2007

# What, Where and When: Supporting Semantic, Spatial and Temporal Queries in a DBMS

Matthew Perry
*Wright State University - Main Campus*

Amit P. Sheth
*Wright State University - Main Campus*, amit@sc.edu

Farshad Hakimpour

Prateek Jain

Kno.e.sis Center Technical Report
Department of Computer Science & Engineering
Wright State University

---

Technical Report: KNOESIS-TR-2007-01

# What, Where and When: Supporting Semantic, Spatial and Temporal Queries in a DBMS

Matthew Perry, Amit P. Sheth, Farshad Hakimpour, Prateek Jain

April 22, 2007

---

# What, Where and When: Supporting Semantic, Spatial and Temporal Queries in a DBMS*

Matthew Perry[1], Amit P. Sheth[1], Farshad Hakimpour[2], Prateek Jain[2]

[1] kno.e.sis Center, Department of Computer Science and Engineering, Wright State University, Dayton, OH, USA
[2] LSDIS Lab, Department of Computer Science, University of Georgia, Athens, GA, USA
{perry.66, amit.sheth}@wright.edu, {fhakimpour, prateek}@uga.edu

**Abstract.** Spatial and temporal data are critical components in many applications. This is especially true in analytical domains such as national security and criminal investigation. The outcome of the analytical process in these applications often hinges on uncovering and analyzing complex relationships between disparate people, places and events. Fundamentally new query operators based on the graph structure of Semantic Web data models, such as semantic associations, are proving useful in these applications. However, these analysis mechanisms are primarily intended for thematic relationships. We describe a framework built around the RDF metadata model for analysis of thematic, spatial and temporal relationships between named entities and describe an efficient implementation in Oracle DBMS. Additionally, we demonstrate the scalability of our approach with a performance study using a synthetic dataset from the national security domain.

**Keywords:** Ontology, Semantic Analytics, RDF, SPARQL

## 1 Introduction

Analytical applications are increasingly exploiting complex relationships between named entities as a powerful mechanism to aid in the analysis process. Such "connecting the dots" applications are common in many domains such as national security, drug discovery and medical informatics. Semantic Web technologies are well suited for this type of analysis. First of all, it is often necessary that the analysis process span across multiple heterogeneous data sources. Ontologies and semantic metadata standards help facilitate aggregation and integration of this content. Additionally, standard models for metadata representation on the web, e.g., Resource Description Framework (RDF) [1], model relationships as first-class objects making it very natural to query and analyze entities based on their relationships. Consequently, novel relationship-based query types, such as semantic association [2] and subgraph discovery [3], have been proposed for RDF graphs. These query types have been successfully used in a variety of settings, for example conflict of interest detection [4], patent searching [5] and metabolic pathway discovery [6]. Hereafter, we use the

* This work is partially funded by NSF-ITRIDM Award #0325464 & #0714441 entitled "SemDIS: Discovering Complex Relationships in the Semantic Web."

term *semantic analytics* to refer to the process of searching, analyzing and visualizing named relationships between known entities.

So far, all semantic analytics tools are primarily intended for the analysis of thematic relationships. However, spatial and temporal data play crucial roles in many of these analytical domains, and we argue that our semantic analytics toolbox must be extended so that we can also search and analyze spatial and temporal relationships. We feel there are certain classes of problems for which a native RDF graph is the most appropriate representation, thus the ability to handle spatial and temporal data in this representation is necessary. Furthermore, as discussed in [7], modeling spatial, temporal and thematic data using ontologies and RDF results in higher levels of flexibility and extensibility when compared to traditional approaches.

Spatial and temporal data bring many unique challenges to semantic analytics applications. Thematic relationships can be explicitly stated in the RDF graph, but some spatial and temporal relationships (e.g., quantitative relationships like distance) are implicit and only evident after additional computation. Also, it may not be desirable to explicitly record qualitative spatial and temporal relationships because, to ensure completeness, the number of such statements could be quite large. RDFS inferencing rules [8] are also affected as the temporal properties of asserted statements will have implications on the temporal properties of the corresponding inferred statements.

To paint a clearer picture of our needs, consider the following scenario which illustrates the importance of the semantic, spatial and temporal dimensions in analytical applications. Suppose an intelligence analyst is assigned the task of monitoring the health of soldiers in order to detect possible exposure to a chemical or biological agent which may imply a biochemical attack. In this case, the analyst would most likely be interested in relationships between soldiers, chemical or biological agents, enemy groups in the region, their known activities (reports) and capabilities. The analyst might search for relationships connecting a sick soldier to potential chemical or biological agents by matching the soldier's symptoms with known reactions to chemical or biological agents. In addition, the analyst could further determine the likelihood of a particular chemical agent by querying for associations between the agent and enemy groups in the knowledgebase. For example, a member of the group may have worked at a facility which was reported to have produced the chemical. It is doubtful that such an analysis could produce definitive evidence of a biochemical attack, but incorporating spatial and temporal relationships could help in this regard. For instance, the analyst may want to limit the results to soldiers and enemies in close spatial proximity (e.g., find all soldiers with symptoms indicative of exposure to *chemical X* which fought in battles within 2 miles of sightings of any members of *enemy group Y*).

To realize the types of spatial and temporal relationship analysis outlined in the previous scenario, we identify four basic spatial and temporal query operators. The operators are built upon SPARQL-like graph patterns [9]. For example, we may pose the following query for the search outlined previously:

```
select a from table (spatial_eval ('(?a has_symptom ?b)
  (Chemical_X induces ?b)(?a fought_in ?c)', ?c,
  '(?d member_of Enemy_Group_Y)(?d spotted_at ?e)', ?e,
  'geo_distance(distance=2 units=mile)'));
```

With this query, we are using the *spatial_eval* operator to specify a relationship between a soldier, a chemical agent and a battle location and a relationship between members of an enemy organization and their known locations. We are then limiting the results based on the spatial proximity of the battles and enemy sightings.

This paper focuses on providing a framework to support spatial and temporal analysis of RDF data. We address problems of both data storage and operator design and implementation. Specifically, the contributions of this paper are:

- A storage and indexing scheme for spatial and temporal RDF data
- An efficient treatment of temporal RDFS inferencing
- The definition of four spatial and temporal query operators
- An efficient implementation of the defined query operators in Oracle DBMS
- A performance study using a large, synthetically-generated RDF dataset

The remainder of the paper is organized as follows. Section 2 discusses background information and related work regarding data modeling and querying. Section 3 introduces the set of spatial and temporal query operators. Section 4 describes the implementation of this framework in Oracle DBMS. An experimental evaluation of this implementation follows in Section 5, and Section 6 gives conclusions.

