713)"^^virtrdf:Geometry .
lgd:node97810431       geo:geometry "POINT(23.7271 37.9714)"^^virtrdf:Geometry .
ns2:node97810425       geo:geometry "POINT(23.7262 37.9716)"^^virtrdf:Geometry .
```

The triples above describe Parthenon according to the LGD dataset, that is derived from OSM data. Note that the geometry of Parthenon is defined both using a sequence of constituent nodes and their geometry along with the object value of property `georss:polygon` *that is a string with all lat/long pairs of constituent nodes.*

Example 16. *stRDF triples that represent the Parthenon according to LinkedGeoData.*

```
lgd:way10973689  rdf:type       lgdo:Tourism, lgdo:Building, lgdo:HistoricRuins;
                 lgdo:hasNodes  lgdw_way10973689:nodes ;
                 lgdp:int_name  "Parthenon" ;
                 georss:polygon "POLYGON(( 23.7262015 37.9715909, 23.7262856 37.9712993,
                                23.7270791 37.9714414, 23.7269951 37.971733, 23.7262015
                                37.9715909))"^^<http://strdf.di.uoa.gr/ontology#WKT> ;
```

In order to utilize LGD data in a fire monitoring application using Strabon, we omitted constituent nodes for each way and defined its geometry according to the stRDF data model [KKN+11].

3.2.2 GeoNames

GeoNames[5] is a gazetteer which collects both spatial and thematic information for various place-names around world. Data of GeoNames is available through various Web services, but are also published as Linked Data. The database of GeoNames contains over 10 million geographical names corresponding to over 7.5 million unique features. We used data related with Greece that comprise approximately 575,000 triples describing around 41,000 features.

GoeNames provides a point approximation for the geometry of each feature and is expressed as a longitude/latitude pair expressed in WGS84. The following example shows how Athens is represented according to the GeoNames ontology.

Example 17. *RDF representation of Athens according to GeoNames.*

```
<http://sws.geonames.org/264371/> a gn:Feature ;
    wgs84_pos:lat "37.97945" ;
    wgs84_pos:long "23.71622" ;
    gn:alternateName "Athens"@en ;
    gn:countryCode "GR" ;
    gn:featureClass gn:P ;
    gn:featureCode gn:P.PPLC ;
    gn:locationMap <http://www.geonames.org/264371/athens.html> ;
    gn:name "Athens" ;
    gn:nearbyFeatures <http://sws.geonames.org/264371/nearby.rdf> ;
    gn:parentADM1 <http://sws.geonames.org/6692632/> ;
    gn:parentADM2 <http://sws.geonames.org/264353/> ;
    gn:parentCountry <http://sws.geonames.org/390903/> ;
    gn:parentFeature <http://sws.geonames.org/264353/> ;
    gn:population "729137" ;
```

[5]http://www.geonames.org/

```
    gn:wikipediaArticle <http://af.wikipedia.org/wiki/Athene> ;
    rdfs:isDefinedBy "http://sws.geonames.org/264371/about.rdf" ;
    owl:sameAs dbpedia:Athens ;
    wgs84_pos:alt "70" .
```

The triples above describe Athens according to the GeoNames ontology. In order to use the spatial extent of this description in Strabon we replaced the triples representing the location of Athens with the following triple
`<http://sws.geonames.org/264371/> noa:hasGeography "POINT(23.71622 37.97945)"^^strdf:WKT.`

3.2.3 Greek administrative geography

Recently, governments publish open data about administrative divisions, statistical information, etc. in their portals [GDH08]. We have taken advantage of this data openness for Greece[6] and since these haven't been published as Linked Data in the past, we chose to publish them and focused on the Greek administrative division and the geometry of lowest divisions (municipalities).

3.3 Building a Fire Monitoring Application Using Strabon

In this section we show how Semantic Web technologies and Linked Open Data can enhance Earth Observation applications like fire monitoring. Specifically, we demonstrate how to use the aforementioned datasets to build a Fire Monitoring Application for the NOA use case. We demonstrate how to discover raw data, shapefiles using stSPARQL queries and show how one can use stSPARQL to generate on the fly maps for the fire monitoring domain.

3.3.1 Discovering Earth Observation Data

In this section we demonstrate how to discover Earth Observation data, such as files containing raw data, by using stSPARQL to express such queries.

Example 18. *Retrieve shapefiles that contains acquisitions performed by sensor* `MSG1_RSS` *that were taken between 20:00 and 20:30 of August 21, 2010.*

