Deep integration of spatial query processing into native RDF triple stores

Andreas Brodt
Universität Stuttgart,
Universitätsstraße 38
70569 Stuttgart, Germany
brodt@ipvs.uni-stuttgart.de

Daniela Nicklas
Carl von Ossietzky Universität
Oldenburg
26111 Oldenburg, Germany
dnicklas@acm.org

Bernhard Mitschang
Universität Stuttgart,
Universitätsstraße 38
70569 Stuttgart, Germany
mitschang@ipvs.uni-stuttgart.de

ABSTRACT

Semantic Web technologies, most notably RDF, are well-suited to cope with typical challenges in spatial data management including analyzing complex relations between entities, integrating heterogeneous data sources and exploiting poorly structured data, e.g., from web communities. Also, RDF can easily represent spatial relationships, as long as the location information is symbolic, i.e., represented by places that have a name. What is widely missing is support for geographic and geometric information, such as coordinates or spatial polygons, which is needed in many applications that deal with sensor data or map data. This calls for efficient data management systems which are capable of querying large amounts of RDF data and support spatial query predicates. We present a native RDF triple store implementation with deeply integrated spatial query functionality. We model spatial features in RDF as literals of a complex geometry type and express spatial predicates as SPARQL filter functions on this type. This makes it possible to use W3C's standardized SPARQL query language as-is, i.e., without any modifications or extensions for spatial queries. We evaluate the characteristics of our system on very large data volumes.

Categories and Subject Descriptors

H.2 [**Database Management**]: Database Applications—*Spatial databases and GIS*

Keywords

RDF, SPARQL, triple store, spatial database, GIS

1. INTRODUCTION

The use of semantic web technologies—most notably RDF [10] and the SPARQL query language [16]—for integrating and analyzing data sets is widely acknowledged. The characteristics of their underlying data model make it easier to represent, exchange, combine, and link information from different, heterogeneous data sets. Huge repositories exist that publish and link data sets in RDF format, e.g., the data sets on Linked Data,[1] the Uniprot data set,[2] the Semantic Web Challenge 2010 data set,[3] or many data sets published on the data.gov catalog[4] of the USA. Those data sets can be used by researchers from many different domains.

Many of these data sets contain geographical or symbolic location information, and spatial relations between data entities play a crucial role for searching and analyzing spatial data. Consequently, the ontological modeling of geographic entities and their geospatial relations is a significant research direction in the Geographic Information Science (GIS) community. The Resource Description Framework (RDF) [10] developed as part of W3C's Semantic Web Activity[5] treats relationships as first class objects, which suits it very well to model and query complex relationships between resources. In addition, Semantic Web technologies deal well with problems such as non-unique names or subclass relationships, which typically occur in data integration tasks that are very common when working with spatial data [9]. Moreover, Semantic Web technologies provide a great deal of schema flexibility which is useful for analyzing and integrating poorly structured data, e.g., web- or community-based data, such as map data from the OpenStreetMap project [3].

RDF and related technologies were long deemed inefficient. Yet, in recent years the database community has addressed the problem of querying large amounts of RDF data efficiently and achieved enormous performance improvements in so-called RDF *triple stores* [1, 19, 13, 6]. As long as primitive data types, most notably strings, are concerned, these achievements can be well utilized to analyze geospatial data—depending on the used location model [4]: systems working with symbolic coordinate systems, for instance many indoor location systems, typically use a graph model connecting symbolic names. These systems can directly make use of efficient RDF data management. However, queries involving geographical positions represented as coordinates, e.g., range queries or nearest-neighbor queries, require vector-based coordinates. If these coordinates are represented as a string, no system can make use of the specific features of location information (e.g., the need for spatial indexes) or provide type-safe spatial functions. However, a widely

ACM GIS '10 , November 2-5, 2010. San Jose, CA, USA

[1] http://linkeddata.org/
[2] http://dev.isb-sib.ch/projects/uniprot-rdf/
[3] http://challenge.semanticweb.org
[4] http://www.data.gov
[5] http://www.w3.org/2001/sw/

accepted standard for representing spatial information exists [7], and the RDF data model allows for extension by new data types. But to the best of our knowledge, no RDF triple store does address native spatial data processing yet. Until now, efficient spatial analyses of RDF data would require external processing, e.g., using a geo-enabled database.

In this paper, we show how spatial information can be deeply integrated into the RDF data model, so that native RDF triple stores can efficiently process spatial queries and analyses. This is a leap forward from the state of the art (Section 2): For the RDF data model, deep integration means that we model geographic data in RDF as complex objects represented as literals of an abstract geometry type (following the OpenGIS Simple Features Specification [7]). By this, spatial features, such as coordinate-based points, line strings or polygons, are treated as data types like strings or numbers, and can be manipulated, queried and processed by a standardized set of spatial functions (Section 3). Likewise, we propose to integrate spatial predicates by means of SPARQL filter functions on this geometry type (Section 3.2). This has the advantage of being expressible in W3C's SPARQL query language without any language extensions. To deeply integrate these concepts into RDF query processing, we consider two approaches, a spatial selection operator and a spatial index, which are both implemented in a native RDF triple store (Section 4) and evaluated with generated and real-world spatial RDF data (Section 5). Finally, we conclude our work and discuss promising future research directions in Section 6.

2. STATE OF THE ART AND FOUNDATIONS

In this section, we review existing approaches to spatial RDF modeling and processing, and give foundations about the underlying technology when needed to understand our approach. In the illustrating examples throughout the paper, we use the namespaces defined in Table 1.

2.1 The Resource Description Framework

The Resource Description Framework (RDF) [10] was standardized by W3C as a key enabler of the Semantic Web to express metadata on the web. Its flexibility and its strength to model relations between data entities lead to a wide adoption in many other domains, such as life sciences or information integration. RDF models resources that are identified by a Uniform Resource Identifier (URI) and described through their relationships. The relationships are identified by URIs as well and connect a resource with another resource or a

Prefix	Namespace URI and Comment
`gas:`	`http://example.org/gas` example for a user-defined ontology
`geo:`	`http://www.w3.org/2003/01/geo/wgs84_pos` W3C Geo [5]
`georss:`	`http://www.georss.org/georss` the GeoRSS Standard [17]
`geordf:`	`http://example.org/geo` namespace of our approach
`gml:`	`http://www.opengis.net/gml` the OpenGIS GML Standard [18]
`osm:`	`http://example.org/osm` the test data used in our evaluation

Table 1: Namespace prefixes used in this paper.

