12-2009

S2ST: A Relational RDF Database Management System

Anthony T. Piazza
University of Texas-Pan American

S2ST: A RELATIONAL RDF DATABASE MANAGEMENT SYSTEM

A Thesis

by

ANTHONY T. PIAZZA

Submitted to the Graduate School of the
University of Texas - Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2009

Major Subject: Computer Science

S2ST: A RELATIONAL RDF DATABASE MANAGEMENT SYSTEM

A Thesis
by
ANTHONY T. PIAZZA

Approved as to style and content by:

Dr. Artem Chebotko
Chair of Committee

Dr. Robert Schweller
Committee Member

Dr. Andres Figueroa
Committee Member

Dr. Richard Fowler
Committee Member

December 2009

ABSTRACT

Piazza, Anthony T., S2ST: A Relational RDF Database Management System. Master of Science (MS), December, 2009, 45 pp., 11 figures, references, 27 titles.

The explosive growth of RDF data on the Semantic Web drives the need for novel database systems that can efficiently store and query large RDF datasets. To achieve good performance and scalability of query processing, most existing RDF storage systems use a relational database management system as a backend to manage RDF data. In this paper, we describe the design and implementation of a Relational RDF Database Management System. Our main research contributions are: (1) We propose a formal model of a Relational RDF Database Management System (RRDBMS), (2) We propose generic algorithms for schema, data and query mapping, (3) We implement the first and only RRDBMS, S2ST, that supports multiple relational database management systems, user-customizable schema mapping, schema-independent data mapping, and semantics-preserving query translation.

ACKNOWLEDGEMENTS

First, I want to thank my entire family for all of their encouragement and support. Second, I want to thank my advisor, Dr. Artem Chebotko, for allowing me to work with him. Lastly, I want to thank all of the graduate students I have studied with, for making this a truly memorable experience.

TABLE OF CONTENTS

Page

ABSTRACT…….. iii

ACKNOWLEDGEMENTS…….. iv

TABLE OF CONTENTS…….. v

LIST OF FIGURES……... vii

CHAPTER I. INTRODUCTION…….. 1

Research Motivation……. 2

Research Contributions……. 3

Organization of this Document……. 3

CHAPTER II. FOUNDATIONS OF RELATIONAL RDF DATABASE MANAGEMENT SYSTEMS……... 4

Preliminaries: RDF and SPARQL……. 4

Relational RDF Database Management System…… 6

Logical Schema……. 7

Physical Schema…… 11

Data Management Operations…… 18

CHAPTER III. SERVICE-ORIENTED ARCHITECTURE OF AN RRDBMS……... 23

CHAPTER IV. DESIGN AND IMPLEMENTATION OF S2ST…….. 27

Schema Mapping Services……. 27

Data Mapping Services…….. 30

Query Mapping Services…… 34

CHAPTER V. RELATED WORK……. 37

CHAPTER VI. CONCLUSION AND FUTURE WORK……. 41

REFERENCES…….. 42

BIOGRAPHICAL SKETCH……. 45

LIST OF FIGURES

Page

Figure 1: Sample RDF Graph……. 5

Figure 2: Function Compute-α…… 13

Figure 3: Function Compute-β…… 13

Figure 4: Algorithm CreateLS……. 18

Figure 5: Algorithm SM…….. 20

Figure 6: Algorithm DM…….. 22

Figure 7: Service-Oriented Architecture of an RRDBMS……... 24

Figure 8: S2ST Metadata Model…….. 30

Figure 9: Data Mapping Architecture of S2ST…… 31

Figure 10: Data Mapping Sequence Diagram…….. 32

Figure 11: Query Mapping Sequence Diagram…… 35

CHAPTER I

INTRODUCTION

We are living in a time of unprecedented information growth. It is estimated that the amount of digital information in the world today is doubling in size every 18 months. This explosive growth presents a number of interesting challenges for organizations that produce, collect and process large amounts of digital information. One common problem is that much of the data being produced lacks semantics. Semantics provides meaning to data. The World Wide Web currently contains a vast amount of data without semantics. The World Wide Web Consortium (W3C) has proposed standards that make it possible for data to be shared and reused across application, enterprise, and community boundaries. These standards promote the development of the next-generation Web, known as the Semantic Web.

The vast majority of information available on the Web today is published using the HyperText Markup Language (HTML). HTML is a standard for describing the structure of published information. Web browsers use this structural information to render the information in a way that facilitates consumption by humans. Information published using HTML is not intended for consumption by computers, which makes it difficult for them to make effective use of the ever increasing volume of information available on the Web. To solve this problem, theW3C has proposed new standards to enable computers to discern the meaning of available information. XML (eXtensible Markup Language) is a

W3C standard that provides a set of rules for encoding information. Adoption of XML has now become widespread. Besides having a standard way to encode information, there needs to be a standard way to express its meaning. That's the purpose of RDF (Resource Description Framework), a W3C standard that supports modeling of information that is made available as web resources. RDF is based on the idea of making statements about web resources in the form of subject-predicate-object expressions, called triples. Triples can be encoded using several different W3C standard formats, including XML, N-Triples and N3. Triples are an intuitive way to describe most of the information being processed by computers today.

A number of systems have been developed to support large-scale RDF storage using relational database backends. To resolve the conflict between the graph RDF data model and the relational data model, these systems require various mappings between the two data models, such as schema mapping, data mapping, and query mapping (a.k.a. query translation). Schema mapping is used to generate a relational database schema for storing RDF data. Data mapping is used to shred RDF triples into relational tuples and insert them into the database. Finally, query mapping is used to translate SPARQL queries into equivalent SQL queries, which are then evaluated by the relational engine and the results are returned as SPARQL query solutions.

Research Motivation

As the use of RDF becomes more widespread, so too will be the need for systems that support storing and querying of RDF data. There have been numerous such systems developed in recent years. Many of these systems are based on relational database technology. This approach leverages the mature and vigorous storage and query

capabilities provided by relational databases. A careful study of these systems reveals a number of significant limitations including: hard-coded support for only one or two fixed schema mapping strategies, data mapping and query translation algorithms that must be reinvented for every new schema mapping strategy and support for very few relational database management systems, just to name a few. Our motivation is to develop a system for storing and querying RDF data based on relational database technology without these limitations.

Research Contributions

Our research contributions are: (1) a formal model for a relational RDF database management system (RRDBMS), (2) generic algorithms for schema, data and query mapping, and (3) the realization of an RRDBMS that supports all of the most popular database management systems.

Organization of this Document

The remaining chapters of this document are organized as follows: Chapter 2 lists a number of preliminary definitions and then presents a formal model for addressing the limitations of existing RDF storage systems, Chapter 3 defines a service-oriented architecture for building a relational RDF database management system, Chapter 4 details the realization of a relational RDF database management system, Chapter 5 reviews related research, and Chapter 6 concludes the document and reviews some interesting research topics for future work.

CHAPTER II

FOUNDATIONS OF RELATIONAL RDF DATABASE MANAGEMENT SYSTEMS

It may be helpful to review some of the fundamental definitions before delving into the more complex topics which depend on them. We start this chapter by discussing some of them most important terms related to semantic web technology. After that, we present a formal model for a Relational RDF Database Management System.

Preliminaries: RDF and SPARQL

Let *I, B, L*, and *V* denote pairwise disjoint infinite sets of Internationalized Resource Identifiers (IRIs), blank nodes, literals, and variables, respectively. Let *IB, IL, IV, IBL*, and *IVL* denote $I \cup B$, $I \cup L$, $I \cup V$, $I \cup B \cup L$, and $I \cup V \cup L$, respectively. Elements of the set *IBL* are also called *RDF terms*. In the following, we formalize the notions of RDF triple, RDF graph, triple pattern, graph pattern, and SPARQL query.

Definition 1 (RDF triple and RDF graph)

An RDF triple t is a tuple $(s, p, o) \in (IB) \times I \times (IBL)$, where *s, p*, and *o* are a subject, predicate, and object, respectively. An RDF graph *G* is a set of RDF triples. We define $\mathcal{T}$ and $\mathcal{G}$ as infinite sets of all possible RDF triples and graphs, respectively.