```
SELECT     ?filename
WHERE {    ?file rdf:type noa:ShpFile .
           ?file noa:hasFilename ?filename .
           ?file noa:hasAcquisitionTime ?sensingTime .
           FILTER( str(?sensingTime) > "2010-08-21T20:00:00" ) .
           FILTER( str(?sensingTime) < "2010-08-21T20:30:00" ) .
           ?file noa:isDerivedFromSensor ?sensor .
           FILTER( str(?sensor) = "MSG1_RSS" ) . }
```

The result of the query is displayed below: The above query searches for shapefiles (instances of class `noa:ShpFile`*). Since each shapefile contains information from a specific acquisition, the above query searches for acquisitions of the sensor* `MSG1_RSS` *taken between 20:00 and 20:30 of August 21, 2010. For every instance found the respective filename is returned. The user can utilize this information to download the respective file.*

[6] `http://geodata.gov.gr/geodata/`

?filename
MSG1_RSS_10-08-21_20:05_cloud-masked.shp
MSG1_RSS_10-08-21_20:05_plain.shp
MSG1_RSS_10-08-21_20:10_cloud-masked.shp
MSG1_RSS_10-08-21_20:10_plain.shp
MSG1_RSS_10-08-21_20:15_cloud-masked.shp
MSG1_RSS_10-08-21_20:15_plain.shp
MSG1_RSS_10-08-21_20:20_cloud-masked.shp
MSG1_RSS_10-08-21_20:20_plain.shp
MSG1_RSS_10-08-21_20:25_cloud-masked.shp
MSG1_RSS_10-08-21_20:25_plain.shp

Example 19. *Discover all files related to the NOA use case.*

```
SELECT ?filename {
  ?collection rdf:type iman:NOA-UC .
  ?file noa:belongToCollection ?collection .
  ?file noa:hasFilename ?filename . }
```

Some of the results of the query are displayed below:

?filename
H-000-MSG1__-MSG1_RSS____-IR_039___-000007___-201008211900-C_
H-000-MSG1__-MSG1_RSS____-IR_039___-000007___-201008211905-C_
H-000-MSG1__-MSG1_RSS____-IR_039___-000007___-201008211910-C_
....
MSG1_RSS_10-08-21_19:00_cloud-masked.shp
MSG1_RSS_10-08-21_19:00_plain.shp
MSG1_RSS_10-08-21_19:05_cloud-masked.shp
MSG1_RSS_10-08-21_19:05_plain.shp
MSG1_RSS_10-08-21_19:10_cloud-masked.shp
MSG1_RSS_10-08-21_19:10_plain.shp
MSG1_RSS_10-08-20_08:10_plain.shp
....

The above query make use of the ImageAnnotation ontology so that users who are not aware of the available data and are interested in products related to fire monitoring, can discover relative filenames. According to the extension of EOLO described in Section 3.1.3, only files that measure land temperature (or derive from such measurements) will be returned. As long as there is not some additional, restriction the results will be both files with raw data and produced shapefiles.

Example 20. *Discover the shapefiles that related to the NOA use case.*