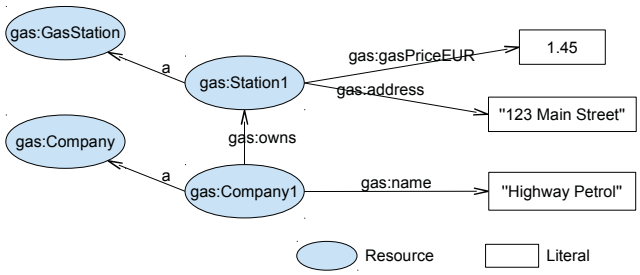


Figure 1: An RDF graph describing a gas station.

literal denoting a certain value. A literal is given as a string and may carry a URI that defines its data type.

Although RDF data is often serialized using XML, the basic data model of RDF consists of *(subject, predicate, object)*-triples, called *statements*: *subject* is the URI of the resource being described, *predicate* denotes the particular relationship through its URI, and *object* is either the URI of another resource, or a literal. All statements together make up a directed labeled graph that represents the resources and their relations. As an example, an RDF graph describing a gas station might look as show in Figure 1.

The example shows how RDF models attributes, e.g. the address of the gas station, and relations between two resources, e.g., the ownership of the gas station by the oil company, equally. This is what makes RDF an ideal choice for data with large amounts of relations between resources.

Modeling Spatial Features in RDF

There are many ways to model spatial features in RDF. Spatial features, as standardized by the OpenGIS Simple Features Specification [7], are complex structures. A point, for instance, consists of two or three coordinates, a linestring or a linear ring comprises many points and a polygon is bounded by a linear ring; it may also have holes, which are linear rings, too. These relationships can be directly modeled as RDF statements. This approach decomposes every spatial feature into several separate statements resulting in a large total amount of statements for spatial data. For feature types consisting of multiple parts, such as Polygon or MultiLineString, every part is modeled as a separate data object having its own URI. This is useful if other data objects need to reference a single part of a spatial feature. E.g., the boundary of a lake may be explicitly referenced to represent a hole in the surrounding meadow. However, the decomposed approach is unfavorable for processing the feature as a whole, e.g., perform calculations, index it, etc., as the feature first needs to be reassembled from its parts.

The W3C Geo Vocabulary [5] (now considered deprecated) was an extreme example of the decomposed approach. It represented points using two separate statements for latitude and longitude. It did not support any other feature types, though. To express the geographic position of the aforementioned gas station, W3C Geo would add the two statements shown in Listing 1 to the RDF graph.

```
gas:Station1234 geo:lat "48.77".
gas:Station1234 geo:lon "9.18".
```

Listing 1: W3C Geo example.

The initial version of Geography Markup Language (GML) had an explicit RDF/XML binding [18]. Later GML versions use an object-property pattern and XML linking which maps directly to RDF. Thus, it is straightforward to transform GML data to an RDF graph. The direct translation results in the decomposed approach with the difference that a list of coordinates is modeled as a single string containing space-separated floating point numbers.

GeoRSS GML [17] uses a GML profile to embed spatial data into RSS feeds, which can be represented in RDF. GeoRSS GML represents spatial features in a class hierarchy which consists of the abstract Geometry class and its subclasses Point, Line, Box and Polygon. GeoRSS GML omits multi-geometries and polygons with holes. Every spatial feature is represented as a resource having its own URI. In the preferred serialization format, the coordinates are supplied as a single string literal containing a space-separated coordinate list. This string literal is connected to the spatial feature via the `gml:pos` predicate. GeoRSS GML models the location of the gas station from the upper examples as shown in Listing 2.

```
gas:Station1234 georss:where gas:Point2094.
gas:Point2094   a            gml:Point.
gas:Point2094   gml:pos      "48.77 9.18".
```

Listing 2: GeoRSS GML example.

As can be seen, GeoRSS GML uses two RDF statements to specify the location: one to specify the class of the feature and one to supply the coordinates. I.e., a more complex feature type, e.g. a polygon, would require two statements as well. Thus, GeoRSS GML can be seen as a *hybrid* approach which does model a spatial feature as a discrete resource, but does not model its parts separately. The feature still carries its own URI, which enables adding further metadata to it, e.g., accuracy or provenance information. It is also reusable, as other resources may reference it. Yet, to process the feature, the two RDF statements still need to be joined.

2.2 The SPARQL Query Language

The SPARQL query language was standardized by W3C [16] to search RDF repositories and is the current W3C recommended query language for RDF data. SPARQL expresses graph pattern matching queries on an RDF graph as conjunctions (and also disjunctions) of triple patterns. A triple pattern may contain variables that are bound to a URI or literal of the RDF graph and that form the result of a select query. The variable bindings can be restricted through filters, which are essentially functions returning a boolean. SPARQL specifies a number of built-in filter functions, such as `regex` or the usual comparison operators (`>`, `!=`, ...). Yet, SPARQL explicitly allows additional filter functions that are identified by a URI. Listing 3 shows an exemplary SPARQL select query returning the address string of all known gas stations that sell gas cheaper than 1.40 EUR.

Expressing Spatial Query Predicates in SPARQL

Kolas [11] proposed to express spatial query predicates using so-called *query premises*, which declare the query parameters as a non-materialized part of the graph pattern in SPARQL. Kolas gives an example similar to the query depicted in Listing 4, which is meant to find all gas stations within distance 1 from point `(48.765 9.175)`.

```
SELECT ?address WHERE {
 ?station a               gas:GasStation.
 ?station gas:address     ?address.
 ?station gas:gasPriceEUR ?price.
 FILTER (?price < 1.40)
}
```

Listing 3: SPARQL example.

```
SELECT ?x WHERE {
 ?x a gas:GasStation. ?x georss:where ?y.
 ?y rcc:part ?p. ?p a gml:Buffer. ?p gml:radius "1".
 ?p gml:bufferGeometry ?g.
 ?g a gml:Point. ?g gml:pos "48.765 9.175".
}
```

Listing 4: SPARQL example using a *query premise* for spatial query predicates [11].

This solution mixes graph patterns addressing the materialized RDF data set and spatial query parameters. The two parts are connected via the `rcc:part` predicate, which expresses the spatial relationship to be evaluated on the fly (Kolas also supports `rcc:connected`). This complicates the query processor and is difficult for humans to read, as different parts of the query have different semantics. To address this problem, Kolas suggests to move the query premise to a separate section of the query and thus proposes the scheme `SELECT ?x PREMISE {...} WHERE {...}`. However, this does no longer comply with the SPARQL specification.