A sample RDF graph that we use for subsequent examples is shown in Figure 1. The RDF graph is represented as a set of 11 triples, as well as a labeled graph, in which edges are directed from subjects to objects and represent predicates, circles denote IRIs, and rectangles denote literals.

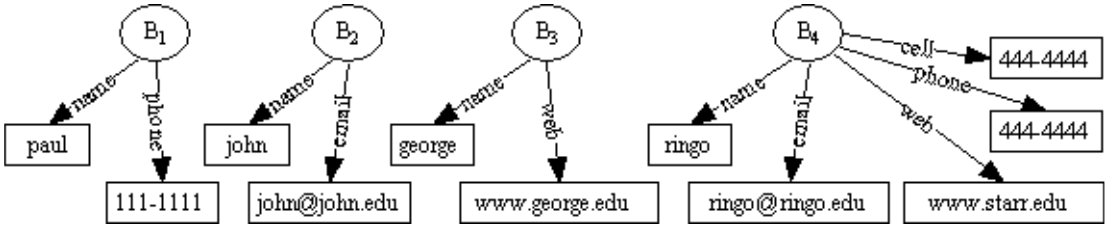

Figure 1: Sample RDF Graph

We focus on the core fragment of SPARQL defined in the following.

Definition 2 (Triple pattern)

A triple pattern *tp* is a triple $(sp, pp, op) \in (IVL) \times (IV) \times (IVL)$, where *sp, pp*, and *op* are a subject pattern, predicate pattern, and object pattern, respectively. We define $\mathcal{TP}$ as an infinite set of all possible triple patterns.

Definition 3 (Graph pattern)

A graph pattern *gp* is defined by the following abstract grammar:

$$gp \rightarrow tp \mid gp \text{ AND } gp \mid gp \text{ OPT } gp \mid gp \text{ UNION } gp \mid gp \text{ FILTER } expr$$

where AND, OPT, and UNION are binary operators that correspond to SPARQL conjunction, OPTIONAL, and UNION constructs, respectively. FILTER *expr* represents

the FILTER construct with a boolean expression *expr*, which is constructed using elements of the set *IVL*, constants, logical connectives (¬, ∨, ∧), inequality symbols (<, ≤, ≥, >), the equality symbol (=), unary predicates like bound, isIRI, and other features defined in [24]. We define $\mathcal{GP}$ as an infinite set of all possible graph patterns.

Definition 4 (SPARQL query)

A SPARQL query *sparql* is defined as

$$sparql \rightarrow \text{SELECT } varlist \text{ WHERE } (gp)$$

where $varlist = (v_1, v_2, ..., v_n)$ is an ordered list of variables and $varlist \subseteq var(gp)$. We define $\mathcal{Q}$ as an infinite set of all possible SPARQL queries that can be generated by the defined grammar.

Relational RDF Database Management System

A Relational RDF Database Management System (RRDBMS) relies on a Relational Database Management System (RDBMS) to store and query RDF datasets. RRDBMS provides a collection of data structures and algorithms that map operations on RDF data to equivalent operations on relational data in an RDBMS. In this section, we formalize the notion of RRDBMS by giving its high-level definition first and defining its individual components afterwards.

Definition 5 (Relational RDF Database Management System)

A relational RDF database management system (RRDBMS) is a tuple ($\mathcal{RDBMS}$, $\mathcal{DB}$, $\mathcal{LS}$, $\mathcal{PS}$, $\mathcal{ALG}$), where

- $\mathcal{RDBMS}$ is a set of RDBMS backends that manage RDF data,
- $\mathcal{DB}$ is a set of relational databases implemented in the RDBMS backends to store RDF data,
- $\mathcal{LS}$ is a set of logical schemas that specify how a new relational database (becomes an element in $\mathcal{DB}$) can be created,
- $\mathcal{PS}$ is a set of physical schemas that are extended instantiations of logical schemas, such that each physical schema $PS \in \mathcal{PS}$ describes a relational database $DB \in \mathcal{DB}$ and is derived from a logical schema $LS \in \mathcal{LS}$, and
- $\mathcal{ALG}$ is a collection of algorithms that perform operations in the RRDBMS, such as creation of a logical schema, creation of a physical schema and relational database schema, mapping of RDF data to relational data, and SPARQL-to-SQL query translation.

While the notions of RDBMS and relational database are well-understood, RRDBMS logical schemas, physical schemas and algorithms require further explanation found in the following subsections.

Logical Schema

The purpose of a logical schema is to encode the structure of a relational database that can be used for RDF storage, such that this structure can be later instantiated in one or more RDBMSs. Therefore, the logical schema should record a set of relation names $\mathcal{R}$ and a set of relational attribute names $\mathcal{A}$, such that each $a \in \mathcal{A}$ is associated with one or many relations in $\mathcal{R}$. While attribute names (further "attributes" for simplicity) are represented by string literals, relation names (further "relations" for simplicity) may be

data-driven, i.e., they may depend on values found in RDF data, and thus may have more complex structure. In addition, the logical schema should capture the information about what triples each relation can store and what attributes of the relation are used to store the components (subject, predicate, and object) of triples. To achieve this, we introduce two mappings, called γ and δ.

Definition 6 (Mapping γ)

Given a set of relations $\mathcal{R}$ and a set of triple patterns $\mathcal{TP}$, a mapping γ is a many-to-many mapping $\gamma : \mathcal{R} \rightarrow \mathcal{TP}$, if given a relation $R \in \mathcal{R}$, $\gamma(R)$ is a set of triple patterns $\mathcal{TP}_R = \{tp_1, tp_2, ..., tp_n\} \subset \mathcal{TP}$, such that for any two distinct triple patterns $tp_i \in \mathcal{TP}_R$ and $tp_j \in \mathcal{TP}_R$, tp_i does not subsume tp_j and tp_j does not subsume tp_i.

Mapping γ precisely defines what RDF triples can be stored in relation $R \in \mathcal{R}$, such that if triple $t \in \mathcal{T}$ matches triple pattern $tp \in \gamma(R)$, then R is used to store t. As we mentioned earlier, besides string literals, $R \in \mathcal{R}$ may include one or more special variables *%sub%, %pre%*, and *%obj%*, that are interpolated using the corresponding values of a triple $t = (s, p, o) \in \mathcal{T}$, such that t matches a triple pattern $tp \in \gamma(R)$. This provides support for data-driven relations, whose names are derived only when RDF data is being inserted into an RRDBMS.

Mapping δ defines what specific components of RDF triples, i.e., subject, predicate, and object, relational attributes can store.

Definition 7 (Mapping δ)

Given a set of relations $\mathcal{R}$, a set of relational attributes $\mathcal{A}$, and a set $\mathcal{P} = \{sub, pre, obj\}$, a mapping δ is a many-to-one mapping $\delta : \mathcal{R} \times \mathcal{A} \rightarrow \mathcal{P}$, if given a relation $R \in \mathcal{R}$ and its attribute $a \in \mathcal{A}$, $\delta(R, a)$ returns a position $pos \in \mathcal{P}$, such that for any two distinct attributes a_1 and a_2 of R, if $pos_1 = \delta(R, a_1)$ and $pos_2 = \delta(R, a_2)$, then $pos_1 \neq pos_2$.

Mapping δ restricts a relational attribute to store subjects, predicates or objects, but not the combination of those, i.e., the same attribute cannot store a subject of one triple and an object of another triple. In addition, if one attribute of a relation stores triple subjects, no other attribute can store subjects; the same is true for predicates and objects. Therefore, a relation can have at most one attribute for each position.

The last mapping that we need is denoted as τ and captures data types $\mathcal{D}$ of attributes $\mathcal{A}$ found in relations $\mathcal{R}$. To avoid dependence on data types in a particular RDBMS, we can use generic data types, such as string, date, and double, defined in the XML Schema language.