```
SELECT ?filename {
  ?collection rdf:type iman:NOA-UC .
  ?file noa:belongToCollection ?collection .
  ?file noa:hasFilename ?filename .
  ?collection eolo:hasProcessingLevel eolo:L1 . }
```

A part of the result of the query is displayed below: The difference between this query and the previous one is that it imposes some more restrictions. It specifically asks only for files belonging to collections with processing level 1, i.e., results of processing on raw data. Thus, it will only retrieve filenames of shapefiles.

?filename
MSG1_RSS_10-08-21_19:00_cloud-masked.shp
MSG1_RSS_10-08-21_19:00_plain.shp
MSG1_RSS_10-08-21_19:05_cloud-masked.shp
MSG1_RSS_10-08-21_19:05_plain.shp
MSG1_RSS_10-08-21_19:10_cloud-masked.shp
MSG1_RSS_10-08-21_19:10_plain.shp
MSG1_RSS_10-08-20_08:10_plain.shp
....

3.3.2 Fire Monitoring

As far as automatic map generation is concerned, Semantic Web technologies provide tools for handling heterogeneous data in a homogeneous way, while Linked Open Data Cloud supplies an abundance of data in addition to internal EO data. Thus, a user has in her hands datasets covering a large variety of geospatial information from minor, but interesting, entities like fire stations or hospitals, to large entities like countries and their administrative divisions. So, instead of manually combining heterogeneous data, a user can pose an stSPARQL query for each layer that she wants to depict in a map and overlay the retrieved data. In order to depict the query results in a map, there should be exactly one geospatial variable in the select clause of each query. Results of such queries can be encoded in KML format or exported as shape files.

By posing the queries presented in this section and overlaying their results, one can create a map that depicts the fires of August of 2007 along with auxiliary information (Figure 3.2).

Example 21. *Retrieve all hotspots in Peloponnese that were detected from the 23rd to 26th of August 2007.*

```
SELECT ?hotspot ?hGeo ?hAcqTime ?hConfidence ?hProvider ?hConfirmation ?hSensor
WHERE {
  ?hotspot  a noa:Hotspot ;
            noa:hasGeometry ?hGeo ;
            noa:hasAcquisitionTime ?hAcqTime ;
            noa:hasConfidence ?hConfidence ;
            noa:isProducedBy ?hProvider ;
            noa:hasConfirmation ?hConfirmation ;
            noa:isDerivedFromSensor ?hSensor ;
            FILTER( "2007-08-23T00:00:00" <= str(?hAcqTime) ) .
            FILTER( str(?hAcqTime) <= "2007-08-26T23:59:59" ) .
            FILTER(  strdf:contains("POLYGON((21.027 38.36, 23.77 38.36, 23.77
                     36.05, 21.027 36.05, 21.027 38.36))"^^strdf:WKT, ?hGeo) ) .}
```

This query retrieves all hotspots that where detected between 23rd-26th of August 2007. Along with the URIs, additional information about the hotspots is retrieved, e.g., time of acquisition (`?hAcqTime`), provider (`?hProvider`) and geometry (`?hGeo`).

Example 22. *Retrieve all areas in Peloponnese that are characterized as water bodies, artificial surfaces, wet lands, agricultural areas or forests and semi natural areas.*

```
SELECT ?area ?aGeo ?aLandUseType
WHERE {
    ?area  a clc:Area ;
```

```
        clc:hasLandUse ?aLandUse ;
        noa:hasGeometry ?aGeo .
    ?aLandUse a ?aLandUseType .
    FILTER( ?aLandUseType = clc:Water_Bodies || ?aLandUseType = clc:ArtificialSurfaces
         || ?aLandUseType = clc:Wetlands || ?aLandUseType = clc:AgriculturalAreas
         || ?aLandUseType = clc:ForestsAndSemiNaturalAreas ) .
    FILTER( strdf:contains("POLYGON((21.027 38.36, 23.77 38.36, 23.77 36.05,
            21.027 36.05, 21.027 38.36))"^^strdf:WKT, ?aGeo) ) . }
```

This query retrieves all areas with the desired land use according the Corine nomenclature and their respective first level categorization.

Example 23. *Get all primary roads in Peloponnese.*

```
SELECT ?road ?rGeo
WHERE {
    ?road  a ?rType ;
    noa:hasGeometry ?rGeo .
    FILTER( ?rType = lgdo:Primary ) .
    FILTER( strdf:contains("POLYGON((21.027 38.36, 23.77 38.36, 23.77 36.05,
            21.027 36.05, 21.027 38.36))"^^strdf:WKT, ?rGeo) ) .}
```

This query utilizes information from the LinkedGeoData dataset and retrieves all primary roads of Peloponnese.

Example 24. *Get all seats of a first-order administrative division in Peloponnese.*