## 2 Background and Related Work

In this section, we discuss background information on data modeling and related work in querying semantic data models. Specifically, we cover background information on the RDF data model, temporal RDF graphs and how we model spatial and temporal data using ontologies and temporal RDF graphs. This is followed by a discussion of approaches to querying RDF data.

**RDF and Ontologies.** RDF has been adopted by the W3C as a standard for representing metadata on the Web. Resources in RDF are identified by Uniform Resource Identifiers (URIs) that provide globally-unique and resolvable identifiers for entities on the Web, yielding a decentralized information space. These resources are described through participation in relationships. Relationships in RDF are called *Properties* and are binary relationships connecting resources to other resources or resources to *Literals*, i.e., literal values such as Strings or Numbers. These binary relationships are encoded as triples of the form (*Subject*, *Property*, *Object*), which denotes that a resource – the *Subject* – has a *Property* whose value is the *Object*. These triples are referred to as *Statements*. RDF also allows for anonymous nodes called *Blank Nodes* which can be used as the *Subject* or *Object* of a statement. We call a set of triples an *RDF graph*, as RDF data can be represented as a directed, labeled graph with typed edges and nodes. In this model, a directed edge labeled with the *Property* name connects the *Subject* to the *Object*.

RDF Schema (RDFS) provides a standard vocabulary for describing the classes and relationships used in RDF statements and consequently provides the capability to define ontologies. An ontology is classically defined as a specification of a conceptualization [10]. Ontologies serve to formally specify the semantics of RDF

data so that a common interpretation of the data can be shared across multiple applications. RDFS allows us to define hierarchies of class and property types, and it allows us to define the domain and range of property types.

**Temporal RDF Graphs.** In order to analyze the temporal properties of relationships in RDF graphs, we need a way to record the temporal properties of the statements in those graphs, and we must account for the effects of those temporal properties on RDFS inferencing rules. For this purpose, we adopt temporal RDF graphs defined in [11]. Temporal RDF graphs model absolute time and are defined as follows. Given a set of discrete, linearly ordered time points $T$, a temporal triple is an RDF triple with a temporal label $t \in T$. The notation $(s, p, o) : [t]$ is used to denote a temporal triple. The expression $(s, p, o) : [t_1, t_2]$ is a notation for $\{(s, p, o) : [t] \mid t_1 \leq t \leq t_2\}$. A statement's temporal label represents its valid time. A temporal RDF graph is a set of temporal triples. For example, consider a soldier assigned to the 1st Armored Division from April 3, 1942, until June 14, 1943, and then assigned to the 3rd Armored Division from June 15, 1943, until October 18, 1943. The relationship connecting the soldier to the 1st Armored Division would be labeled with the closed interval [04:03:1942, 06:14:1943] and the relationship connecting the soldier to the 3rd Armored Division would be labeled with the closed interval [06:15:1943, 10:18:1943]. Any temporal ontology which defines a vocabulary of time units can be used to precisely specify the start and end points of time intervals.

As discussed in [11], we must account for temporal inferencing in temporal RDF graphs. A set of entailment rules are defined for RDF and RDFS [12]. These rules essentially specify that an additional triple can be added to the RDF graph if the graph contains triples of a specific pattern. Such rules describe, for example, the transitivity of the *rdfs:subClassOf* property. To incorporate temporal inferencing we must use a basic arithmetic of intervals to derive the temporal label for the inferred statements. For example, interval intersection would be needed for *rdfs:subClassOf* (e.g., $(x,$ *rdfs:subClassOf*, $y) : [1, 4] \wedge (y,$ *rdfs:subClassOf*, $z) : [3, 4] \rightarrow (x,$ *rdfs:subClassOf*, $z) : [3, 4]$).

**Modeling Theme, Space and Time.** In a previous work [7], we presented an approach for modeling thematic entities and their spatial and temporal properties using ontologies and temporal RDF graphs. We showed how a small upper-level ontology can be used to define the basic classes of and relationships between the thematic and spatial dimensions. Temporal RDF graphs were used to incorporate the temporal dimension into this model. Deeper domain ontologies can be integrated with this upper-level ontology through *rdfs:subClassOf* and *rdfs:subPropertyOf* statements. This approach has the benefit of greater flexibility because we use relationships in the thematic domain to indirectly associate thematic entities with spatial objects, and it allows the direct application of existing thematic analytics techniques.

**Related Work.** Many RDF query languages have been proposed in the literature. These include SQL-like languages (e.g., SPARQL [9]), functional languages (e.g., RQL [13]), rule-based languages (e.g., TRIPLE [14]) and graph traversal languages (e.g., RxPath [15]). Efficient implementations of these languages for persistent RDF

data usually involve translation into a SQL query against an underlying RDBMS representation of the RDF data (e.g., Jena2 [16], RDFSuite [17]). As an alternative to defining a new query language, Oracle defines a SQL table function for querying RDF [18]. This allows for querying an ontology directly in SQL. Consequently, it facilitates easy integration with other SQL queries against traditional relational data and saves the overhead of translating data from SQL to the RDF query language data format. We follow this approach and introduce new table functions which enable spatial and temporal querying of RDF data.

Work is somewhat limited with regards to incorporating spatial and temporal relationships into queries over Semantic Web data. Examples of querying geospatial RDF data are mostly seen in web applications and semantic geospatial web services [19, 20] in the spirit of the Semantic Geospatial Web [21]. In general, query processing proceeds by translating RDF representations of spatial features into geometric representations on the fly and then performing spatial calculations, and the focus is more on interoperability than efficient query processing. The SPIRIT spatial search engine [22] combines an ontology describing the geospatial domain with the searching and indexing capability of Oracle Spatial for the purposes of searching documents based on the spatial features associated with named places mentioned in the document. However, the types of ontology-based searching operations used in SPIRIT are not intended for general purpose querying of ontological and spatial relationships as are the ones discussed in this paper. Querying for temporal data in RDF graphs is less complicated as RDF supports typed literals such as *xsd:date*, and corresponding query languages support filtering results based on literal values. However, this is far from supporting the temporally-reified RDF graphs discussed in this paper. In addition to formally defining temporal RDF graphs, Gutierrez et al. sketched a query language for these graphs, but no implementation issues were discussed [11]. Also, to the best of our knowledge, this paper is the first to investigate implementation of RDFS inferencing which incorporates the concept of valid time for RDF statements.