To express query predicates which require parameters, SPARQL provides filter functions. Perry [14] formulates spatial query predicates using a `SPATIAL FILTER` clause followed by calls to spatial comparison functions. This is similar to SPARQL filter functions, but does not match the exact syntax (due to further extensions, this was not Perry's goal). Kolas criticizes that expressing spatial relations as functions breaks the RDF and SPARQL philosophy of modeling all relations between objects as graph edges [11]. Yet, filter functions appear a more natural way to formulate the spatial query predicates, as done by relational spatial databases.

2.3 RDF Data Management

Engines to store and query RDF data, so-called *triple stores*, were long of limited interest outside the Semantic Web community and earlier RDF frameworks, such as Jena [8], focused on functionality over performance. Only in recent years, database research addressed the problem of querying very large RDF datasets efficiently. The challenge of efficient RDF queries originates from the decomposed triple structure and lies in the many join operations that are required to reassemble the data. All state-of-the-art approaches [1, 19, 13, 6, 2] have in common that they first map all URIs and literals to integer IDs. Internally, the RDF statements are stored, indexed, and queried by these IDs, which is a lot faster than processing entire strings. Only to return a query result, the IDs are mapped back to URIs or literals.

Different indexing techniques for efficient RDF queries have been published. SW-Store [1] creates a two-column table in a column-store database for each RDF predicate and stores the *(subject, object)*-pairs of the RDF statements in the respective tables. Virtuoso [6] uses a bitmap index for fast bit vector joins. BitMat [2] stores RDF statements in a compressed bit-matrix structure and processes the joins

by initial pruning followed by a variable-binding-matching algorithm. Hexastore [19] and RDF-3X [13] create an index on all six permutations of the RDF statements, so that every index lookup returns a sorted list and allows fast merge joins.

Spatial RDF Databases

Perry et. al. presented a framework for analysis of spatial and temporal RDF data [14, 15] that was implemented as a set of user-defined functions in Oracle DBMS. It models geographic features in an ontology based on GeoRSS GML [17], but stores them as complex objects in relational tables. Its major disadvantage is the lack of a standardized query language.

Kolas et. al. proposed to use W3C's SPARQL query language [16] for spatial RDF data [12]. They use the GeoRSS RDF vocabulary to model spatial features and formulate spatial queries by means of the query premises, as discussed above. Their implementation [11] builds on top of the Jena Semantic Web Framework [8] and uses a main memory grid file. No performance results were reported.

Recently, Virtuoso implemented support for spatial data[6] using an approach similar to ours. They support spatial joins (which we will address in future work), but are restricted to point data thus lacking arbitrary shapes, e.g. Multipolygons. No performance results were reported on spatial queries.

To the best of our knowledge, no approach exists that natively integrates arbitrary geographic information into the RDF data model and allows efficient processing of spatial operators using a standardized query language. Thus, we propose our approach of *deep integration*: location information is treated like other basic data types (e.g. string), which all benefit from type-safe type-specific functions, query predicates, and efficient processing due to type-specific index support of industrial-strength data management systems.

3. MODELING AND QUERYING SPATIAL LITERALS IN RDF

This section introduces our approach for deep integration of spatial features into RDF processing. For this, we have to extend both the modeling of spatial features in RDF triples, and we have to provide spatial predicates in the standard RDF query language, SPARQL. However, our approach fully complies with the RDF and SPARQL specifications.

3.1 Spatial Literals in RDF

As described in Section 2.1, most existing approaches decompose the spatial information into multiple RDF triples. Our approach to model spatial features in RDF is entirely opposed to the decomposed modeling approach. We represent spatial features as a complex self-contained data type and store them in RDF literals. The literals contain the spatial features expressed in the Well-Known Text (WKT) format, as standardized in the OpenGIS Simple Features Specification [7]. The literals carry a type URI which denotes that the literal is to be processed as a spatial feature rather than as an ordinary string. Listing 5 shows how our approach represents the position of the gas station.

Syntactically, any predicate can connect the literal to any resource. Thus, to process the spatial feature, only the single RDF statement which contains the spatial literal is required, which has two advantages: (1) The spatial feature can be processed independently of the schema. In GeoRSS GML, by contrast, the processing system must know that the `gml:Point` type and the `gml:pos` predicate are related to spatial data to interpret them accordingly. A data set using a different schema, e.g., W3C Geo, will not be supported unless also the spatial RDF classes and predicates are made available to the processing system. (2) The processing system is simplified, as it does not need to reassemble the spatial feature from statements. It may process the RDF data statement by statement.

[6]http://docs.openlinksw.com/virtuoso/rdfsparqlgeospat.html

```
gas:Station1234 gas:locatedAt
           "POINT (48.77 9.18)"^^geordf:geography.
```

Listing 5: Spatially typed literals.

Our approach does not assign a URI to a spatial feature, so that it cannot be directly referenced or augmented by metadata. However, if this is required, one can easily introduce place resources in a specific ontology. These place resources, naturally, are identified by their URI, carry the geometry literal, and may possess further metadata, possibly including a human-readable name. Their URIs may even be used as symbolic coordinates. Still, the place resources would keep all information related to geographic coordinates in the single RDF statement which carries the spatial literal. All further statements related to the place resource are "ordinary" RDF and can be designed in any ontology. Listing 6 shows the gas station example using a place resource.

```
gas:Station1234 gas:place     gas:Place9274.
gas:Place9274   gas:clearName "123 Main street".
gas:Place9274   gas:source    "GPS sensor 2000".
gas:Place9274   gas:locatedAt
           "POINT (48.77 9.18)"^^geordf:geography.
```

Listing 6: A place resource using a typed literal.

3.2 SPARQL Filter Functions

Clearly, it is desirable to express spatial queries on RDF data in SPARQL as well, rather than introducing yet another specialized query language. From this arises a challenge, as SPARQL is designed to search for exact patterns in a (materialized) RDF graph. For spatial joins on some spatial predicates, such as *within*, *covers*, *crosses*, etc., it would be possible to materialize the respective relationships as an explicit RDF statement (which would require the spatial features to carry a URI). These statements could be queried by ordinary SPARQL graph patterns, but would lead to a combinatorial explosion in the total amount of RDF statements. Generally, spatial predicates rarely search for exact relations between data objects but involve calculations which often require parameters. Examples include the maximal distance in a range query or a given constant geometry with which to compare the query result.