Definition 8 (Mapping τ)

Given a set of relations $\mathcal{R}$, a set of relational attributes $\mathcal{A}$, and a set of XML Schema data types $\mathcal{D}$, a mapping τ is a many-to-one mapping $\tau : \mathcal{R} \times \mathcal{A} \rightarrow \mathcal{D}$, such that given a relation $R \in \mathcal{R}$ and its attribute $a \in \mathcal{A}$, $\tau(R, a)$ returns a data type $d \in \mathcal{D}$.

These three mappings constitute a logical schema.

Definition 9 (Logical Schema)

A logical schema $\mathcal{LS}$ is a tuple (*lsid*, γ, δ , τ), where *lsid* is a unique identifier of the schema, γ is a mapping as in Definition 6, δ is a mapping as in Definition 7, and τ is a mapping as in Definition 8.

The logical schema definition is very flexible, enabling encoding different types of relations supported in schema-oblivious, schema-aware, data-driven, and hybrid relational RDF stores. Moreover, γ and δ allow the design of new types of relations, resulting in a novel user-customized approach to schema design. In the following example, we show a logical schema that implements relations used by different approaches.

Example 10 (Logical Schema)

A database designer may specify the following logical schema that may be used for the sample RDF graph in Figure 1.

lsid: 1

γ: *Triple* → {(*?s, ?p, ?o*)},
Name → {(*?s, name, ?o*)},
Class%obj% → {(*?s, type, ?o*)},
Phone → *{(?s, cell, ?o*), (*?s, phone, ?o*)}.

δ:				τ:			
	(*Triple, s*)	→	*sub*		(*Triple, s*)	→	*xsd:string*
	(*Triple, p*)	→	*pre*		(*Triple, p*)	→	*xsd:anyURI*
	(*Triple, o*)	→	*obj*		(*Triple, o*)	→	*xsd:string*
	(*Name, s*)	→	*sub*		(*Name, s*)	→	*xsd:anyURI*
	(*Name, o*)	→	*obj*		(*Name, o*)	→	*xsd:string*
	(*Class%obj%, i*)	→	*sub*		(*Class%obj%, i*)	→	*xsd:anyURI*
	(*Phone, s*)	→	*sub*		(*Phone, s*)	→	*xsd:anyURI*
	(*Phone, p*)	→	*pre*		(*Phone, p*)	→	*xsd:anyURI*
	(*Phone, o*)	→	*obj*		(*Phone, o*)	→	*xsd:unsignedInt*

According to this schema, three relations with fixed names (*Triple*, *Name*, and *Phone*) and one data-driven relation *Class%obj%* are defined. *Triple* can store all possible RDF triples as specified by the triple pattern (*?s, ?p, ?o*) in three columns *s, p, o* that correspond to a subject, predicate, and object, and have data types *xsd:string*, *xsd:anyURI*, and *xsd:string*, respectively. Similarly, the structure of relations *Name* and *Phone* is defined as *Name(s : xsd:anyURI, o : xsd:string)* and *Phone(s : xsd:anyURI, p : xsd:anyURI , o : xsd:unsignedInt)*. *Name* is intended to store subjects and objects of any RDF triple whose predicate is *name*, i.e., the triple matches triple pattern *(?s, name, ?o)*. More interestingly, *Phone* is allowed to store any RDF triple whose predicate is *cell* or *phone*, i.e., the triple matches *(?s, cell, ?o)* or *(?s, phone, ?o)*. Finally, the actual name of relation *Class%obj%* is derived from a triple itself, such that special variable *%obj%* is interpolated with the object value of a triple that matches triple pattern (*?s, type, ?o*). For example, if triple (B_1*, type, Person*) is in the graph, its subject is to be stored by relation *ClassPerson(i : xsd:anyURI)*.

The four relations are representative of four different approaches to schema design, namely schema-oblivious (*Triple*), schema-aware (*Name*), data-driven (*Class%obj%*), and user-driven (*Phone*), resulting in a flexible hybrid design.

Physical Schema

The logical schema serves as a template that can be applied to generate relational database schemas in one or more RDBMS. Once a relational database schema is created in an RDBMS, we derive a new set of mappings that describe the concrete storage structure. This set of mappings is referred to as *physical schema*.

In a physical schema, mappings γ and δ are initially inherited from the corresponding logical schema. If data-driven relations are used, these mappings may be augmented with new instances. Similarly, mapping τ is inherited from the corresponding logical schema with generic data types mapped to RDBMS-specific data types. τ may also evolve when data-driven relations are created.

Next, while mappings γ and δ are good means to capture what data can be stored in relations, they are not very straightforward to use for deciding how to insert new triples or match SPARQL triple patterns over relations. One step towards this goal is defining reverse mappings $\gamma^{-1} : \mathcal{TP} \rightarrow \mathcal{R}$ and $\delta^{-1} : \mathcal{P} \rightarrow \mathcal{R} \times \mathcal{A}$. The reverse mappings may not be easy to use, because γ^{-1} is defined on a finite set of triple patterns that may subsume other triple patterns and δ^{-1} returns a set for a given position. Therefore, to better support data mapping and query translation, we introduce mappings α and β, deriving them from γ^{-1} and δ^{-1}, respectively.

Definition 11 (Mapping α)

Given a set of triple patterns $\mathcal{TP}$, a set of triples $\mathcal{T}$, and a set of relations $\mathcal{R}$, a mapping α is a many-to-many mapping $\alpha : \mathcal{TP} \cup \mathcal{T} \rightarrow \mathcal{R}$, if given a triple pattern $tp \in \mathcal{TP}$ (or triple $t \in \mathcal{T}$), $\alpha(tp)$ (or $\alpha(t)$) is a set of relations $R_{tp} = \gamma^{-1}(tp_1) \cup \gamma^{-1}(tp_2) \cup \cdots \cup \gamma^{-1}(tp_n) \subseteq \mathcal{R}$, where γ^{-1} is defined on tp_1, tp_2, ..., tp_n and each triple pattern tp_i subsumes tp (or matches t). Therefore, each relation $R \in \alpha(tp)$ is used to store all the triples that match tp or each relation $R \in \alpha(t)$ is used to store triple t.

A function for computing α is shown in Figure 2.

```
01  Function Compute-α
02  Input: triple pattern or triple tp, mapping γ^-1
03  Output: set of relations R_tp that store triples that match tp
04  Begin
05      Let R_tp be an empty set
06      Let D be a domain of γ^-1; D is a set of triple patterns
07      For each tp' ∈ D do
08          If /* tp' subsumes tp */
09            [tp'.sp ∈ V or (tp.sp ∉ V and tp.sp == tp'.sp)] and
10            [tp'.pp ∈ V or (tp.pp ∉ V and tp.pp == tp'.pp)] and
11            [tp'.op ∈ V or (tp.op ∉ V and tp.op == tp'.op)] and
12            [tp'.sp ≠ tp'.pp or tp.sp == tp.pp] and
13            [tp'.sp ≠ tp'.op or tp.sp == tp.op] and
14            [tp'.pp ≠ tp'.op or tp.pp == tp.op]
15          then R_tp = R_tp ∪ γ^-1(tp')
16          End If
17      End For
18  Return R_tp
19  End Function
```

Figure 2: Function Compute-α

Definition 12 (Mapping β)

Given a set of relations $\mathcal{R}$, a set of relational attributes $\mathcal{A}$, and a set of positions $\mathcal{P}$ = {*sub, pre, obj*}, a mapping β is a many-to-one mapping $\beta : \mathcal{R} \times \mathcal{P} \rightarrow \mathcal{A}$, if given a relation $R \in \mathcal{R}$ and a position $pos \in \mathcal{P}$, β(R, *pos*) is a relational attribute $a \in \mathcal{A}$ that belongs to *R* and is used to store triple components at position *pos*.