## 3 Query Operators

In this section, we introduce a set of spatial and temporal query operators for searching and analyzing spatial and temporal relationships between named entities in temporal RDF graphs. SPARQL-like graph patterns form the basis for these operators. We provide a means to extract spatial and temporal properties for graph pattern instances, and capabilities are provided to filter graph pattern instances based on the extracted spatial and temporal properties. The operators are implemented as SQL table functions. Table functions produce a set of rows as output which can be queried. They are used in SQL queries in the same manner as a database table name. For example, we may have the query `select x, y from table (table_func (...)) order by x`.

**Graph Patterns.** Intuitively, a *graph pattern* is a set of RDF triples where the subjects, properties and/or objects may be replaced with variables. In general, a graph

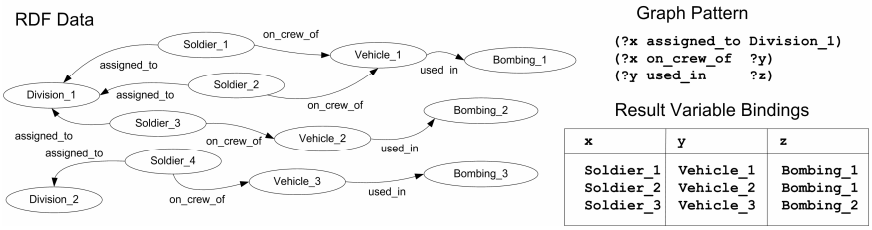


**Fig. 1.** Example graph pattern with resulting variable bindings.

pattern query against an RDF graph *G* proceeds by finding a set of mappings between the variables in the graph pattern and terms (URIs, Blank Nodes and Literals) in *G* such that substituting the mapped terms into the graph pattern results in a set of triples actually present in *G*. We refer to the set of triples resulting from a substitution as a *graph pattern instance*, and the result of a graph pattern query on a given RDF graph *G* is the set of variable bindings for all matching graph pattern instances in *G*. Fig. 1 illustrates these concepts for an example graph pattern query.

**Spatial Query Operators.** We define two spatial query operators for RDF graphs containing geospatial data. The following descriptions assume the existence of a class *Geometry* in the ontology which models spatial objects, and we use the term *spatial feature* to refer to an *SDO_GEOMETRY* object that would be stored in Oracle Spatial. The basic idea of these operators is that we use thematic relationships in the ontology to connect a non-spatial entity with a *Geometry* in a meaningful way. For example, we may connect a soldier to the locations of battles in which he fought or alternatively we may connect the solider to the locations at which he trained. The key idea here is that we are utilizing indirect relationships to connect thematic entities with spatial objects in a variety of meaningful ways instead of enforcing a one-to-one mapping between thematic entities and spatial objects.

The first spatial operator, *spatial_extent*, is intended to retrieve the spatial feature of the *Geometry* connected to a thematic entity and optionally filter the results based on the properties of the spatial feature. The signature for the corresponding table function is shown below:

```
spatial_extent (graphPattern VARCHAR, spatialVar
  VARCHAR, ontology RDFMODELS, <geom SDO_GEOMETRY>,
  <spatialRelation VARCHAR>)
returns AnyDataSet;
```

The *graphPattern* parameter specifies the relationship between a non-spatial entity and a *Geometry*, for example (*Soldier*, *fought_in*, *Battle*) (*Battle*, *located_at*, *Geometry*). The *spatialVar* parameter identifies the variable in the graph pattern that corresponds to a *Geometry*, and *ontology* determines the ontology to search against. This function returns a table with rows containing columns for each variable in the graph pattern and a column for the spatial features. Each row contains the URI bound to each variable and a spatial feature corresponding to the *Geometry* bound to *spatialVar*. Two optional parameters, a spatial feature and a spatial relationship, can be used to filter the graph pattern instances. In this case, the table would only contain those graph pattern instances whose associated spatial features satisfy the specified spatial relation with the input spatial feature. We support the following spatial

relationships: *touch*, *overlap*, *equal*, *inside*, *covered by*, *contains*, *covers*, *any interact*, and *within distance*.

The second spatial operator, *spatial_eval*, acts as a spatial join between graph pattern instances. It is intended to allow for searching thematic entities based on their spatial relationships. The signature for the corresponding table function is shown below:

```
spatial_eval (graphPattern VARCHAR, spatialVar VARCHAR,
  graphPattern2 VARCHAR, spatialVar2 VARCHAR,
  spatialRelation VARCHAR, ontology RDFModels)
return AnyDataSet;
```

*graphPattern* and *spatialVar* specify the left hand side of the join operation, while *graphPattern2* and *spatialVar2* specify the right hand side. *spatialRelation* identifies the spatial join condition. This function returns a table containing a column for each variable in *graphPattern* and *graphPattern2* and a column for each associated spatial feature (*sf*1 and *sf*2). For each row in the resulting table, *sf*1 *spatialRelation sf*2 evaluates to *true*.

**Temporal Query Operators.** We define two temporal query operators for temporal RDF graphs. The basic idea behind the operators is that we compute a temporal interval for a graph pattern instance based on the temporal properties of the triples making up the graph pattern instance. We provide operators to compute these intervals, filter graph patterns based on these intervals and join graph pattern instances based on the temporal relationships between their intervals.

The first temporal operator, *temporal_extent*, is used to compute the temporal interval for a graph pattern instance and optionally filter the results based on the computed temporal interval. We support two basic intervals for a graph pattern instance: the interval during which the entire graph pattern instance is valid (*INTERSECT*) and the interval during which any part of the graph pattern is valid (*RANGE*). The signature for the corresponding table function is shown below.

```
temporal_extent (graphPattern VARCHAR, intervalType
  VARCHAR, ontology RDFModels, <start DATE>,
  <end DATE>, <temporalRel VARCHAR>)
return AnyDataSet;
```

This function takes three parameters as input, specifically a graph pattern, a String value specifying the interval type (*INTERSECT* or *RANGE*), and a parameter specifying the temporally-reified ontology to search against. The table returned contains a column for each variable in the graph pattern and two *DATE* columns which specify the start and end of the time interval computed for the graph pattern instance. Three optional parameters, two *DATE* values to identify the boundaries of a time interval and a temporal relationship, can be used to filter the found graph pattern instances. In this case, assuming the *DATE* columns in the returned table are named *stDate* and *endDate*, each row in the result satisfies the condition [*stDate*, *endDate*] *temporlRel* [*start*, *end*]. We currently support seven temporal relationships: *before*, *after*, *during*, *overlap*, *during_inv*, *overlap_inv*, *any interact*.