Our approach to express spatial query predicates in SPARQL uses filter functions that are identified by a URI. We use the functions of the OpenGIS Simple Features Specification [7], as they are well-established. We defined a URI for each of them. The filter functions act on variables which bind a spatially typed literal, as described in Section 3. Thus, the filter functions complement well our approach to model spatial features in RDF as typed literals. Geometry constants to compare the value of the variable are specified as spatially typed literals, too; they are given in the well-known text

(WKT) format and carry a URI denoting the spatial type. Listing 7 shows a fully standard-compliant SPARQL query to find all gas stations within a given area that specifies the spatial query predicate as a filter function.

```
SELECT ?x WHERE {
 ?x a gas:GasStation.
 ?x geordf:hasGeography ?geo.
 FILTER geo:within (?geo, "POLYGON((48.765 9.175,
   48.775 9.175, 48.775 9.185, ...))"^^geordf:geography)
}
```

Listing 7: Spatial query predicates expressed as SPARQL filter functions.

4. IMPLEMENTATION

We consider two fundamental approaches to implement an RDF triple store with support for spatial query processing: a spatial selection operator and a spatial index. In a full-fledged database system, both may be combined for optimal performance. A spatial selection operator can be implemented on top of an existing triple store. Thus, the query is split into a pure RDF pattern matching query and the spatial query predicate. First, the pattern matching query is evaluated entirely by the triple store. In a second step, the selection operator evaluates the spatial query predicate on every tuple returned by the triple store and discards tuples that do not match. The two steps are performed sequentially (at least conceptually) and there is no way for the selection operator to restrict the query result at an earlier stage. To avoid unnecessary intermediate results, the selection can be pushed down in the query graph. However, this rules out an implementation on top of the triple store, as the query planner of the triple store needs to be modified.

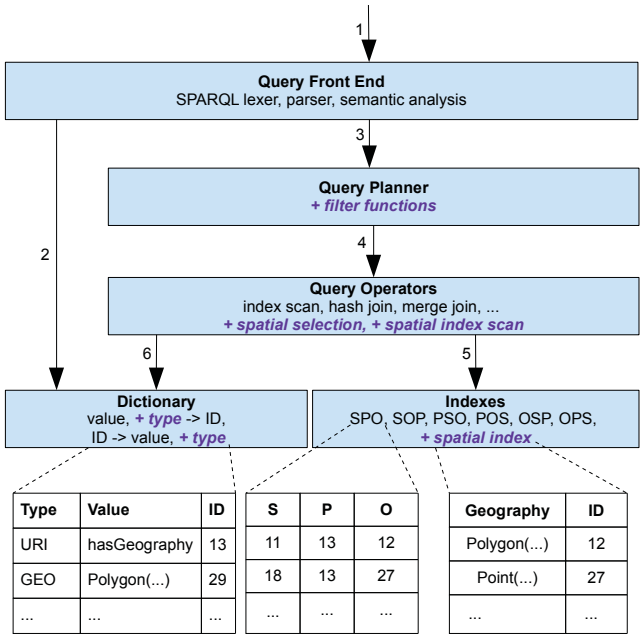


Figure 2: The architecture of the triple store (RDF-3X) and our modifications for deeply integrated support for spatial queries (marked with +)

Also, the selection operator must cope with the internal data representation of the triple store in that case.

A spatial index, on the other hand, may select only spatial features which match the spatial query predicates, right from the start. A spatial index cannot be deployed without a comprehensive deep integration of spatial query processing functionality into a triple store. First of all, the spatial index requires its own database segment to store the spatial features. Whenever spatial data is loaded, the triple store must recognize it and forward it to the spatial index. Moreover, the query planner must be deeply modified to recognize the spatial index and optimize the joins of the spatial features it returns with other intermediate query results. Naturally, the spatial index must also provide the spatial features in a way they can be joined, i.e., it must follow the internal processing mechanisms of the triple store.

We implemented both approaches. We used RDF-3X [13] version 0.3.4 as the starting point for a triple store with deeply integrated spatial query processing. We chose RDF-3X as it outperforms most other triple stores [13, 2] and is able to process very large amounts of data. Moreover, contrarily to other published approaches, RDF-3X is implemented as a complete end-to-end system and is available as open source.[7] However, as all state-of-the-art triple stores share common characteristics (see Section 2.3), our results are applicable for other triple store implementations as well.

4.1 Architecture and Processing Model

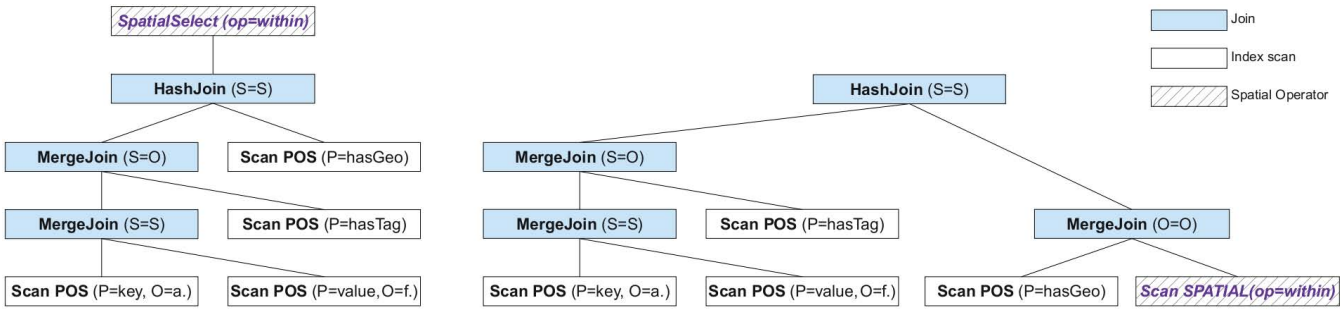
To illustrate our implementation, we first introduce the architecture and the processing model, as shown in Figure 2. The triple store consists of a query front end, a query planner, physical query operators, a dictionary, and indexes. RDF-3X indexes the RDF statements (*S*ubject, *P*redicate, *O*bject) in all six possible permutations: SPO, SOP, PSO, POS, OSP, OPS. Note that it possesses further indexes [13] which we omit here for brevity. The indexes do not list the actual statements but consist of integer IDs, which the dictionary maps to the respective URIs or literals. This saves memory and enables fast join processing.

When the query front end receives a SPARQL query (1), it parses the query and immediately calls the dictionary (2) to resolve all URIs and literals to integer IDs. The logical query graph, which the semantic analysis produces, is unaware of URIs or literals but uses these IDs exclusively. Subsequently, the query planner finds an optimal execution plan (3), as described in [13]. The resulting physical operator graph is instantiated (4) using the query operators. The operators query the indexes (5) and join the scanned values to determine the query result. As all internal processing is done using IDs, the dictionary finally needs to map the query result back to URIs and literals (6).