A function for computing β is shown in Figure 3.

```
01  Function Compute-β
02  Input: relation R, position pos, mapping δ^-1
03  Output: a relational attribute of R that corresponds to pos
04  Begin
15      For each (R', a) ∈ δ^-1(pos) do
16          If R == R' then Return a End If
17      End For
20  Return undef /* undefined */
21  End Function
```

Figure 3: Function Compute-β

For example, given a triple pattern *tp* and mappings α and β, an RRDBMS can determine a set α(tp) of relations that store RDF data that may match *tp*, choose one relation *R* from this set, and identify relational attributes β(*R, sub*), β(*R, pre*), and β(*R, obj*) that should be checked to match *tp*'s subject pattern, predicate pattern, and object pattern, respectively.

Mappings γ, δ, τ, α, and β constitute a physical schema.

Definition 13 (Physical Schema)

A physical schema $\mathcal{PS}$ is a tuple (*psid, lsid, rdbms*, γ, δ, τ, α, β), where *psid* is a unique identifier of the physical schema, *lsid* is a unique identifier of the corresponding logical schema, *rdbms* is a reference to the corresponding RDBMS, γ is a mapping as in Definition 6, δ is a mapping as in Definition 7, τ is a mapping as in Definition 8 with the generic data types substituted by data types supported by *rdbms*, α is a mapping as in Definition 11, and β is a mapping as in Definition 12.

A physical schema is required to perform operations in an RRDBMS, such as mapping of RDF data to relational data, SPARQL-to-SQL query translation, and reconstruction of original RDF data from relational data.

Example 14 (Physical Schema)

We can derive a physical schema based on the mappings in Example 10. The first step is to select a specific RDBMS - we choose Oracle version 10g for this example and assume valid RDBMS credentials (username and password) are provided. First, we describe the usage of a physical schema for data mapping. In this situation we use a physical schema to insert triples into the appropriate relational tables.

lsid: 1

γ: *Triple* → {(*?s, ?p, ?o*)},
Name → {(*?s, name, ?o*)},
Class%obj% → {(*?s, type, ?o*)},
Phone → {(*?s, cell, ?o*), (*?s, phone, ?o*)}.

δ:				τ:			
	(*Triple, s*)	→	*sub*		(*Triple, s*)	→	*VARCHAR2(256)*
	(*Triple, p*)	→	*pre*		(*Triple, p*)	→	*VARCHAR2(256)*
	(*Triple, o*)	→	*obj*		(*Triple, o*)	→	*VARCHAR2(256)*
	(*Name, s*)	→	*sub*		(*Name, s*)	→	*VARCHAR2(256)*
	(*Name, o*)	→	*obj*		(*Name, o*)	→	*VARCHAR2(256)*
	(*Class%obj%, i*)	→	*sub*		(*Class%obj%, i*)	→	*VARCHAR2(256)*
	(*Phone, s*)	→	*sub*		(*Phone, s*)	→	*VARCHAR2(256)*
	(*Phone, p*)	→	*pre*		(*Phone, p*)	→	*VARCHAR2(256)*
	(*Phone, o*)	→	*obj*		(*Phone, o*)	→	*VARCHAR2(256)*

To store the following three triples, we use the algorithms listed in Figure 2 and Figure 3 to determine which tables and attributes will be used.

α(*B1, type, Person*) = {*Class%obj%, Triple*},
β (*Class%obj%, sub*) = *i,*
β (*Class%obj%, pre*) = *undef,*
β (*Class%obj%, obj*) = *undef,*
β(*Triple, sub*) = *s,*
β(*Triple, pre*) = *p,*
β(*Triple, obj*) = *o.*

For this triple (*B1, type, Person*), α returns a set containing two relations, *Class%obj%* and *Triple*. This means that the triple must be stored in both relations. The first one, *Class%obj%* represents a data-driven (or dynamic) relation. At runtime the name of the relation is derived using the object of the specified triple; in this case, it would be *ClassPerson*. It is possible that this relation may not exist at runtime. If necessary it can be created on-the-fly before the triple is inserted. The triple must also be

stored in the *Triple* relation. In this case the relation already exists because it was created during schema mapping. Once we know which tables will store the triple, β gives us the attributes that will be used to store the subject, predicate and object. Using β we know that attribute *i* should be used to store the subject in the relation named *ClassPerson*. When β returns *undef*, nothing is stored for the specified position. In this case it means that the predicate and object are not stored in the *ClassPerson* relation. Using β we know that attributes *s, p*, and *o* store the subject, predicate and object, respectively, in the relation named *Triple*.

$$\begin{aligned} \alpha(B_1, name, paul) &= \{Name, Triple\}, \\ \beta(Name, sub) &= s, \\ \beta(Name, pre) &= undef, \\ \beta(Name, obj) &= o, \\ \beta(Triple, sub) &= s, \\ \beta(Triple, pre) &= p, \\ \beta(Triple, obj) &= o. \end{aligned}$$

For this triple (B_1, *name, paul*), α returns a set containing two relations *Name* and *Triple*. Again, the triple must be stored in both relations. In this case, both relations already exist so the next step is to determine where to store the subject, predicate and object of this triple. For the *Name* relation, the subject is stored in attribute *s* and the object is stored in attribute *o*. The *Triple* relation is handled in exactly the same way as the previous triple.

$$\begin{aligned} \alpha(B_1, phone, 111\text{–}1111) &= \{Phone, Triple\}, \\ \beta(Phone, sub) &= s, \\ \beta(Phone, pre) &= p, \\ \beta(Phone, obj) &= o, \\ \beta(Triple, sub) &= s, \\ \beta(Triple, pre) &= p, \\ \beta(Triple, obj) &= o. \end{aligned}$$

For this triple, (*B1, phone, 111–1111*), α returns a set containing two relations

Phone and *Triple*. For both of these relations, β returns *s, p*, and *o* to store the subject, predicate and object, respectively.

Next, we describe the usage of the physical schema for query translation. In this scenario, SPARQL queries provide graph patterns to be matched. Consider the following graph patterns:

(*?a ?b ?c*) For this graph pattern, α returns a set containing one relation, *Triple*. This relation will be used to satisfy this query.

(*?a cell ?b*) For this graph pattern, α returns a set containing two relations, *Phone* and *Triple*. In this scenario, we have the choice of which relation to execute the query against. Depending on the specifics of the query mapping algorithms, there may be different reasons for selecting one relation over another. In this example, the *Phone* relation likely has fewer tuples and may therefore provide faster query execution.

(*?a type Person*) For this graph pattern, α returns a set containing one relation, *Class%obj%*. As described previously, the name of this relation is derived at runtime. In this case, it would be *ClassPerson*. During query translation, we must determine whether or not this relation has actually been realized. If it has, it can be queried and the results returned. If not, the query returns no results. From this example we can see that the usage of α during query translation is different from its use during data mapping.

```
01 Algorithm CreateLS
02 Input: ∅
03 Output: logical schema (lsid, γ, δ, τ)
04 Begin
05     Generate a unique identifier lsid
06     Construct empty mappings γ, δ, and τ
07     Let designComplete = false
08     While !designComplete do
09         Get relation name R and triple patterns {tp1, tp2, ..., tpn} from a user
10         γ(R) ={tp1, tp2, ..., tpn}
11         Get attributes names a1, a2, ..., am of R from a user
12         For each ai and tpj do
13             Get a position pos ∈ {sub, pre, obj} from a user
14             δ(R, ai) + = (tpj, pos)
15         End For
16         For each ai do
17             Get a data type d of ai from a user
18             τ(R, ai) = d
19         End For
20         designComplete = new value from a user (true or false)
21     End While
22 Return (lsid, γ, δ, τ)
23 End Algorithm
```

Figure 4: Algorithm CreateLS

Data Management Operations

There are four fundamental operations that an RRDBMS should support to store and query RDF data. The first operation is the *specification of a logical schema*, which can be done via a graphical user interface by a data architect. The architect is required to specify (1) relations and triple patterns that "describe" their purpose via mapping γ, (2) relational attributes and their relationship to triple pattern positions via mapping δ, and (3) data types of relational attributes via mapping τ. A high-level algorithm for this operation is shown in Figure 4. A logical schema produced by the *CreateLS* algorithm is stored by the RRDBMS and serves as a template for creating relational database schemas; the logical schema can be completely deleted if needed, but not altered.