```
SELECT ?feature ?fName ?fGeo
WHERE {
    ?feature  a gn:Feature ;
    noa:hasGeography ?fGeo ;
    gn:name ?fName ;
    gn:featureCode ?fCode .
    FILTER( ?fCode = gn:P.PPLA || ?fCode = gn:P.PPLA2 ) .
    FILTER( strdf:contains("POLYGON((21.51 36.41, 22.83 36.41,
            22.83 37.69, 21.51 37.69, 21.51 36.41))"^^strdf:WKT, ?fGeo)). }
```

This query asks for every feature in GeoNames that is located at Peloponnese and its `gn:featureCode` *is* `gn:P.PPLA` *that is the code for seats of first-order administrative divisions. Apart from information about every node (variables* `?n` *and* `?nGeo`*), the geometry (*`?nGeo`*) of the feature is also returned so it can be depicted in a map.*

Example 25. *Get all Municipality boundaries in Peloponnese.*

```
SELECT ?municipality ?mYpesCode ?mContainer ?mLabel ( strdf:boundary(?mGeo) as ?mBoundary )
WHERE {
    ?municipality  a gag:Dhmos ;
    noa:hasYpesCode ?mYpesCode ;
    gag:isPartOf ?mContainer ;
    rdfs:label ?mLabel ;
    strdf:hasGeometry ?mGeo .
    FILTER( strdf:contains("POLYGON((21.027 38.36, 23.77 38.36, 23.77 36.05,
            21.027 36.05, 21.027 38.36))"^^strdf:WKT, ?gGeo) ) . }
```

This query makes use of information related to the Greek Administrative Geography. Especially, it retrieves all lowest administrative divisions (municipalities) along with some information about them (`?gYpesCode`*,* `?gContainer` *and* `?gLabel`*) along with their boundaries.*

By posing the queries presented above and overlaying the results on a map, one can create a map that depicts the fires of August of 2007 along with auxiliary information (Figure 3.2).

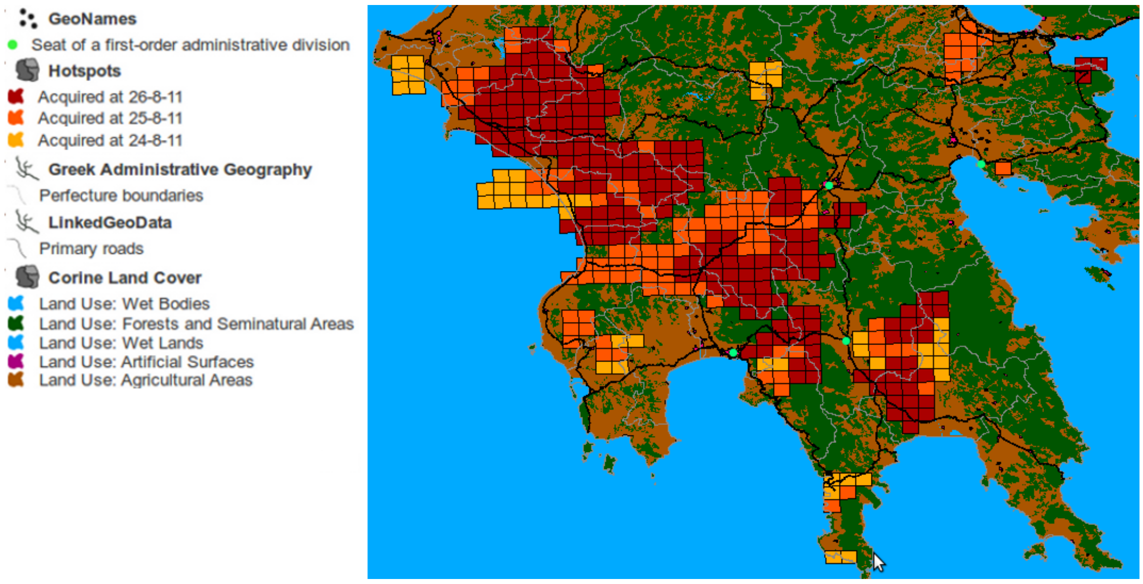

Figure 3.2: A map created by overlaying data retrieved by queries on EO and Linked Geospatial Data

3.4 Summary

In this chapter we presented a real world scenario where Strabon was used to build a fire monitoring application. We described the ontologies that were developed in the context of TELEIOS that model data used in fire monitoring by NOA. We also presented auxiliary datasets from the Linked Open Data Cloud that were used to enhance fire monitoring products. Finally we presented example queries that demonstrate how Strabon can be utilized both for data discovery and for creating mapping products with added value for fire monitoring by utilizing Semantic Web Technologies.

4. Source code

In this chapter we describe the directory structure of the Strabon prototype. We also briefly mention any external dependencies this implementation has. We then present the application programming interfaces that can be used in order to store stRDF files and evaluate stSPARQL queries in Strabon.

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

4.1 Directory structure

The source code of Strabon is available for download from a Mercurial repository[1] hosted by ACS. What we have made available for download is a transparent addon for Sesame 2.6.3. Following its download, the inner directories of the downloaded folder will have the following structure:

- `evaluation/`: Contains classes implementing the external geospatial functions we have added to Sesame's functionality. Also contains classes overriding Sesame's default functionality in the case of aggregate functions.
- `generaldb/`: Contains the abstract extension of Sesame's rdbms layer in order to allow the storage of stRDF triples and querying using stSPARQL.