4.2 Spatial Selection Operator

As our first step towards a triple store supporting spatial queries, we implemented a spatial selection operator that filters the results of a pure RDF pattern matching query. The selection operator supports all comparisons specified in the OpenGIS Simple Features Specification [7]. To implement the selection we used the GEOS C++ library,[8] which GIS systems, such as PostGIS, use as well. Rather than

[7] http://www.mpi-inf.mpg.de/~neumann/rdf3x/
[8] http://trac.osgeo.org/geos/




```
SELECT * WHERE {
  ?tag osm:key "amenity".  ?tag osm:value "fuel".  ?node osm:hasTag ?tag.  ?node geordf:hasGeography ?geo.
  FILTER geo:within (?geo, "POLYGON(( ... ))"^^geordf:geography)
}
```

Figure 3: Physical operator graph using a spatial selection operator (left) and a spatial index (right). The query finds all gas stations within a given region in an RDF graph following the schema of OpenStreetMap.

implementing the selection strictly on top the triple store, we integrated the selection as an additional query operator. For this, we had to modify the query front end, as it neither recognized filter functions nor typed literals; RDF-3X does not implement these parts of the SPARQL specification. We also had to modify the dictionary to store type information with literals. We did *not* modify the query planner. Instead, we simply put the spatial selection operator in front of the execution plan which the planner returns. Finally, we had to modify the result printer to display typed literals correctly with their type URI. The left side of Figure 3 illustrates a operator graph that uses the spatial selection.

RDF-3X does support SPARQL filters as long as they are restricted to identity comparisons (`=` and `!=`). This is because the integer IDs used in internal processing allow no further comparisons of the values. The spatial selection operator needs to perform non-trivial calculations to compare spatial features. There is no way but to look up every single ID from the dictionary and to restore the actual coordinates before the spatial predicate can be evaluated. As a dictionary look-up is a costly operation, we did not consider pushing the spatial selection down in the query graph. Performing the selection in the very end prevents unnecessary look-ups.

4.3 Spatial Index

The spatial selection on top of the pattern matching query enables the triple store to support spatial query predicates fully and is always applicable. Yet, it performs well only on queries with a restrictive graph pattern, which return a rather small number of features to compare. To select only those spatial features that are of interest to the query, a spatial index is needed (even if not every spatial index is applicable on all comparisons, e.g., disjoint). The spatial index maps the features to their dictionary IDs, so that other query operators can process them further. We implemented a spatial index based on the R-Tree index of the libspatialindex C++ library.[9] The spatial index required deep modifications of the triple store. First of all, we had to modify the data import to recognize spatial literals and to insert them into the spatial index. From our work on the spatial selection, the query front end was already prepared to interpret the spatial filter functions. The essential (and most complicated) modification was to make the query planner generate plans which use the spatial index and join the results with other parts of the query.

The fastest join operation of the triple store is the merge join, which requires both operands to be sorted. However, the spatial index partitions the features based on geographic proximity and thus cannot return the feature IDs in a defined order. For this reason, our spatial index scan buffers all feature IDs in memory and sorts them before returning them. Subsequently, a merge join can intersect the feature IDs efficiently with RDF statements that are sorted by the variable which binds the features. The alternative to this sort-merge join would have been a hash join, which buffers all results in a hash table before joining them.

The right side of Figure 3 shows a query graph which uses a spatial index scan (displayed on the very right). The depicted index scan returns exactly those IDs corresponding to the features which match the spatial query predicate. In the example, these are features within the polygon of the FILTER clause. The features are bound to the variable `?geo`, which occurs in the pattern `?node geordf:hasGeography ?geo`. To determine valid bindings for the `?node` variable, the features need to be joined with RDF statements containing the `geordf:hasGeography` predicate. An index scan on one of the six RDF statement indexes selects these statements. The query planner chooses the POS index, so that the predicate in question is selected and the resulting (object, subject)-pairs are sorted by `?geo`. Subsequently, a merge join combines these pairs with the feature IDs, which were sorted by `?geo`, too. The resulting set of Node URIs (`?node`) and spatial features (`?geo`), is still sorted by `?geo` and must be joined with the Tags that indicate a gas station. It takes three index scans and two merge joins to determine the right set of Tags. The Tags are sorted by the `?tag` variable, so a merge join is not possible. Thus, a hash join completes the query result.

4.4 Storing the Features

As described in Section 3.1, our approach expresses spatial features as literals of a complex spatial type. The features are formulated in the Well-Known Text (WKT) format and need

[9] `http://sourceforge.net/projects/spatialindexlib`

to be parsed to evaluate a query predicate on them. This is the case both for the spatial selection and for the spatial index. The index only selects candidates based on their bounding box; a second refine step is required to evaluate the query predicate exactly. In order to accelerate parsing the features, we store them in the dictionary and in the spatial index by means of the Well-Known Binary (WKB) format. WKB consumes significantly less space and is much easier to parse. The downside of this approach is that the features must be converted back to WKT before the final query result can be returned. Thus, whenever the dictionary resolves the ID of a spatial feature, it recognizes its type and parses the WKB string into a geography object. Then it serializes the object to a WKT string and discards the object.

5. EVALUATION

We carried out extensive performance measurements. Our approach combines RDF triple store technology with spatial data processing to a new kind of system, which makes it difficult to compare to other systems. A relational spatial database is likely to perform better, as the database schema may model resources as complete records. This avoids most of the joins which a triple store requires as the price to pay for the schema flexibility (and other advantages) of RDF. Comparisons to other triple stores are not applicable, as they lack arbitrary spatial functionality. Moreover, our implementation builds on RDF-3X, which has been extensively evaluated in literature [13, 2]. Instead, we focused on illustrating the specific characteristics of our implementation and evaluated the effect of our modifications to RDF-3X.

5.1 Test Setup

The test data we used in our evaluation follows the schema of OpenStreetMap, as shown in Figure 4. In OpenStreetMap, every spatial resource is a Node. A Node carries a geographic location and a number of Tags. The Tags describe the Node through their key and value attributes, which are arbitrary strings. We imported data from OpenStreetMap and converted it to RDF with spatially typed literals. Moreover, we generated large amounts of artificial test data, which makes it easier to estimate the number of features which match a spatial query predicate. For this we created Node resources which are located on a grid. The Nodes carry one or more spatial features, which are points on the grid. The Nodes own the features exclusively, i.e., there is a 1:1 or a 1:n ratio between Nodes and points. To evaluate an RDF pattern matching part in queries, we generated Tags which simply carry integer values. To achieve different selectivities of RDF patterns, we tagged *every* Node with key 1, every *second* Node with key 2, every *fourth* Node with 4, etc., up to 1024. This enables us to select a fraction of $\frac{1}{2^n}$ ($n \in \{1..10\}$) of all Nodes. We used two basic grid sizes. The *small grid* contained 1.05 million Nodes which resulted in 11.5 million RDF statements and database files of 856 MB size. The *large grid* contained 104.88 million Nodes, 1.15 billion RDF statements and made up database files of 89.29 GB.