The second operation, called *schema mapping*, involves the creation of a physical schema based on a given logical schema and generation of a relational database schema in an RDBMS based on the physical schema. Through the schema mapping process, the same logical schema can be instantiated multiple times in the same or different RDBMSs. Each time, a distinct physical schema and database schema are created, where the former describes how the latter is used to store RDF data. The schema mapping algorithm, *SM*, is shown in Figure 5. It first creates the mappings that constitute a physical schema and then proceeds with the creation of relations in an RDBMS. Relation and attribute names are derived from the domains of γ and δ, respectively. Data-driven relations, whose names include special variables *%sub%*, *%pre%*, and *%obj%*, are skipped, since these variables can only be interpolated when RDF data is stored into the database.

The third essential operation in an RRDBMS is data mapping, which inserts RDF data into a relational database according to a given physical schema. The data mapping algorithm is presented in Figure 6. The algorithm relies on α to get the set of relations where a triple *t* must be inserted. This set is divided into two disjoint sets, RS_1 and RS_2, where the former contains relation names with special variables that need to be interpolated and the latter contains regular relations. Relations in RS_1 are meant to be data-driven and may not yet exist in the database. Therefore, their schemas are created on the fly and the physical schema is updated accordingly. Newly created relations are also added to RS_2 to be used later for data insertion. Once all relations in RS_1 are processed, the algorithm inserts *t* into every relation in RS_2. Mapping β is used to identify relational attributes, if defined, that can store subject *t.s*, predicate *t.p*, and object *t.o*. As a result of

this operation, both database schema and physical schema may be updated and the relations are populated with RDF triples that are mapped into relational tuples.

```
01 Algorithm SM
02 Input: RDBMS rdbms and logical schema LS=(lsid, γ′, δ′, τ′)
03 Output: database schema in rdbms and physical schema PS=(psid, lsid, rdbms, γ, δ, τ, α, β)
04 Begin
05     /* physical schema creation */
06     Generate a unique identifier psid
07     Let γ = γ′, δ = δ′, and τ = τ′
08     In τ, replace all generic data types by RDBMS-specific data types supported by rdbms
09     Let α and β be functions defined in Fig. 2.2 and Fig. 2.3, respectively
10     /* database schema generation */
11     Let D_γ be a domain of mapping γ
12     Let D_δ be a domain of mapping δ
13     For each R ∈ D_γ do
14         If relation name R does not contain special variables %sub%, %pre%, and %obj% then
15         /* Relations with %sub%, %pre%, and %obj% are created during data mapping */
16             Let A = {a|(R, a) ∈ D_δ} be a set of all R's attributes
17             Execute SQL statement in rdbms:
18             Create Table R (a1 τ(R, a1), a2 τ(R, a2), ..., am τ(R, am));
19             where a1, a2, ..., am ∈ A and m = |A|
20         End If
21     End For
22 Return (psid, lsid, rdbms, γ, δ, τ, α, β)
23 End Algorithm
```

Figure 5: Algorithm SM

The final operation that an RRDBMS needs is *query translation*, such that a SPARQL query can be translated into an equivalent SQL query, which can be further executed by an RDBMS. The translation is the most complex operation in an RRDBMS, since not only locating data in correct relations and attributes is involved, but also mapping of SPARQL constructs to relational operators is required. Our semantics preserving SPARQL-to-SQL translation, called *trans*, is presented in [8]. *trans* is parameterized with mappings α′ and β' (denoted as α and β in [8]) that have similar meaning as α and β defined in this paper. The first mapping is defined as a many-to-one mapping $\alpha' : \mathcal{TP} \rightarrow \mathcal{R}$, which only differs from α in two aspects: (1) α's domain includes a set of triples to support the data mapping operation ($\alpha : \mathcal{TP} \cup \mathcal{T} \rightarrow \mathcal{R}$) and (2) α is a many-to-many mapping. While the first difference does not affect the translation, the second one

requires selecting one relation from a set returned by α. This, in fact, is an advantage of α over α′ that enables additional optimizations in an RRDBMS. In particular, a query optimizer may select to use a relation that (1) has the smallest cardinality, (2) is in cache, (3) is already chosen in another part of the same query, (4) has appropriate indexes, and so forth. Therefore, α′ can be straightforwardly derived from α: in the worst case, by randomly selecting a relation in the result of α. The second mapping is defined as a many-to-one mapping $\beta' : \mathcal{TP} \times \mathcal{P} \rightarrow \mathcal{A}$, which again is slightly different from a many-to-one mapping $\beta : \mathcal{R} \times \mathcal{P} \rightarrow \mathcal{A}$. Since a triple pattern maps to exactly one relation by α′, for some $tp \in \mathcal{TP}$ and $pos \in \mathcal{P}$, $\beta'(tp, pos) = (\alpha'(tp), pos)$. Thus, *trans* can support SPARQL-to-SQL translation for physical schemas defined in our work.

The presented set of operations is rather minimal. Additional useful operations, such as deletion of a logical schema, physical schema, and database schema, RDF data update and deletion, can be supported by an RRDBMS. We also did not touch indexing of RDF data. These are beyond the scope of this document.

01 **Algorithm** *DM*
02 **Input:** set of triples T, RDBMS $rdbms$, and physical schema PS=($psid$, $lsid$, $rdbms$, γ, δ, τ, α, β)
03 **Output:** tuples are inserted into the database in $rdbms$; the database schema and PS may be updated
04 **Begin**
05 **For** each triple $t \in T$ **do**
06 $RS_1 = \{R \mid R \in \alpha(t)$ and R contains special variables $\%sub\%$, $\%pre\%$, and $\%obj\%$ $\}$
07 /* RS_1 is a set of relations with special variables to insert t */
08 $RS_2 = \{R \mid R \in \alpha(t)$ and R contains no special variables $\%sub\%$, $\%pre\%$, and $\%obj\%$ $\}$
09 /* RS_2 is a set of relations without special variables to insert t */
10 **While** $RS_1 \neq \emptyset$ **do** /* Creating data-driven relations */
11 Choose $R \in RS_1$
12 $R' = R$ /* make a copy of R */
13 Replace $\%sub\%$, $\%pre\%$, and $\%obj\%$ with $t.s$, $t.p$, and $t.o$, respectively, in R
14 **If** $R \notin RS_2$ /*relation does not exist*/ **then**
15 /* Creating a table */
16 Let D_δ be a domain of mapping δ
17 Let $A = \{a|(R', a) \in D_\delta\}$ be a set of all R's attributes
18 Execute SQL statement in $rdbms$:
19 `Create Table` R `(`a_1 $\tau(R', a_1)$`,` a_2 $\tau(R', a_2)$`, ...,` a_m $\tau(R', a_m)$`);`
20 where a_1, a_2, ..., $a_m \in A$ and $m = |A|$
21 /* Updating the physical schema */
22 $\gamma(R) = \gamma(R')$
23 $\delta(R, a_1) = \delta(R', a_1)$, $\delta(R, a_2) = \delta(R', a_2)$, ..., $\delta(R, a_m) = \delta(R', a_m)$
24 $\tau(R, a_1) = \tau(R', a_1)$, $\tau(R, a_2) = \tau(R', a_2)$, ..., $\tau(R, a_m) = \tau(R', a_m)$
25 α and β are recomputed
26 $RS_2 = RS_2 \cup \{R\}$ /* Updating RS_2 with a newly created relation */
27 **End If**
28 $RS_1 = RS_1 - \{R'\}$ /* R' has been processed */
29 **End While**
30 **While** $RS_2 \neq \emptyset$ **do** /* Inserting a triple into the relations */
31 Choose $R \in RS_2$
32 *attribute-list* = "" /* empty string */
33 *value-list* = "" /* empty string */
34 **If** $\beta(R, sub) \neq undef$ /* attribute to store a subject exists */ **then**
35 *attribute-list* += $\beta(R, sub)$
36 *value-list* += $t.s$
37 **End If**
38 **If** $\beta(R, pre) \neq undef$ /* attribute to store a predicate exists */ **then**
39 **If** *attribute-list* $\neq$ "" **then** *attribute-list* += ", " **End If**
40 **If** *value-list* $\neq$ "" **then** *value-list* += ", " **End If**
41 *attribute-list* += $\beta(R, pre)$
42 *value-list* += $t.p$
43 **End If**
44 **If** $\beta(R, obj) \neq undef$ /* attribute to store an object exists */ **then**
45 **If** *attribute-list* $\neq$ "" **then** *attribute-list* += ", " **End If**
46 **If** *value-list* $\neq$ "" **then** *value-list* += ", " **End If**
47 *attribute-list* += $\beta(R, obj)$
48 *value-list* += $t.o$
49 **End If**
50 Execute SQL statement in $rdbms$:
51 `Insert Into` R `(`*attribute-list*`) Values (`*value-list*`);`
52 $RS_2 = RS_2 - \{R\}$ /* R has been processed */
53 **End While**
54 **Return** ($psid$, $lsid$, $rdbms$, γ, δ, τ, α, β)
55 **End Algorithm**