- `monetdb/`: Specialization of generaldb that allows using MonetDB as a backend for Sesame.
- `postgis/`: Specialization of generaldb that allows using PostGIS as a backend for Sesame.
- `resultio/`: Contains classes enabling the output of triples containing spatial literals.
- `jars/`: After compiling Strabon, this folder contains all necessary jars for running Strabon.
- `runtime/`: Contains the classes exposing Strabon's functionality to the user.
- `pom.xml`: XML file that contains information about the project and configuration details used by Maven to build Strabon.

4.2 External Dependencies

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

- PostgreSQL JDBC
- PostGIS
- MonetDB JDBC
- Maven (along with many Maven plugins, e.g., surefire)
- GeoAPI
- GeoTools
- JUnit
- Apache Commons
- SLF4J: Simple Logging Facade for Java

[1]ssh://opteleios@teleios-repo.acsys.it//home/opteleios/TELEIOS/system

4.3 Application Programming Interface

The class that is used to display Strabon's functionality is `Strabon`, contained in the `runtime` package. The implementation of the class slightly differs depending on the relational backend (PostGIS / MonetDB). The functions used to access Strabon are the following:

- `eu.earthobservatory.runtime.postgis.Strabon`: The constructor of our system in the case of a PostGIS backend. This class creates a repository and establishes a connection with it.
- `eu.earthobservatory.runtime.monetdb.Strabon`: The constructor of our system in the case of a MonetDB backend. This class creates a repository and establishes a connection with it.
- `store`: This function stores in the underlying repository the triples contained in the file or URL specified by the user. 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.
- `query`: This function can be used in order to pose queries on previously stored data. In addition to the query's string representation, the user can specify the format in which he wants his results displayed. Supported formats include RDF/XML, KML and HTML representation.
- `update`: This function can be used in order to pose update queries on previously stored data or on an empty graph. If update query is posed on an empty graph it is expected to be a data insertion

4.4 Summary

In this chapter we provided a brief description of the source code of our implementation. We described the directory structure, listed the external dependencies, and enumerated the methods of the API.

5. Installation and Execution

In this chapter, we provide Strabon installation guidelines, using either MonetDB or PostGIS backend, and present how we can store and query stRDF datasets using Strabon.

For the convenience of the reader, we give specific instructions for executing each step in a computer running Ubuntu 11.10 64-bit.

5.1 Prerequisites

Before downloading and compiling Strabon, the following steps should be executed:

- Install maven. More information can be found at `http://maven.apache.org/`.

  ```
  $> sudo apt-get install maven2
  ```

- Install java 6. More information can be found at `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

  ```
  $> sudo add-apt-repository ppa:ferramroberto/java
  $> sudo apt-get update
  $> sudo apt-get install sun-java6-jdk sun-java6-plugin
  ```

- Install Mercurial. More information can be found at `http://mercurial.selenic.com/`.