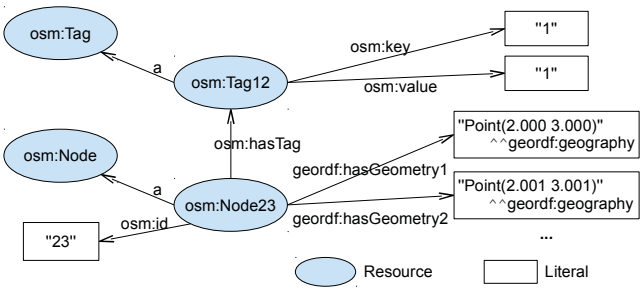


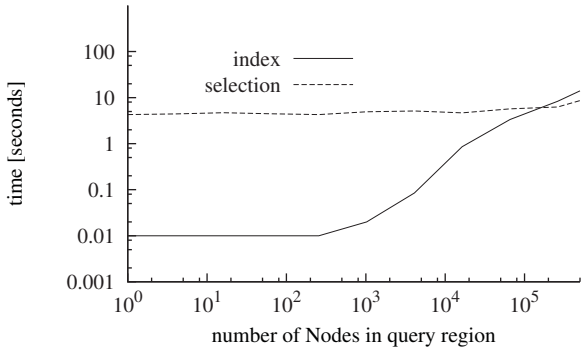
Figure 4: The data model used in our evaluation follows the schema of OpenStreetMap (osm).

We measured the end-to-end execution time counted from the time of query submission to the time including outputting the final results (except for the dictionary test in Section 5.3). We ran all queries on cold and warm caches. For cold caches we dropped the file systems caches using `/bin/sync` and `echo 3 > /proc/sys/vm/drop_caches`. For warm caches we ran the query once before measuring the time. We measured all queries ten times. Our figures report, on a logarithmic scale, the median of ten test runs.

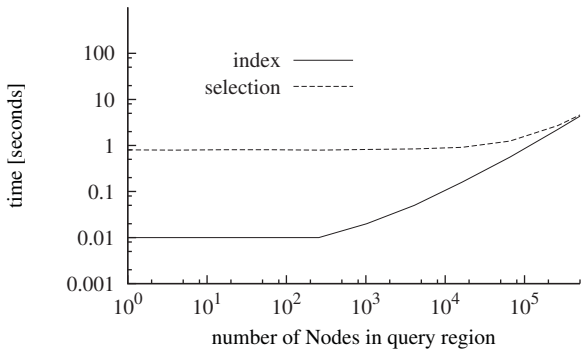
We ran all tests on a Dell Optiplex 755 equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We used two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.31 Linux kernel.

5.2 Spatial Selection vs. Spatial Index

First, we compared the spatial selection operator with the spatial index. As discussed in Section 4, the selection must evaluate the spatial predicate on all features returned by the RDF pattern matching query. The index selects only the relevant features for further processing. We ran the query



(a) small grid, cold caches



(b) small grid, warm caches

Figure 5: Spatial selection vs. spatial index.

```
SELECT * WHERE {
 ?node geordf:hasGeography ?geo.
 FILTER geo:within (?geo,
   "POLYGON(( <<QUERY REGION>> ))"^^geordf:geography)
}
```

Listing 8: Query to compare the spatial selection to the spatial index for different query regions.

of Listing 8 with different query regions (polygons) on the small grid of Nodes. The RDF pattern returns all Nodes in the database, so the query is only selective on the spatial filter. Figure 5 shows the results. It is obvious that the spatial selection performs equally for all query regions, as the pattern matching part always returned all stored features to the selection. For small regions, fewer query results are produced, but the selection already resolved the respective IDs from the dictionary, so that they are always cached. The spatial index is faster for up to about half a million selected Nodes, especially on cold caches. Beyond that point, the costs for joining the intermediate query results dominate the selection. This strongly depends on how many tuples the graph patterns produce. For the large grid, the selection constantly took hundreds of seconds as it had to evaluate all 105 million Nodes. Thus, an advanced query planner should choose between the selection and the spatial index, depending on the specific cardinalities of the query parts.

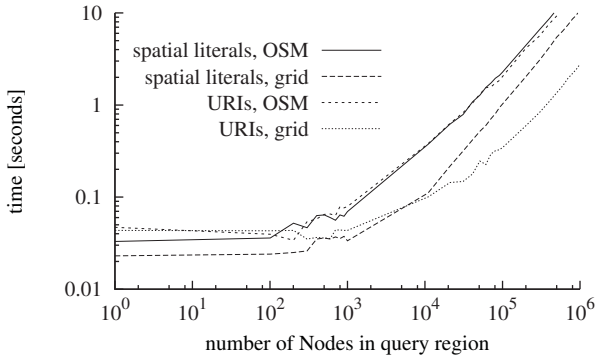
5.3 Dictionary Performance

As outlined in Section 4.4, we modified the dictionary to store spatial features in the Well-Known Binary (WKB) format for faster parsing. The downside is that the geometries must be parsed in all cases, i.e., also to print them as typed literals in the Well-Known Text (WKT) format. We ran a series of tests to determine the parsing overhead. Using the query of Listing 9 we produced different amounts of IDs for the dictionary to resolve to URIs or spatial literals. We used the generated large grid database and a real-world data set from OpenStreetMap (OSM) of similar size. OSM contains more complex features, such as Polygons and LineStrings, in addition to Points. In contrast to all other tests, we did *not* record end-to-end execution times in this test, but measured only the time to resolve the IDs. Figure 6 reports the median of ten runs.