Figure 6: Algorithm DM

CHAPTER III

SERVICE-ORIENTED ARCHITECTURE OF AN RRDBMS

S2ST is our implementation of an RRDBMS. It is built on a layered architecture that is commonly used for enterprise applications. Figure 7 shows the layers of the architecture. The *Presentation Layer* provides functionality for performing user tasks such as creating schemas, loading data, and executing queries. The *Services Layer* provides functionality needed by the *Presentation Layer* such as schema mapping, data mapping and query mapping. The *Data Layer* provides storage and query functionality needed by the *Services Layer*. One of the primary motivations for using a layered architecture is to provide support for web services. This architectural style is called a service-oriented architecture (SOA). The *Services Layer* of S2ST provides a functionality which may be accessed via a web services interface. The promise of SOA is that it enables the creation of applications by combining loosely coupled and interoperable services. Support for SOA is growing across a number of important application domains including those that have already adopted semantic web technology. We expect that support for semantic web and SOA technologies will be important requirements of future applications.

The S2ST architecture features a metadata model designed to resolve the conflict between the graph RDF data model and the relational data model. The metadata model defines RDF-to-Relational mappings, which capture how an RDF graph is stored in a

relational database. This makes it possible to develop generic algorithms for data and query mapping. All of this functionality can be exposed via web services to facilitate the integration of semantic web technology with existing applications. S2ST can support all schema mapping strategies used by existing systems and also future strategies as they become available. This flexibility is not available in any existing relational RDF database. Details of the metadata model are provided in Chapter 4.

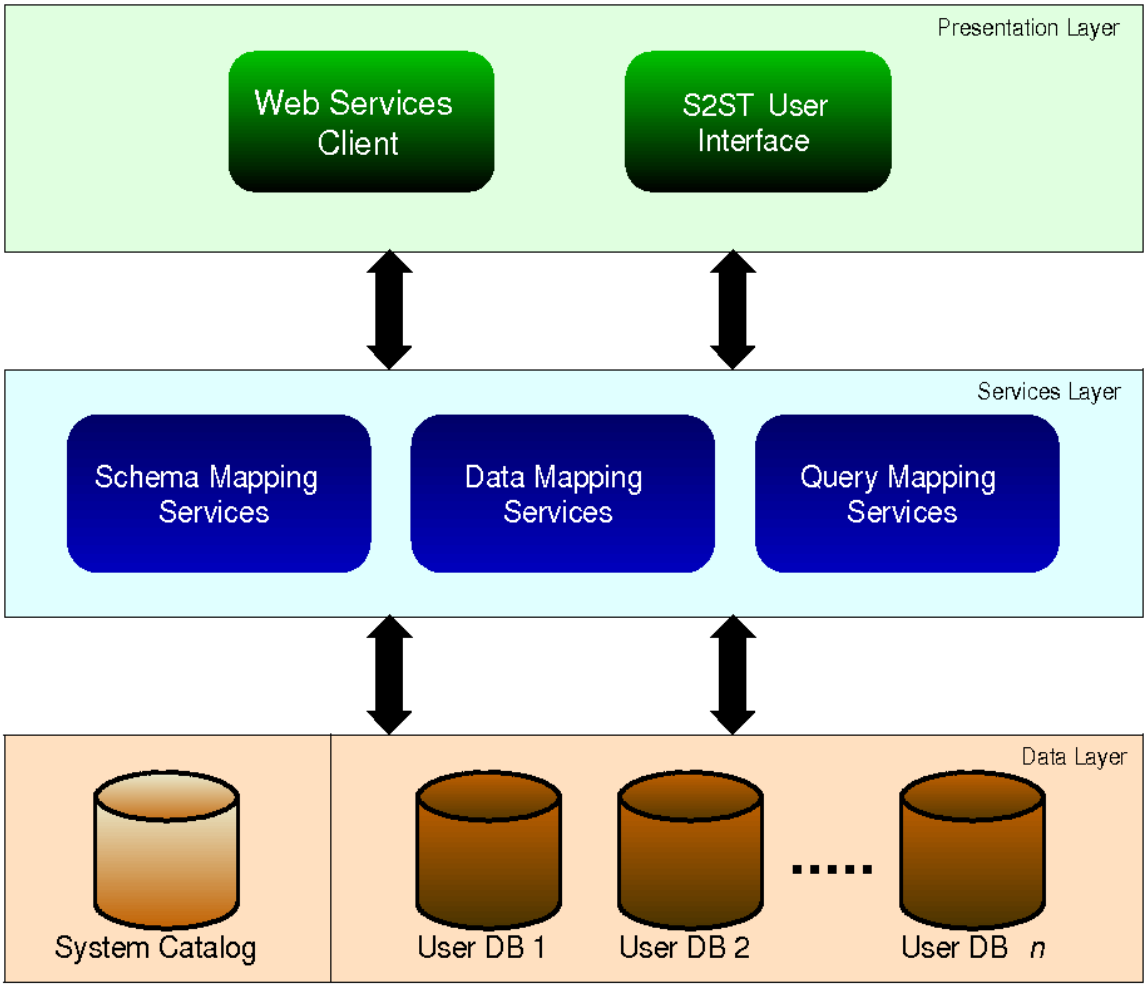

Figure 7: Service-Oriented Architecture of an RRDBMS

There are several architectural decisions that have strongly influenced the design and implementation of S2ST. The first decision has to do with the choice of software development platform. In the case of S2ST, the decision was made to build on the Java Enterprise Edition (Java EE) platform. This decision has many implications including availability of software components and tools as well as wide support within the academic and software development communities. The Java EE platform is a set of coordinated technologies for developing, deploying, and managing layered, server-centric applications. We have chosen to develop S2ST using an open-source web application framework (Grails) based on the Java EE platform and the Groovy programming language. Grails focus on coding by convention allowed us to quickly build the infrastructure for S2ST while eliminating most of the boilerplate code that is common in web applications. The Grails framework is based on the popular Model-View-Controller (MVC) architectural pattern. The MVC pattern separates domain logic (the Model) from input (the Controller) and presentation (the View). This facilitates independent development, testing and maintenance of each of these software components. The Groovy programming language is a dynamic language for the Java Virtual Machine (JVM). This means that applications built with Grails can be written in Groovy, Java or a mixture of both. S2ST includes code written in both Java and Groovy. All of the View and Controller code is written in Groovy. The performance-critical components of the Model are written in Java and the remaining components are written in Groovy.