```
$> sudo apt-get install mercurial
```

5.2 Installing the underlying RDBMS

Let us now proceed with the installation of the RDBMS that will be used as a backend for Strabon. The following sections provide installation instructions for MonetDB and PostGIS respectively.

Installing MonetDB

In this section we provide some basic information on how to build MonetDB from sources. More information can be found in [ZKI+12] and `http://www.monetdb.org/Developers/SourceCompile`.

- Extract the source code of MonetDB from the accompanying code package of Deliverable D5.1 (`MonetDB-11.7.8-NOA_20120228a_SQL_C_Strabon.tar.bz2`).
- Compile and install MonetDB.

5.2.1 Installing PostgreSQL and PostGIS

- Install PostgreSQL 9.0 or higher. More information can be found at `http://www.postgresql.org/download/`.

```
$> sudo apt-get install postgresql-9.1
```

- Install PostGIS 1.5 or higher. More information can be found at `http://postgis.refractions.net/download/`.

```
$> sudo apt-get install postgresql-9.1-postgis
```

- Provide a password for default user (postgres)

```
$> sudo -u postgres psql -c "ALTER USER postgres WITH PASSWORD ’postgres’;"
```

5.2.2 Creating a spatially enabled database

Spatially-enabled databases permit the use of spatial function calls. MonetDB creates spatially-enabled databases by default if you have enabled the `geom` module. More information on how to create a spatially-enabled database in PostGIS can be found at `http://postgis.refractions.net/docs/`.

- Set postgis-1.5 path.

```
$> POSTGIS_SQL_PATH=‘pg_config --sharedir‘/contrib/postgis-1.5
```

- Create the spatial database that will be used as a template.

```
$> createdb -E UTF8 -T template0 template_postgis
```

- Add PLPGSQL language support.

```
$> createlang -d  template_postgis plpgsql
```

- Load the PostGIS SQL routines.

```
$> psql -d template_postgis -f $POSTGIS_SQL_PATH/postgis.sql
$> psql -d template_postgis -f $POSTGIS_SQL_PATH/spatial_ref_sys.sql
```

- Allow users to alter spatial tables.

```
$> psql -d template_postgis -c "GRANT ALL ON geometry_columns TO PUBLIC;"
$> psql -d template_postgis -c "GRANT ALL ON geography_columns TO PUBLIC;"
$> psql -d template_postgis -c "GRANT ALL ON spatial_ref_sys TO PUBLIC;"
```

- Perform garbage collection.

```
$> psql -d template_postgis -c "VACUUM FULL;"
$> psql -d template_postgis -c "VACUUM FREEZE;"
```

- Allows non-superusers the ability to create from this template.

```
$> psql -d postgres -c "UPDATE pg_database SET datistemplate='true'
   WHERE datname='template_postgis';"
$> psql -d postgres -c "UPDATE pg_database SET datallowconn='false'
   WHERE datname='template_postgis';"
```

- Create a spatially-enabled database named `endpoint`.

```
$> createdb endpoint -T template_postgis
```

5.3 Compiling and Running Strabon

The following sections describe how we can download, install and use Strabon.

5.3.1 Download and Compile Strabon

In order to download and compile Strabon, the steps below should be followed:

- Clone the source code of the TELEIOS system from the TELEIOS mercurial repository hosted by ACS.

```
$> hg clone ssh://opteleios@teleios-repo.acsys.it//home/opteleios/TELEIOS/system
```

- Change to the directory that Strabon source code resides in.

```
$> cd system/components/Strabon
```

- Build Strabon.

```
$> mvn clean package
```

5.3.2 Store

If the installation of Strabon has been completed successfully, the user can proceed to store an stRDF dataset. If PostGIS is used as the relational backend, the user has to execute the `StoreOp` class of the package `eu.earthobservatory.runtime.postgis` of module `strabon-runtime`.

This class requires the following arguments:

- `HOST`: The name of the database host (e.g., localhost)
- `PORT`: The port which your connection listens to.

- `DATABASE`: The name of the spatially-enabled database you created previously.
- `USERNAME`: The username used to connect to the database.
- `PASSWORD`: The password used to connect to the database.
- `FILE`: The stRDF document to be stored.
- `FORMAT`: The format of the triples.

Incremental storing using MonetDB as relational backend is currently work in progress, and will be completed in subsequent releases of Strabon. As far as bulk storage of triples is concerned, the user can store triples in either MonetDB or Postgres using the loader we created. This program performs bulk loading in both MonetDB and PostGIS, and is provided in the accompanying code package with instructions on how to setup and run it. The only argument that the Loader requires is the name of the file containing all the stRDF triples to be stored. An example demonstrating the usage of Loader is the script `createDBmonet.sh`, located in the accompanying code package of this deliverable. The script invokes an instance of the Loader, using the dataset utilized in the NOA use case as input. The loader produces numerous csv files that are subsequently stored in MonetDB tables that can be used as the relational backend of Strabon.

We present an example on how to run Strabon from a console, connect to PostgreSQL and store an RDF file. Let us assume that the file shown in Figure 5.1 resides in `/tmp/triples.nt` (encoded according to the N3 format) and a spatially-enabled database named `endpoint` has been created:

```
@prefix dc:   <http://purl.org/dc/elements/1.1/> .
@prefix :     <http://example.org/book/> .
@prefix ns:   <http://example.org/ns#> .

:book1  dc:title  "SPARQL Tutorial" .
:book1  ns:price  42 .
:book2  dc:title  "The Semantic Web" .
:book2  ns:price  23 .
```

Figure 5.1: Example triples in N3 format

After executing the following command the file will be stored in the `endpoint` database.

```
$> cd jars/target && java -cp $(for file in ‘ls -1 *.jar‘; do myVar=$myVar./$file":";
done;echo $myVar;) eu.earthobservatory.runtime.postgis.StoreOp localhost 5432 endpoint
postgres postgres /tmp/triples.nt N3
```

5.3.3 Query

If storing has completed successfully, evaluation of stSPARQL queries can be performed using Strabon. The user has to use the corresponding `QueryOp` class of the package `eu.earthobservatory.runtime.postgis` or `eu.earthobservatory.runtime.monetdb` of module `strabon-runtime`. This class requires the following arguments:

- `HOST`: The name of the database host (e.g., localhost)
- `PORT`: The port which your connection listens to.
- `DATABASE`: The name of the spatially-enabled database you created previously.

- `USERNAME`: The username used to connect to the database.
- `PASSWORD`: The password used to connect to the database.
- `QUERY`: The stSPARQL query to evaluate.
- `FORMAT`: The format of the results.

Further to our previous example we present how one can run Strabon from a console, connect to PostgreSQL and evaluate an stSPARQL query. Suppose that the user wants to retrieve titles of books above a specific value (e.g., 30 dollars). Then, she should run the following command:

```
$> cd jars/target && java -cp $(for file in ‘ls -1 *.jar‘; do myVar=$myVar./$file":";
done;echo $myVar;) eu.earthobservatory.runtime.postgis.QueryOp  localhost 5432 endpoint
postgres postgres "PREFIX dc: <http://purl.org/dc/elements/1.1/> PREFIX ns:
<http://example.org/ns#> SELECT ?title WHERE {?book dc:title ?title. ?book ns:price
?price. FILTER(?price > 30)}";
```

The results that will be returned are:

?title
“SPARQL Tutorial”

5.3.4 Update

In addition, one can perform stSPARQL updates in Strabon by using the `UpdateOp` class of the package `eu.earthobservatory.runtime.postgis` of module `strabon-runtime`. This class requires the following arguments:

- `HOST`: The name of the database host (e.g., localhost)
- `PORT`: The port which your connection listens to.
- `DATABASE`: The name of the spatially-enabled database you created previously.
- `USERNAME`: The username used to connect to the database.
- `PASSWORD`: The password used to connect to the database.
- `UPDATE`: The stSPARQL update query to evaluate.

Update functionality using MonetDB as a relational backend is currently work in progress, and will be fully implemented in subsequent releases of Strabon.

In the following example, the command given replaces the price of book “The Semantic Web” with a higher one.