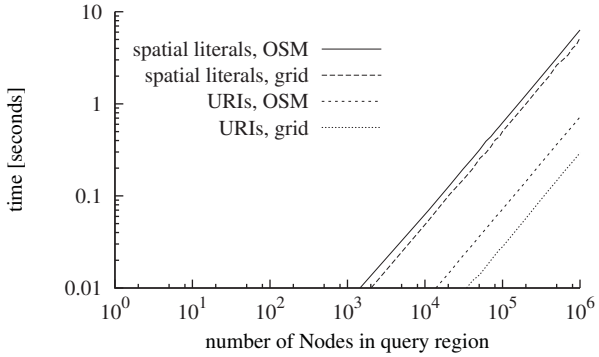
The dictionary requires exactly two page reads to look up any ID. Moreover, the IDs are resolved in ascending order, causing an ideal cache hit ratio. Thus, for small amounts of IDs the lookup time is very small on cold caches and hardly measurable on warm caches. The overhead to parse spatial literals is observable, but only starts to play a role for very large ID sets. For cold caches, the characteristics of the data, e.g., page locality, dominate parsing: the grid URIs perform better than OSM URIs. Also, the more complex OSM literals take negligibly longer to parse. [2] observed the performance impact of resolving very large query results. Our measurements confirm this. Yet, the additional impact

```
SELECT <<?node | ?geo>> WHERE {
 ?node geordf:hasGeography ?geo.
} LIMIT <<NUMBER OF DICTIONARY ENTRIES>>
```

Listing 9: Query to measure the dictionary performance.



(a) large grid and OpenStreetMap (OSM), cold caches



(b) large grid and OpenStreetMap (OSM), warm caches

Figure 6: Dictionary performance.

of parsing WKB literals is only a tiny fraction of the overall execution time as shown in our further evaluation.

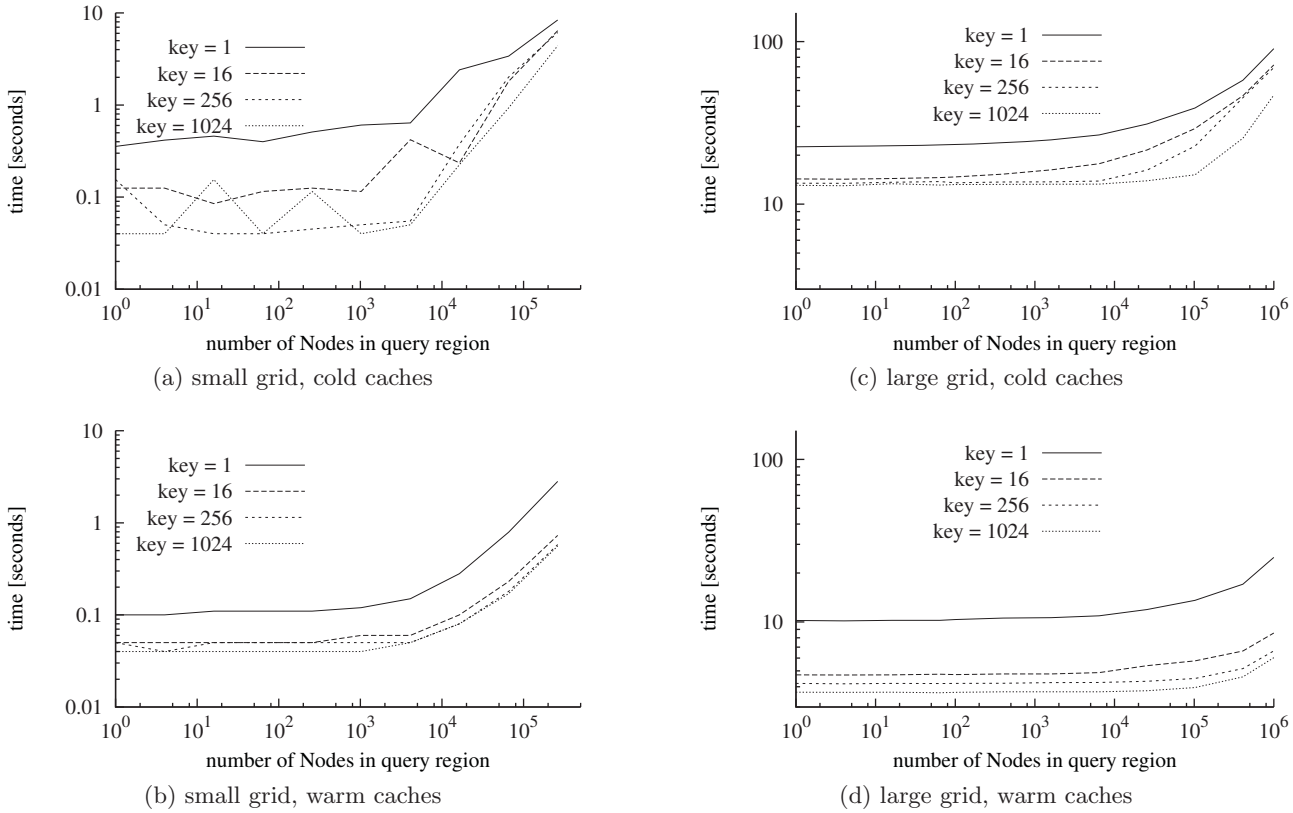
5.4 Different Selectivities

To get a broader understanding of the general performance, we ran a series of queries with different selectivities in the RDF pattern matching part and in the spatial query predicate. We queried Nodes with a particular Tag key to influence the pattern matching part, as every Tag key occurs at a different fraction of the Nodes (see Section 5.1). We influenced the spatial selectivity through different query regions and evaluated the spatial query predicate on the spatial index. Listing 10 shows the test query. Note that the query resolves and prints all bound variables. Figure 7 shows the results, both for the small and the large grid database.

Much more than selectivity does the database size make an impact. Even though RDF-3X is good at discarding unnecessary intermediate tuples early [13, 2], the larger database inevitably causes more of them. For up to 10^4 resp. 10^5 Nodes, the results are nearly independent of the query region,

```
SELECT * WHERE {
 ?tag  osm:hasKey          "<<TAG KEY>>".
 ?node osm:hasTag          ?tag.
 ?node geordf:hasGeography ?geo.
 FILTER geo:within (?geo,
   "POLYGON(( <<QUERY REGION>> ))"^^geordf:geography)
}
```

Listing 10: Query to compare performance with different selectivities.



(a) small grid, cold caches

(c) large grid, cold caches

(b) small grid, warm caches

(d) large grid, warm caches

Figure 7: Different selectivities in the RDF pattern matching part and in the spatial predicate of a query.

which indicates that the query plan did not use the spatial index optimally. Joining the spatial index with the rest of the query earlier would avoid intermediate tuples. In our preliminary implementation the query planner does not use spatial statistics, which leaves potential for future work. Further, the less selective Tag keys create higher load, but the overhead is much smaller than the difference in selectivity. Generally, we think the shown performance is very good.