The second architectural decision has to do with providing support for all of the most popular database management systems. Most of the existing systems that we studied provide support for only a limited number of them. Consider two of the best known

existing systems, Sesame and Jena. Sesame currently supports only 4 databases (PostgreSQL, MySQL, Microsoft SQL Server and Oracle). If your organization wants to use DB2, Sesame is not an option. Jena, which supports more databases than most existing systems, currently supports Oracle, Microsoft SQL Server, DB2, PostgreSQL, and MySQL. If your organization wants to use Sybase Adaptive Server, Jena is not an option. S2ST avoids this problem by taking advantage of an Object Relational Mapping framework known as Hibernate. We avoid writing vendor-specific SQL by relying on Hibernate to generate all DDL statements. We do anticipate the development of some vendor-specific query optimizations in the future. Even without these optimizations, S2ST can execute queries on all the popular databases by generating ANSI-compliant SELECT statements.

CHAPTER IV

DESIGN AND IMPLEMENTATION OF S2ST

Schema Mapping Services

As mentioned in Chapter 3, the architecture of S2ST is based on a unique metadata model. This model is what separates S2ST from existing systems. Figure 8 is a UML class diagram of the S2ST metadata model. The model is populated during schema mapping. Chapter 5 explains the different strategies employed by existing systems. It is clear that there is no 'one size fits all' schema mapping strategy. Which strategy should be used depends on a number of factors including the size and shape of the ontology, the amount of data being stored and the types of queries that will be executed. For the purpose of this chapter we refer to the four categories of schema mapping strategies listed in Chapter 5: schema oblivious, schema-aware, data-driven and hybrid. The first step in schema mapping is the creation of a Logical Schema, which is the central object in the metadata model.

During schema mapping, one or more Relations are created which have a set of Attributes and associated Triple Patterns. A Relation is a logical representation of a table which will be realized in a specific RDBMS. An Attribute is a logical representation of a column which will be realized in a specific table. As mentioned previously, RDF is based on subject-predicate-object expressions, called triples. Triple Patterns are specialized triples whose subject, predicate and object expressions can be used as variables. Triple

Patterns are used in SPARQL queries to specify the matching of triples. It is worth mentioning that there is a many-to-many relationship between Triple Patterns and Relations in the S2ST metadata model. This provides significant flexibility in specifying which triples are stored in particular Relations.

Once a Logical Schema has been created, one or more associated Physical Schemas can be created. A Physical Schema represents an instance of a Logical Schema on a specific database management system (e.g. Oracle 10g). S2ST creates an actual relational schema during the creation of a Physical Schema. The relational schema created in the target database contains a subset of the S2ST system catalog as well as tables for storing triples. The actual schema created depends on the details provided by the user during creation of the Logical Schema. For example, if the user chooses a Schema-Oblivious schema mapping strategy, then only a single table is created for storing all triples. If the user chooses a Schema-Aware schema mapping strategy, then tables would be created for each Class and Property found in the ontology specified during creation of the Logical Schema. S2ST even provides the user with the ability to exclude any of the Classes or Properties found in the specified ontology. This means that the user controls not only what data gets stored but also how it is stored. No other system provides this level of customization. Namespaces represent the prefixes found by S2ST when parsing the ontology specified during creation of the Logical Schema. They are used during schema and data mapping to disambiguate the names of classes and properties. Transactions represent data mapping operations performed on a specific Physical Schema. This allows S2ST to track the source of triples being stored and also gives us the option of implementing a delete feature at some point in the future.

Logical Schema, Relation, Attribute, Triple Pattern, Namespace, Physical Schema and Transaction are domain objects implemented as Java classes. Instances of these classes are persisted using Hibernate, which requires mapping metadata. Hibernate supports two ways of providing mapping metadata: annotations or XML. We chose to use XML. There is a separate XML mapping file for each domain class. Hibernate uses the mapping metadata to generate DDL statements needed to create the relational tables for storing instances of domain classes. Hibernate is able to generate platform-specific DDL statements for the target database management system.

The code for schema mapping consists of 6 views, 1 controller class and 1 service class. The schema mapping views correspond to specific use cases: View All Logical Schemas, View Logical Schema, Create Logical Schema, Create Physical Schema, Load Physical Schema, and Query Physical Schema. These views are implemented using Groovy Server Pages (GSP), which is the view technology bundled with the Grails framework. The schema mapping controller class, Schema Controller, is implemented as a Grails controller, which is written in Groovy. A Grails controller handles all HTTP requests from clients for a specific web context (URL pattern). It is common for code in a Grails controller to invoke methods in one or more Grails service classes.

The schema mapping service class, SchemaMappingService, is implemented as a Grails service and is also written in Groovy. Service classes encapsulate operations that deal with domain objects and are often transactional.

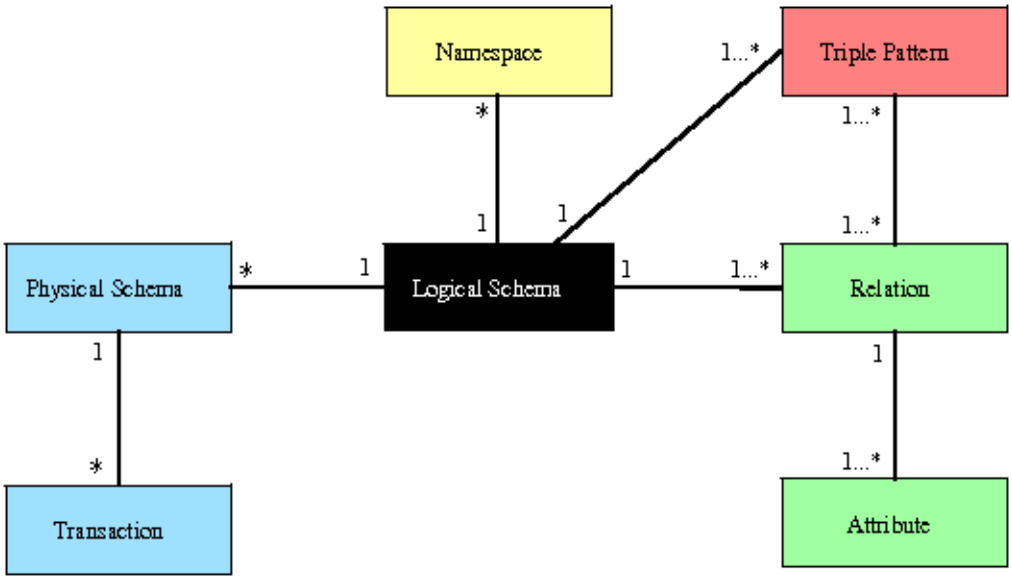

Figure 8: S2ST Metadata Model

Data Mapping Services

The data mapping services provided by S2ST allow users to load data into physical schemas created by S2ST. The purpose of data mapping is to shred triples in the source datastream and insert them into a relational database. Figure 9 below shows the architecture of our data mapping service. The shredding of triples is done by the Parser component. Most RDF database systems support multiple input formats including RDF/XML, N-Triple, N3, and Turtle. Rather than writing our own parser, we made the decision to use the ARP parser which is part of the Jena Semantic Web Framework. The Mapper component is responsible for shredding the triples produced by the Parser and feeding them as tuples to Loader components. A Loader component simply inserts the tuples into a specific relational table.