```
$> cd jars/target && java -cp $(for file in ‘ls -1 *.jar‘; do myVar=$myVar./$file":";
done; echo $myVar;) eu.earthobservatory.runtime.postgis.UpdateOp  localhost 5432 endpoint
postgres postgres "PREFIX dc: <http://purl.org/dc/elements/1.1/> PREFIX ns:
<http://example.org/ns#> DELETE {?book ns:price ?price} INSERT {?book ns:price 32} WHERE
{?book dc:title \"The Semantic Web\". ?book ns:price ?price.}";
```

After executing this command the results of the previous query will be:

?title
‘‘SPARQL Tutorial’’
‘‘The Semantic Web’’

5.4 Summary

In this chapter, we provided instructions on how to install Strabon. The user can pick between having MonetDB or PostGIS as a relational backend. We also presented how one can store and query stRDF datasets using Strabon, providing an example of how to run the Strabon PostGIS store and query methods from an Ubuntu 11.10 command line.

6. Conclusions

In this report we presented the implementation process followed for the development of the system Strabon. We documented the steps followed to port our implementation on top of MonetDB. We also discussed the process we followed to enhance our system with additional functionality motivated by the use cases of TELEIOS. What is more, we presented how Strabon can be used in a real world scenario. Finally, we gave a short tutorial on how to compile and run Strabon. This report accompanies the code for our implementation which is available for download from the TELEIOS Mercurial repository[1].

In the forthcoming deliverable named "An implementation of a temporal and spatial extension of RDF and SPARQL on top of MonetDB - Phase II" we will report on our work on extending Strabon in order to support $stRDF^i$.

[1]`teleios-repo.acsys.it`

Bibliography

[BHBL09] C. Bizer, T. Heath, and T. Berners-Lee. Linked data-the story so far. *Int. J. Semantic Web Inf. Syst.*, 2009.

[BNM10] Andreas Brodt, Daniela Nicklas, and Bernhard Mitschang. Deep integration of spatial query processing into native rdf triple stores. In *ACM SIGSPATIAL*, 2010.

[CDFvO93] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In David Abel and Beng Chin Ooi, editors, *Advances in Spatial Databases*, volume 692 of *Lecture Notes in Computer Science*, pages 277–295. Springer Berlin / Heidelberg, 1993.

[cga] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[Fuk] Komei Fukuda. cddlib library. `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html`.

[GDH08] J. Goodwin, C. Dolbear, and G Hart. Geographical Linked Data: The Administrative Geography of Great Britain on the Semantic Web. *Transactions in GIS*, 12:19–30, 2008.

[GPP] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 Update. W3C Working Draft 05 January 2012. `http://www.w3.org/TR/sparql11-update/`.

[HM83] 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.

[KKK$^+$11] Manos Karpathiotakis, Kostis Kyzirakos, Zoi Kaoudi, Manolis Koubarakis, Josep Rodriguez, and Juanjo Aparicio. Evaluation of the registry services. Deliverable D3.4 Version 1.0, SemSorGrid4Env, August 2011.

[KKN$^+$11] Manolis Koubarakis, Kostis Kyzirakos, Babis Nikolaou, Michael Sioutis, Stavros Vassos, and Consortium Members. A data model and query language for an extension of rdf with time and space. Deliverable D2.1 Version 1.0, TELEIOS, November 2011.

[KPMm] Charalabos (Haris) Kontoes, Ioannis Papoutsis, Dimitrios Michail, and Consortium members. Requirements specification for the real-time fire monitoring application. Deliverable D7.1, TELEIOS project.

[MRTT53] T.S. Motzkin, H. Raifa, GL. Thompson, 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.

[OGC10a] Open Geospatial Consortium Inc OGC. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option. OpenGIS Implementation Standard, 08 2010. `http://portal.opengeospatial.org/files/?artifact_id=25354`.

[OGC10b] Open Geospatial Consortium Inc OGC. OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common Architecture. OpenGIS Implementation Standard, 08 2010. `http://portal.opengeospatial.org/files/?artifact_id=25355`.

[Pod] Marius Podwyszynski. Knowledge-based search for Earth Observation products. Diploma Thesis.

[RSSG03] 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.

[ssg] Semantic Sensor Grids for Rapid Application Development for Environmental Management (SemsorGrid4Env). `http://www.semsorgrid4env.eu`.

[ZKI+12] Ying Zhang, Martin Kersten, Milena Ivanova, Holger Pirk, and Stefan Manegold. An implementation of ad-hoc array queries on top of MonetDB. Deliverable D5.1, TELEIOS, February 2012.