The second temporal operator, *temporal_eval*, acts as a temporal join operator for graph pattern instances. The corresponding table function has the following signature:

```
temporal_eval (graphPattern VARCHAR, intervalType
  VARCHAR, graphPattern2 VARCHAR, intervalType2
  VARCHAR, temporalRel VARCHAR, ontology RDFModels)
```

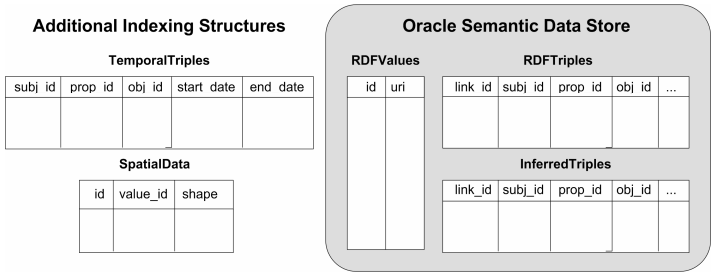


**Fig. 2**. Storage structures for RDF data. Existing tables of Oracle Semantic Data Store are shown on the right, and our additional tables for efficiently searching spatial and temporal data are shown on the left.

```
  return AnyDataSet;
```

*graphPattern* and *intervalType* specify the left hand side of the join operation, while *graphPattern2* and *intervalType2* specify the right hand side. *temporalRel* identifies the join condition. This function returns a table containing a column for each variable in *graphPattern* and *graphPattern2* and four *DATE* columns (*start1*, *end1*, *start2, end2*) to indicate the time interval for each found graph pattern instance. For each row in the resulting table [*start1*, *end1*] *temporalRel* [*start2*, *end2*] evaluates to *true*.

## 4 Implementation in Oracle

In this section, we describe the implementation of our spatial and temporal RDF query operators in Oracle DBMS. The implementation builds upon Oracle's existing support for RDF storage and inferencing and support for spatial object types and indexes. We create SQL table functions for each of the previously discussed query operators. Additional structures are created to allow for spatial and temporal indexing of the RDF data for efficient execution of the table functions.

**Existing Oracle Technologies.** Oracle's Semantic Data Store [23] provides the capabilities to store, infer, and query semantic data, which can be plain RDF descriptions and RDFS based ontologies. To store RDF data, users create a model (ontology) to hold RDF triples. The triples are stored after normalization in two tables: an *RDFValues* table which stores RDF terms and a numeric id and an *RDFTriples* table which stores the ids of the subject, predicate and object of each statement. Users can optionally derive a set of inferred triples based on user-defined rules and/or RDFS semantics. These triples are materialized by creating a rules index and stored in a separate *InferredTriples* table. These storage structures are illustrated in Fig. 2. A SQL table function is provided that allows issuing graph pattern queries against both asserted and inferred RDF statements.

Oracle Spatial [24] provides facilities to store, query, and index spatial features. It supports the object-relational model for representing spatial geometries. A native spatial data type, *SDO_GEOMETRY*, is defined for storing vector data. Database tables can contain one or more *SDO_GEOMETRY* columns. Oracle Spatial supports

R-Tree and Quad-Tree indexes on *SDO_GEOMETRY* columns, and provides a variety of procedures, functions and operators for performing spatial analysis operations.

**Data Representation.** Our Framework supports spatial and temporal data serialized in RDF using an RDFS ontology discussed in [25]. This ontology models the concept of a *Geometry* Class and allows for recording coordinate system information and representing points, lines, and polygons. This model complies with the OGC simple feature specification [26]. Using this representation, spatial features are stored as instances of *Geometry* and are uniquely identified by their URI. Temporal labels are associated with statements using RDF reification, as suggested in [11]. Reification allows us to make statements about RDF statements. Our framework supports time interval values serialized as instances of the Class *Interval* from this ontology. A property type, *temporal*, is defined to assert that a statement has a valid time which is represented as an *Interval* instance.

### 4.1 Indexing Approach

In order to ensure efficient execution of graph pattern queries involving spatial and temporal predicates, we must provide a means to index portions of the RDF graph based on spatial and temporal values. Basically, this is done by building a table mapping *Geometry* instance URIs to their *SDO_GEOMETRY* representation and by building a modified *RDFTriples* table which also stores the temporal intervals associated with the triple. In order to build these indexes, users first load the set of asserted RDF statements into Oracle Semantic Data Store and build an RDFS rules index. After this, both the spatial and temporal indexes can be constructed. This indexing scheme does not support incremental maintenance. However, RDFS rules indexes do not support incremental maintenance either, so this indexing approach is in keeping with the overall scheme of Oracle Semantic Data Store.

**Spatial Indexing Scheme.** We provide the procedure *build_geo_index* (*ontology*, *spatial_table_name*) to construct a spatial index for a given ontology. The parameter *ontology* identifies the ontology model stored in Oracle, and *spatial_table_name* allows the user to name the spatial indexing table created. This procedure first creates the table *spatial_table_name* (*id NUMBER PRIMARY KEY*, *value_id NUMBER*, *shape SDO_GEOMETRY*) for storing spatial features corresponding to instances of the class *Geometry* in the ontology. *id* is a systematically generated key for each geometry; *value_id* is the id given to the URI of the *Geometry* instance in Oracle's *RDFValues* table; and shape stores the *SDO_GEOMETRY* representation of the *Geometry* instance (see Fig. 2). This table is filled by querying the ontology for each *Geometry* instance, iterating through the results and creating and inserting *SDO_GEOMETRY* objects into the spatial indexing table. Finally, to enable efficient searching with spatial predicates on this table, an R-Tree index is created on the *shape* column.

**Temporal Indexing Scheme.** Our temporal indexing scheme is a bit more complicated, as it must account for temporal labels on statements inferred through RDFS semantics. However, we only need to handle a subset of the RDFS inference rules. This is the case because we are not interested in handling temporal evolution of the ontology schema. What we need to handle are temporal properties of instance data. Specifically, we need to account for temporal labels of inferred *rdf:type* statements and statements resulting from *rdfs:subPropertyOf* statements. *rdf:type* statements result from the following rules: (1) (*x*, *rdf:type*, *y*) ∧ (*y*, *rdfs:subClassOf*, *z*) → (*x*, *rdf:type*, *z*), and (2) (*x*, *p*, *y*) ∧ (*p*, *rdfs:domain*, *a*) ∧ (*p*, *rdfs:range*, *b*) → (*x*, *rdf:type*, *a*), (*y*, *rdf:type*, *b*). We infer instance statements from *rdfs:subPropertyOf* using the following rule: (1) (*x*, *p*, *y*) ∧ (*p*, *rdfs:subPropertyOf*, *z*) → (*x*, *z*, *y*). In each case, if we assume that schema level statements in the ontology are eternally true, the temporal label of an inferred instance statement *s* is the union of the time intervals of all statements which can be used to infer *s*.