5.5 Multiple Spatial Features per Resource

In a final series of tests we address the situation of multiple spatial features per resource. It depends on the data model whether this can occur. Models such as OpenStreetMap or those using the place resources discussed in Section 3.1 contain only 1:1 relationships between resources and features, i.e., every feature belongs to exactly one resource. Other models may contain 1:n relationships, e.g., to model both the center point and the boundary of a building. Our spatial index contains all features of the entire RDF graph in a single index structure. Thus, it returns all features that match a spatial query predicate, regardless of the resource it belongs to or the predicate connecting it to the resource. I.e. both the center point and the boundary of the aforementioned building are returned if they match the spatial predicate—even if the query only needs the center point.

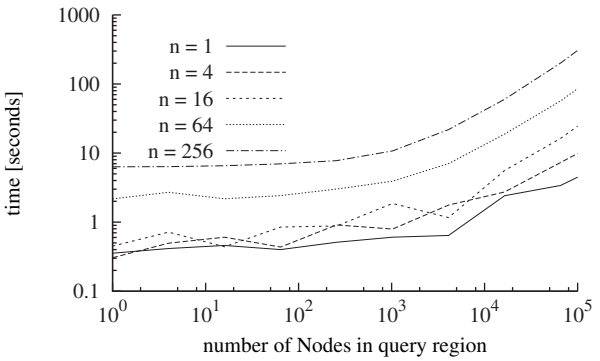
To measure this effect, we generated different variants of the *small* grid database with a different number n of points per Node, resulting in database sizes of up to 33.9 GB. No two points of the database were identical, i.e., all points were represented by a different dictionary ID. Also, all points of the same resource used different predicates (`osm:hasGeography1`, `osm:hasGeography2`, ...). Listing 11 shows the corresponding test query; to include further pattern matching parts, it selects Nodes carrying Tag 1. Figure 8 shows the results. Note that the X axis counts Nodes, not features. Especially for cold caches, a higher amount of features per resource clearly does increase the costs in our indexing approach. However, even though more features per resource take longer, the effect is moderate for queries which select a small to medium amount of features.

```
SELECT * WHERE {
 ?tag  osm:hasKey           "1".
 ?node osm:hasTag           ?tag.
 ?node geordf:hasGeography1 ?geo.
 FILTER geo:within (?geo,
   "POLYGON(( <<QUERY REGION>> ))"^^geordf:geography)
}
```

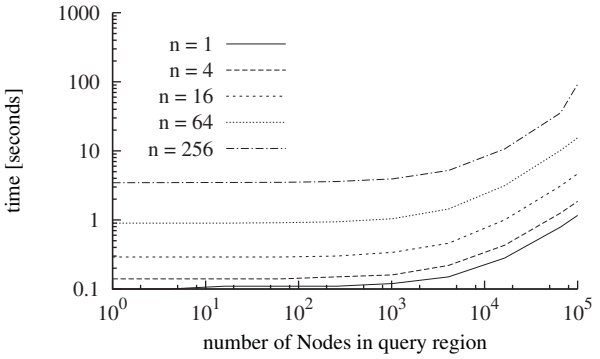
Listing 11: Test query to select one out of multiple features per resource.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we discuss how spatial data processing can be natively integrated into RDF modeling and querying. Our solution models spatial features as typed complex literals, and defines spatial predicates as filter functions in SPARQL. Furthermore, we discuss the deep integration of these concepts into RDF triple stores, and present an implementation of a triple store with spatial functionality. Our evaluation shows that some of the modifications for our deep integration



(a) small grid, cold caches



(b) small grid, warm caches

Figure 8: Multiple spatial features per resource: Every Node possesses n different features.

approach do create some extra overhead for queries selecting very large amounts of spatial features, but we observe excellent performance for most common spatial query types.

Our implementation fully supports the spatial query predicates of the OpenGIS Simple Features Specification [7] by means of SPARQL filter functions. Nevertheless, our work still has promising future research directions. To optimize queries over huge spatial and non-spatial data sets, statistics are necessary on how the spatial data is distributed. This way, the query planner may compute the most efficient query plan and, e.g., decide whether to use the spatial index or a spatial selection. In addition, our implemented filter functions currently compare a spatial feature with a constant value. This is sufficient for many application scenarios. Evaluating a spatial relationship between two variables, i.e., a spatial join, is a challenge we will address in the future. Finally, our work is about storing and querying static RDF data with rare updates, as commonly accepted in RDF scenarios. Another future research direction would be to cope with changes and updates in the location data, and their effects on indexing and data processing.

Acknowledgments

We would like to thank Bastian Reitschuster, Tim Waizenegger, Oliver Schiller and Nazario Cipriani for greatly supporting our work.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. In *WWW*, 2010.

[3] S. Auer, J. Lehmann, and S. Hellmann. LinkedGeoData: Adding a Spatial Dimension to the Web of Data. *The Semantic Web-ISWC 2009*, pages 731–746, 2009.

[4] M. Bauer, C. Becker, and K. Rothermel. Location models from the perspective of context-aware applications and mobile ad hoc networks. *Personal and Ubiquitous Computing*, 6(5/6):322–328, 2002.

[5] D. Brickley. Basic Geo (WGS84 lat/long) Vocabulary. W3C Semantic Web Interest Group, 2003. `http://www.w3.org/2003/01/geo/`.

[6] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. *Networked Knowledge-Networked Media*, pages 7–24, 2009.

[7] J. R. Herring. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture. Candidate, Open Geospatial Consortium, Inc., 2006.

[8] Jena: a Semantic Web framework for Java. http://jena.sourceforge.net/.

[9] W. Kammersell and M. Dean. Conceptual search: Incorporating geospatial data into semantic queries. In *Terra Cognita – Directions to the Geospatial Semantic Web*, 2006.

[10] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. Recommendation, W3C, 2004.

[11] D. Kolas. Supporting Spatial Semantics with SPARQL. *Transactions in GIS*, 12(s1):5–18, 2008.

[12] D. Kolas and T. Self. Spatially augmented knowledge-base. In *ISWC+ASW*, 2007.

[13] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[14] M. Perry. *A framework to support spatial, temporal and thematic analytics over semantic web data*. PhD thesis, Wright State University, 2008.

[15] M. Perry, F. Hakimpour, and A. Sheth. Analyzing theme, space, and time: an ontology-based approach. In *ACM GIS*, 2006.

[16] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Recommendation, W3C, 2008.

[17] R. Singh, A. Turner, M. Maron, and A. Doyle. GeoRSS: Geographically encoded objects for RSS feeds, 2009. `http://georss.org/gml`.

[18] The Open Geospatial Consortium. OpenGIS Geography Markup Language (GML) Encoding Standard - Version 1.0.0, 2000.

[19] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. In *VLDB*, 2008.