We provide the procedure *build_temporal_index* (*ontology*, *rules_index_name*, *min_start_time*, *max_end_time*) to construct a temporal index for a given ontology and rules index. The *ontology* parameter identifies the ontology stored in Oracle; *rules_index_name* identifies the RDFS rules index associated with the ontology; *min_start_time* and *max_end_time* specify the earliest date and the latest date in the ontology. The purpose of these boundary parameters is to act as the start time and end time of statements which are eternally true (i.e. schema-level statements and statements with no asserted temporal properties). This procedure executes in three phases.

The first phase creates the temporary table *asserted_temporal_triples* (*subj_id NUMBER*, *prop_id NUMBER*, *obj_id NUMBER*, *start DATE*, *end DATE*). The ontology is then queried to retrieve all temporal reifications. The subject, property, and object ids of each temporally reified statement and the start time and end time are inserted into this temporary table. The final step of this phase inserts statements without asserted temporal reifications into the *asserted_temporal_triples* table using *min_start_time* and *max_end_time* as the start and end times, and all schema-level statements also receive these start and end values.

At this point, we have recorded the temporal values for each asserted statement, and the second and third phases perform the temporal inferencing process and create the final temporal triples table (see Fig. 2). In the procedure *TemporalInference* (shown below), we first create a second temporary table *redundant_triples* (*subj_id NUMBER*, *prop_id NUMBER*, *obj_id NUMBER*, *start DATE*, *end DATE*). Then, we iterate through the *asserted_temporal_triples* table and add any inferred statements to the *redundant_triples* table. In this step, the temporal label of the asserted statement is directly assigned to the corresponding inferred statements. This procedure results in possibly redundant and overlapping intervals for each statement, so a third phase iterates through this table and cleans up the time intervals for each statement. The cleanup phase first sorts *redundant_triples* by (*subj*, *prop*, *obj*, *start_date*) and then makes a single pass over the sorted set to merge the overlapping intervals. The final result of this process is a table *TemporalTriples* (*subj_id NUMBER*, *prop_id NUMBER*, *obj_id NUMBER*, *start DATE*, *end DATE*) which contains the complete set of asserted and inferred temporal triples (see Fig. 2).

```
Procedure TemporalInference
1:  create temporary table redundant_triples (subj_id, prop_id, obj_id, start, end)
2:  for each row r ∈ asserted_temporal_triples do
3:    if (r.prop = rdf:type) then
4:      for each Class C ∈ SuperClasses(r.obj) do
5:        insert row (r.subj, rdf:type, C, r.start_date, r.end_date) into redundant_triples
6:      end for
7:    else
8:      for each property P ∈ SuperProperties(r.prop) do
9:        insert row (r.subj, P, r.obj, r.start_date, r.end_date) into redundant_triples
10:     end for
11:     x ← domain(r.prop)
12:     for each Class C ∈ SuperClasses(x) ∪ {x} do
13:       insert row (r.subj, rdf:type, C, r.start_date, r.end_date) into redundant_triples
14:     end for
15:     y ← range(r.prop)
12:     for each Class C ∈ SuperClasses(y) ∪ {y} do
13:       insert row (r.obj, rdf:type, C, r.start_date, r.end_date) into redundant_triples
14:     end for
15:   end if
16: end for
```

### 4.2 Operator Implementation

In this section we discuss the implementation of SQL table functions corresponding to the previously defined spatial and temporal operators. The table functions were implemented using Oracle's *ODCITable* interface methods [27]. With this scheme, users implement a *start*(), *fetch*() and *close*() method for the table function. The *start*() method initializes a scan context parameter. In this method, the query parameters are parsed and a SQL query is prepared and executed and a handle to the query is stored in the scan context. The *fetch*() method fetches a subset of rows from the prepared query and returns them. The *fetch*() method is invoked as many times as necessary by the kernel until all result rows are returned. The *close*() method performs cleanup operations after the last *fetch*() call. We also implement an optional *describe*() method which is used notify the kernel of the structure of the data type to be returned (i.e., columns of the table). This method is necessary because the number of columns in the return type depends upon the graph pattern and cannot be determined until query compilation time.

**Graph Pattern to SQL Translation.** Each of the table functions takes a graph pattern and ontology as input. Therefore, the conversion of a graph pattern to a SQL query is a central component of each function. The graph pattern is transformed into a self-join query against the *TemporalTriples* table corresponding to the input ontology. We will illustrate this process with the following example:

```
(?a  on_crew_of ?b)(?b used_in ?c)
```

First, URIs in the graph pattern are resolved to numeric ids through a lookup in the *RDFValues* table. Assume that in this case the ids of *member_of* and *used_in* are 1 and 2 respectively. Next we perform a self join of the *TemporalTriples* table with two

sets of conditions in the where clause: (1) we must restrict the rows of each table based on the ids of the URIs in the graph pattern and (2) we must create a join condition based on variable correspondences between different parts of the graph pattern. We must also join with the *RDFValues* table to resolve the ids of URIs bound to variables to actual URI Strings for return from the function. The graph pattern above results in the following query:

```
select rv1.uri, rv2.uri, rv3.uri
from   TemporalTriples t1, TemporalTriples t2,
       RDFValues rv1, RDFValues rv2, RDFValues rv3
where  t1.prop_id = 1 and t2.prop_id = 2 and
       t1.obj_id = t2.subj_id and rv1.id = t1.subj_id
       and rv2.id = t1.obj_id and rv3.id = t2.obj_id;
```

**Spatial Operators.** Spatial operators are implemented by augmenting the base graph pattern query discussed in the previous section when it is created and executed in the *fetch*() routine.

In the *spatial_extent* operator, we modify the query as follows. First we identify the appropriate column (i.e., *subj_id*, *prop_id*, or *obj_id*) in the *RDFTriples* table which corresponds to the position of the *spatial_variable* parameter. Then we add an additional join matching ids from the *temporal_triples* table with *value_ids* in the *SpatialData* table to select the id of the *SDO_GEOMETRY* object. We must return the *id*, rather than the *SDO_GEOMETRY* object, from *SpatialData* because object types cannot be returned from table functions. In the case of optional result filtering, we need to modify the where clause so that we filter the spatial features from *SpatialData* according to the input spatial feature and spatial relation. This is done by adding the appropriate *sdo_relate* or *sdo_within_distance* predicate available in Oracle Spatial. For example, given the query `spatial_extent (..., sdo_geometry (...), 'geo_relate (inside)')`, we would modify the query as follows: `where ... and sdo_relate (geo.shape, sdo_geometry (...), 'mask=inside') = 'true';`

For the *spatial_eval* operator, we implement what is essentially a nested loop join (NLJ) using the basic *spatial_extent* and filtered *spatial_extent* operators. We first construct and execute a basic *spatial_extent* query in the *start*() routine. Next, in the *fetch*() routine, we consume a row from the *spatial_extent* query and then construct and execute the appropriate filtered *spatial_extent* query using the second pair of graph pattern and spatial variable parameters and the spatial relation parameter. This is repeated until all rows in the outer *spatial_extent* query are consumed. This NLJ strategy is needed to avoid an awkward query plan on what would be a very large single base query.

**Temporal Operators.** The implementation of the temporal operators does not translate directly to a SQL query. We must do some extra processing of the base query results in the *fetch*() routine to form a single time interval for each found graph pattern instance.

For the *temporal_extent* operator, we first augment the basic graph pattern query in *start*() to also select the start and end values for each temporal triple in the graph pattern instance. In the *fetch*() routine, to compute the final temporal interval for each graph pattern instance, we examine the start and end times for each triple and select

the earliest start and latest end (*RANGE*) or the latest start and earliest end (*INTERSECT*). In the case of *INTERSECT*, if the final start value is later than the final end value then the computed interval is not valid and is not included in the final result. When the optional filtering parameters are specified, we must perform additional checking of the found graph patterns to ensure they satisfy the filter condition. In addition to these extra computations in *fetch*(), we augment the base query in *start*() with a series of predicates involving the start and end times of each statement in the graph pattern. This is done to filter the results as much as possible in the base query to reduce subsequent overhead in *fetch*(). To illustrate these additional predicates, consider the following *temporal_extent* query and corresponding base query:

```
select ...
from table(temporal_extent('(?x on_crew_of ?y)(?y
     used_in ?z)', 'range', 1942, 1944, 'during'));

select ...
from ..., TemporalTriples t1, TemporalTriples t2
where ... and t1.start > 1942 and t2.end < 1944
          and t2.start > 1942 and t2.end < 1944;
```

The implementation of the *temporal_eval* operator is similar to the implementation of *spatial_eval*. We first build a basic *temporal_extent* query involving the first pair of graph pattern and interval type parameters which is executed in the *start*() routine. Next, in *fetch*(), we consume a row from the basic *temporal_extent* query and execute an appropriate filtered *temporal_extent* query using the second pair of graph pattern and interval type parameters. This query uses the time interval from the current outer *temporal_extent* result and the inverse of the temporal relation parameter from the original *temporal_eval* query.

## 5 Experimental Evaluation

In this section, we describe the experimental evaluation of our spatial and temporal query operators. All experiments were conducted using Oracle 10g Release 2 running on a Red Hat Enterprise Linux machine with dual Xenon 3.0 GHz processors and 2 GB of main memory. The database used an 8 KB block size and was configured with an *sga_target* size of 512 MB and a *pga_aggregate_target* size of 512 MB. The times reported for each query are an average of 15 trials using a warm cache. Times were obtained by querying for *systimestamp* before and after query execution and computing the difference. Datasets and queries can be downloaded from http://knoesis.wright.edu/students/mperry/STData.html.

**Dataset.** Three synthetically generated datasets were used in our experiments. The datasets correspond to a small ontology schema from the military domain that we created with the overall idea being to analyze historical entities and events of WWII. The ontology schema defined 15 class types and 9 property types. Each dataset was created in three phases. First we populated the thematic portion of the ontology.

Second we added spatial information, and in the final step we generated temporal labels for the statements in the populated ontology.

To populate the thematic portion of the military ontology, we used the ontology population tool described in [28]. This tool inputs an ontology schema and relative probabilities for generating instances of each class type and each property type defined in the schema. Based on these probabilities, it creates instance data, which, in effect, simulates the population of the ontology. We generated three RDF datasets this way. The first contained 95,000 triples, the second contained 1.6 million triples and the third contained over 15 million triples (asserted and inferred statements). We integrated these military RDF graphs with the upper-level ontology described in [7] by adding a handful of *rdfs:subClassOf* statements to each RDF dataset.

To add spatial aspects to this dataset, we randomly assigned spatial features to each instance of *Geometry* in the ontology with uniform probability. We used year 2000 census block group boundary polygons from the US Census Bureau for the spatial features [29]. Differently-sized sets of contiguous US States were chosen in proportion with the ontology size. The total numbers of features for each dataset were 873, 9,352 and 83,236 for the small, medium and large ontology, respectively.

The final phase of dataset generation assigned temporal labels to statements in the ontology. Temporal intervals were randomly assigned to each asserted instance statement in the datasets which resulted from the previous two steps. Start times and end times for each interval were randomly selected with uniform probability from two overlapping date ranges. For example, start times could be selected from the range [1940-01-01 00:00:00, 1943-12-31 23:59:59] and end times from the range [1941-01-01 00:00:00, 1944-12-31 23:59:59]. We ensured that each interval was valid (i.e., start time earlier than end time) before adding it to the dataset. The temporal inferencing procedure described in Section 4.1 was then executed to generate temporal labels for inferred statements.

**Experiments.** Our experiments are designed to characterize the overall performance of our approach with respect to (1) ontology size and (2) graph pattern complexity. For testing, B-Tree indexes were created on each column of the *TemporalTriples* table and on the *id* and *value_id* columns of the *SpatialData* table, and an R-Tree index was created on the shape column of *SpatialData*. We also created two additional B-Tree indexes (*prop_id*, *subj_id*, *obj_id*, *start_date*, *end_date*) and (*prop_id*, *obj_id*, *subj_id*, *start_date*, *end_date*) on the *TemporalTriples* table. For the 15 million triple dataset, the physical size of the *TemporalTriples* table was 642 MB, and the inferencing procedure took 1 hour and 31 minutes to execute, which compared with 1 hour and 11 minutes for Oracle rules index creation. The *SpatialData* table was 47 MB in size.

**Query Execution Time.** Table 1 summarizes the results of our experimentation with respect to ontology size.

Experiments 1 through 4 were designed to test the general scalability of basic *temporal_extent* and *spatial_extent* queries. Experiments 1 and 3 measured the response time (i.e., time to return the first 1000 rows of results) for a very unselective query. Our unselective graph patterns consisted of 3 triples and 4 variables. For each triple in the pattern a constant URI was given for the *property*, and the *subject* and

*object* were left as variables. We used 4 different graph patterns for *temporal_extent* with an *INTERSECT* type query in each case. For *spatial_extent*, 3 different graph patterns were used. In each case, the DBMS uses a NLJ strategy for evaluating the base query which results in response times which are essentially constant across each dataset. Experiments 2 and 4 are designed to measure scalability for a very selective graph pattern. For experiment 2, we used 5 different graph patterns consisting of 3 triples and 3 variables. For experiment 4 we used 3 different graph patterns with 3 triples and 3 variables. The graph patterns are of the same basic form as the previous experiment except we replace one of the variables in the *subject* or *object* position with a constant URI. This restricts the nodes in the resulting graph pattern instance instead of just the edges, providing a much more selective query. In each case, query execution time increases slightly as the ontology size increases, which is a consequence of scanning larger indexes during query evaluation and querying a larger *SpatialData* table.

**Table 1.** Experimental results for query execution time with respect to ontology size

| Operator (Exp. #) | Graph Pattern Type | | Queries | Avg. Result Size | Avg. Execution Time for each ontology (ms) | | |
|---|---|---|---|---|---|---|---|
| | # Vars | # Triples | | | Small | Medium | Large |
| T-Ext (1) | 4 | 3 | 4 | N/A | 394 | 390 | 385 |
| (2) | 3 | 3 | 5 | 221 | 22 | 32 | 48 |
| S-Ext (3) | 4 | 3 | 3 | N/A | 360 | 350 | 365 |
| (4) | 3 | 3 | 3 | 100 | 22 | 30 | 67 |
| T-Filter(5) | 4 | 3 | 4 | 312 | 157 | 345 | 714 |
| S-Filter (6) | 4 | 3 | 3 | 331 | 173 | 192 | 374 |
| T-Eval(7) | 2/2 | 2/2 | 3 | 129 | 414 | 411 | 437 |
| | 2/3 | 3/3 | 3 | 220 | 306 | 195 | 268 |
| S-Eval (8) | 2/2 | 2/1 | 3 | 244 | 343 | 467 | 485 |
| | 2/2 | 2/3 | 3 | 209 | 251 | 385 | 457 |

In experiment 5, we measured the scalability of the *temporal_extent* operator using optional filtering with respect to dataset size. For these tests, we used very unselective graph patterns in combination with very selective temporal conditions. Note that this represents a worst case scenario for *temporal_extent*. Because we only store the temporal labels for single triples in the DB, we can only index these single triples. The temporal labels for graph pattern instances are constructed during query evaluation and therefore cannot be indexed. We must apply the temporal filter to each graph pattern instance as it is being constructed, which can potentially lead to very large intermediate result sets because in many cases we cannot exclude a graph pattern from the results until its time interval has been fully constructed. Our experiments show an increase in execution time which is roughly linear with respect to ontology size which reflects the growth of intermediate results processed during the query. Each query used the *INTERSECT* option and either a *before*, *after*, or *during* temporal relation.

In experiment 6, we measured the performance of *spatial_extent* using the optional filtering capability as dataset size increases. As with experiment 5 we combined a low selectivity graph pattern with a highly-selective spatial predicate. We used three different queries. The first retrieved results which were within a short distance of a

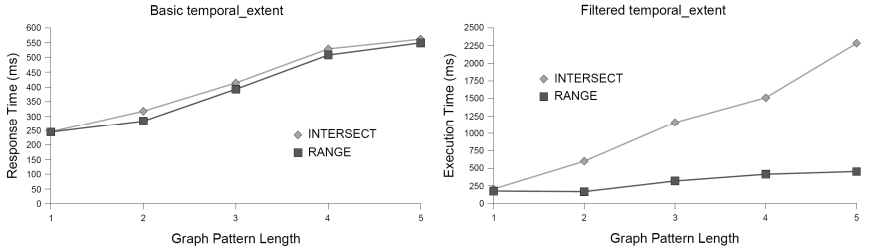


**Fig. 3.** Scalability of temporal operators with respect to graph pattern size

point; another retrieved results which were covered by an input polygon, and the final query retrieved results which intersected with an input polygon. The results show that *spatial_restrict* with filter scales better than its temporal counterpart because we can effectively index the spatial features and quickly reduce the search space using the spatial index. The execution time increases because larger indexes must be scanned when evaluating the graph pattern.

Experiment 7 illustrates the scalability of selective *temporal_eval* queries. For this test, we used selective graph patterns for both the LHS and RHS input patterns. We varied the constant URIs in the graph pattern and the temporal condition so that the result set sizes were constant across each dataset. The results show that execution time is roughly constant across each dataset with variations resulting from slight differences in the number of query restarts required in *fetch*() and the selectivity of the graph patterns used. Each query used the *INTERSECT* option and either a *before*, *after*, *during* or *any interact* temporal relation.

Experiment 8 characterizes the performance of selective *spatial_eval* queries as the dataset size increases. Again, we used selective graph patterns for both the LHS and RHS pattern and varied the constant URIs and spatial predicates so that result set size was consistent across each dataset. The results show that execution time grows slightly as ontology size increases, which is a result of scanning larger indexes and querying a larger spatial dataset.

Our next experiments were designed to test the scalability of the *temporal_extent* operator as the graph pattern size increased. We elected to perform these experiments only on the temporal queries due to space constraints, and because temporal processing is less efficient than spatial processing in our scheme, these numbers should represent an upper bound. All queries in these tests were run against the 15 million triple dataset. The graph on the left side of Fig. 3 shows the response time (first 1000 rows) of basic temporal extent queries (*INTERSECT* vs. *RANGE*) for low selectivity graph patterns of increasing length. The times are the mean of 4 different queries for a given length. Each graph pattern has a constant URI in each predicate position and variables in each subject and object position. The results show that response time scales roughly linearly with graph pattern size. More processing time is required for *INTERSECT* because of extra join conditions required to ensure valid time intervals. The graph on the right side of Fig. 3 shows the execution time for filtered *temporal_extent* queries using unselective graph patterns and selective temporal predicates. The idea behind this experiment was to bound the execution time for filtered *temporal_extent* queries. In some circumstances we can only place weak conditions on the temporal properties of each triple in the result. For example, using

*INTERSECT* and *during* [*x*, *y*], we can only enforce that each triple does not end *before x* or start *after y*. In contrast, using *RANGE* and *during* [*x*, *y*] we can enforce that each triple both starts *after x* and *ends before y*, which completely filters any unmatching graph patterns. The graph in Fig. 3 (right) shows the execution times for each scenario. Each value is the average of four different queries of that type. We can see that performance using the worst-case scenario scales much worse than the best case, but the growth is still roughly linear. The temporal predicates were increasingly selective as the pattern length increased to keep result set size constant for each query. We should note that we needed to pass a *FIRST_ROWS* hint to the query optimizer to avoid a query plan containing a full table scan in the case of the *RANGE* query (we provide an option to do this with our implementation).

## 6 Conclusions

This paper discussed an approach for realizing spatial and temporal query operators for Semantic Web data. Our work was motivated by a lack of support for spatial and temporal relationship analysis in current semantic analytics tools. Spatial and temporal data is critical in many analytical applications and must be effectively utilized for semantic analytics to reach its full potential. Our approach built upon existing support for storage and querying of RDF data and spatial data in Oracle DBMS. We created additional storage structures which allowed us to efficiently evaluate queries over semantic data which include spatial and temporal aspects. These queries were enabled using SQL table functions. A set of experiments using a synthetic RDF dataset of over 15 million triples showed that our implementation provided reasonable performance for a fairly large populated ontology. Basic *temporal_extent* and *spatial_extent* queries were quite fast in all circumstances. The worst performance was seen with filtered *temporal_extent* queries using low selectivity graph patterns with highly selective temporal predicates. However, the resulting execution times were quite manageable.

In the future, we plan to perform further testing using other ontologies populated with both real and synthetic data. We also plan to investigate extensions of the SPARQL query language which support the types of operations discussed in this paper.

## References

1. Resource Description Framework (RDF). http://www.w3.org/RDF.
2. Anyanwu, K. and A.P. Sheth, *ρ-Queries: Enabling Querying for Semantic Associations on the Semantic Web*, in *12th Int'l World Wide Web Conference*. 2003: Budapest, Hungary.
3. Ramakrishnan, C., et al., *Discovering Informative Connection Subgraphs in Multi-relational Graphs*. SIGKDD Explorations, 2005. 7(2): p. 56-63.
4. Aleman-Meza, B., et al., *Semantic Analytics on Social Networks: Experiences in Addressing the Problem of Conflict of Interest Detection*, in *15th Int'l World Wide Web Conference*. 2006: Edinburgh, Scotland.

5. Mukherjea, S. and B. Bamba, *BioPatentMiner: An Information Retrieval System for BioMedical Patents*, in *30th Int'l Conference on Very Large Data Bases*. 2004: Toronto, Canada.
6. Kochut, K. and M. Janik, *SPARQLeR: Extended Sparql for Semantic Association Discovery*, in *Fourth European Semantic Web Conference*. 2007: Innsbruck, Austria.
7. Perry, M., F. Hakimpour, and A. Sheth, *Analyzing Theme, Space and Time: an Ontology-based Approach*, in *14th ACM Int'l Symposium on Geographic Information Systems*. 2006: Arlington, VA.
8. Brickley, D. and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation. 2000. http://www.w3.org/TR/rdf-schema/.
9. Prud'hommeaux, E. and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
10. Gruber, T.R., *A Translation Approach to Portable Ontology Specifications*. Knowledge Acquisition, 1993. 5(2): p. 199-220.
11. Gutierrez, C., C. Hurtado, and A. Vaisman, *Temporal RDF*, in *European Conference on the Semantic Web*. 2005: Heraklion, Crete, Greece.
12. Hayes, P. RDF Semantics. http://www.w3.org/TR/rdf-mt/.
13. Karvounarakis, G., et al. *RQL: A Declarative Query Language for RDF*. in *11th Int'l World Wide Web Conference*. 2002. Honolulu, Hawaii.
14. Sintek, M. and S. Decker, *TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web*, in *1st Int'l Semantic Web Conference*. 2002: Sardinia, Italy.
15. Souzis, A. RxPath Specification Proposal. 2004. http://rx4rdf.liminalzone.org/RxPathSpec.
16. Wilkinson, K., et al., *Efficient RDF storage and retrieval in Jena2*, in *VLDB Workshop on Semantic Web and Databases*. 2003: Berlin, Germany.
17. Alexaki, S., et al., *On Storing Voluminous RDF Descriptions: The Case of Web Portal Catalogs*. in *4th Int'l Workshop on the Web and Databases*. 2001. Santabarbara, Califonia.
18. Chong, E.I., et al., *An Efficient SQL-based RDF Querying Scheme*, in *31st Int'l Conference on Very Large Data Bases*. 2005: Trondheim, Norway.
19. Kammersell, W. and M. Dean, *Conceptual Search: Incorporating Geospatial Data into Semantic Queries*, in *Terra Cognita - Directions to the Geospatial Semantic Web*. 2006: Athens, GA.
20. Tanasescu, V., et al., *A Semantic Web GIS based Emergency Management System*, in *Int'l Workshop on Semantic Web for eGovernment*. 2006: Budva, Montenegro.
21. Egenhofer, M.J., *Toward the Semantic Geospatial Web*, in *10th ACM Int'l Symposium on Advances in Geographic Information Systems*. 2002: McLean, VA.
22. Jones, C.B., et al., *The SPIRIT Spatial Search Engine: Architecture, Ontologies, and Spatial Indexing*, in *3rd Int'l Conference on Geographic Information Science*. 2004: Adelphi, MD.
23. Oracle Spatial Resource Description Framework (RDF) 10g Release 2. http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b19307/toc.htm.
24. Oracle Spatial User's Guide and Reference 10g Release 2. http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b14255/toc.htm.
25. Hakimpour, F., et al., *Data Processing in Space, Time and Semantics Dimension*, in *Terra Congita - Directions to the Geospatial Semantic Web*. 2006: Athens, GA.
26. Int'l Open GIS Consortium, Open GIS Simple Feature Specification for SQL. 1999. http://portal.opengeospatial.org/files/?artifact_id=829.
27. Oracle Database Data Cartridge Developer's Guide, 10g Release 2. http://download-east.oracle.com/docs/cd/B19306_01/appdev.102/b14289/toc.htm.
28. Perry, M., TOntoGen: *A Synthetic Data Set Generator for Semantic Web Applications*. AIS SIGSEMIS Bulletin, 2005. 2(2): p. 46-48.
29. U.S. Census Bureau, Census 2000 Cartographic Boundary Files. http://www.census.gov/geo/www/cob/bg2000.html.