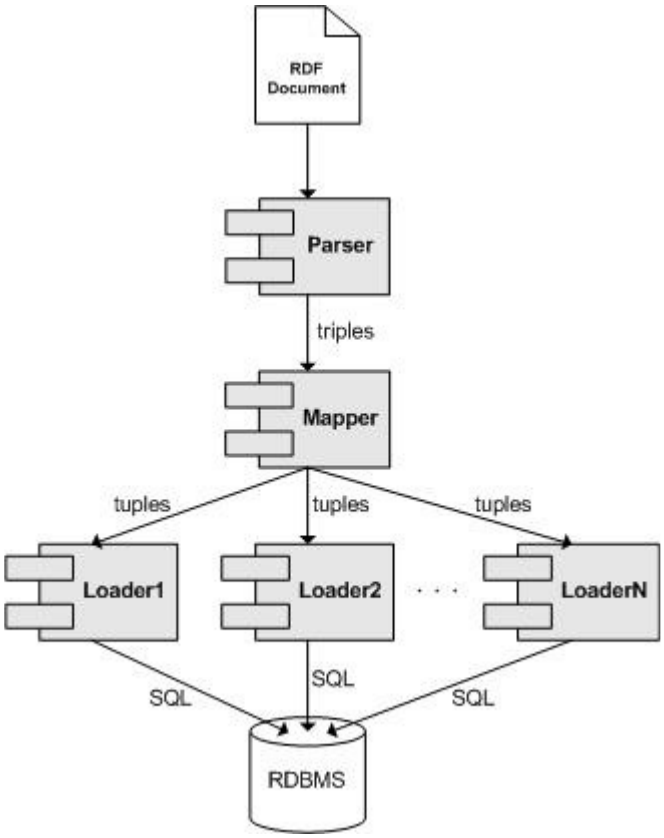

Figure 9: Data Mapping Architecture of S2ST

Any schema mapping strategy based on fixed relations (schema-oblivious, schema-aware and possibly hybrid) results in static data loaders being used to insert data into the appropriate tables. As mentioned previously, there are schema mapping strategies (data-driven and possibly hybrid) that allow tables to be created on-the-fly based on patterns found in the source data. These strategies result in dynamic data loaders being used to insert data into newly created tables.

Figure 10 below is a UML sequence diagram that shows the interaction between the classes involved in data mapping. All that is required to use the ARP parser is a class which implements the StatementHandler interface (defined by Jena library). This interface contains only two overloaded methods which support the shredding of triples

produced by the parser. Our implementation of the StatementHandler interface is the JenaDataMapper class. When an instance of DataMapper is created, a hash table instance is populated with the names of relations as keys and DataLoader instances as values. In other words, there is an instance of DataLoader for each table in the physical schema. The DataMapper interface, which is also implemented by the JenaDataMapper class, contains three methods which are needed to perform data mapping operations. The primary method in this interface is mapTriple, which controls the data mapping process.

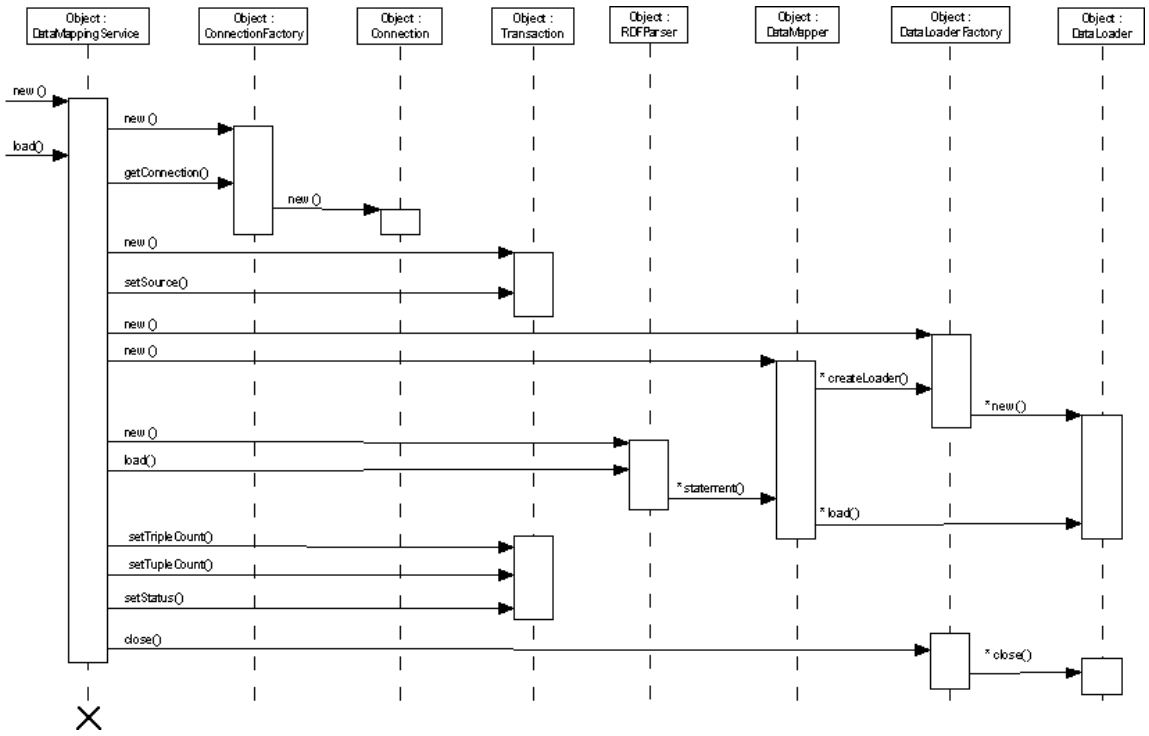

Figure 10: Data Mapping Sequence Diagram

The DataLoader interface contains three methods needed to perform data loading operations. The primary method in this interface is loadTriple. S2ST currently supports only one approach for inserting data into a relational database - JDBC batch loading. Another simpler, more generic approach is to produce a flat file containing SQL

statements (DML and possibly DDL) needed to insert the data. Unfortunately, this approach results in poor performance for data mapping operations on large datasets. Another approach is to use vendor-specific bulk loading facilities to insert the data. Bulk loading usually results in the best performance for data mapping operations on large datasets. Unfortunately, it requires vendor-specific input formats and programming interfaces. Our long term goal is to support all three of these approaches. We decided to implement JDBC batch loading first because it provides reasonable performance and is straightforward to implement for all popular database management systems.

One of the most challenging aspects of implementing data loading functionality is determining what data from each triple to persist. The schema mapping strategy prescribes what data is stored in each relational table in the target database. For example, if the table represents a class relation then only the subject is persisted. If the table represents a property relation then only the subject and object is persisted. S2ST can persist any combination of the subject, predicate and object in each table. It is common for Java applications to use the JDBC PreparedStatement to insert large numbers of rows into a relational table. Creating an instance of PreparedStatement requires a parameterized SQL statement. Once the application is ready to insert data, the parameters are initialized with appropriate values and then the addBatch method is called to add this SQL statement to a buffer for later execution. In the case of a data loader, the table name in the SQL INSERT statement depends on the triple being loaded. S2ST can use Java Unified Expression Language to dynamically generate table names based on patterns specified during schema creation and runtime values of triple properties.

Query Mapping Services

The W3C standard query language for RDF is SPARQL. To allow users to submit SPARQL queries and have them executed against a relational database requires translation to SQL, the standard query language for relational database management systems. S2ST is the first relational RDF database management system to feature semantics-preserving SPARQL-to-SQL query translation. This means that S2ST can guarantee the correctness of translated queries.

The first step in query translation is parsing of the user specified SPARQL query. Rather that writing our own SPARQL parser, we made the decision to use the one included in the Sesame RDF Framework. The Sesame SPARQL parser generates a parse tree containing all of the terms in the specified query. Like many parsers, the Sesame SPARQL parser relies on the Visitor software pattern. This pattern provides a way to separate an algorithm from the data structure it depends on. Sesame defines an interface, QueryModelVisitor, which contains more than 50 overloaded methods that must be implemented. Each of the overloaded methods is intended to support a specific type of node that may be encountered in the generated parse tree. Our QueryTranslator class contains an inner class which implements the QueryModelVisitor interface. Figure 11 is a UML sequence diagram that shows the interaction between the classes involved in query translation. Once the parse tree has been generated, an instance of this inner class uses postorder traversal to visit each node in the parse tree. As each node is visited, an instance of the TranslationObject class is created and pushed onto a stack.

The TranslationObject class is a simple class which contains an SQL query and a set of terms referenced in the query. The QueryTranslator class uses an instance of the

TranslationObjectBuilder class to perform most of the actual query translation. Once child nodes have pushed their results onto the stack, a parent node pops the TranslationObject instances created by the child nodes, creates another TranslationObject and pushes it onto the stack. This continues up the parse tree until the root node is reached. The root node is handled like all other parent nodes and it creates the final TranslationObject. S2ST currently supports a subset of the full SPARQL syntax. We expect to support the full SPARQL syntax at some point in the future.

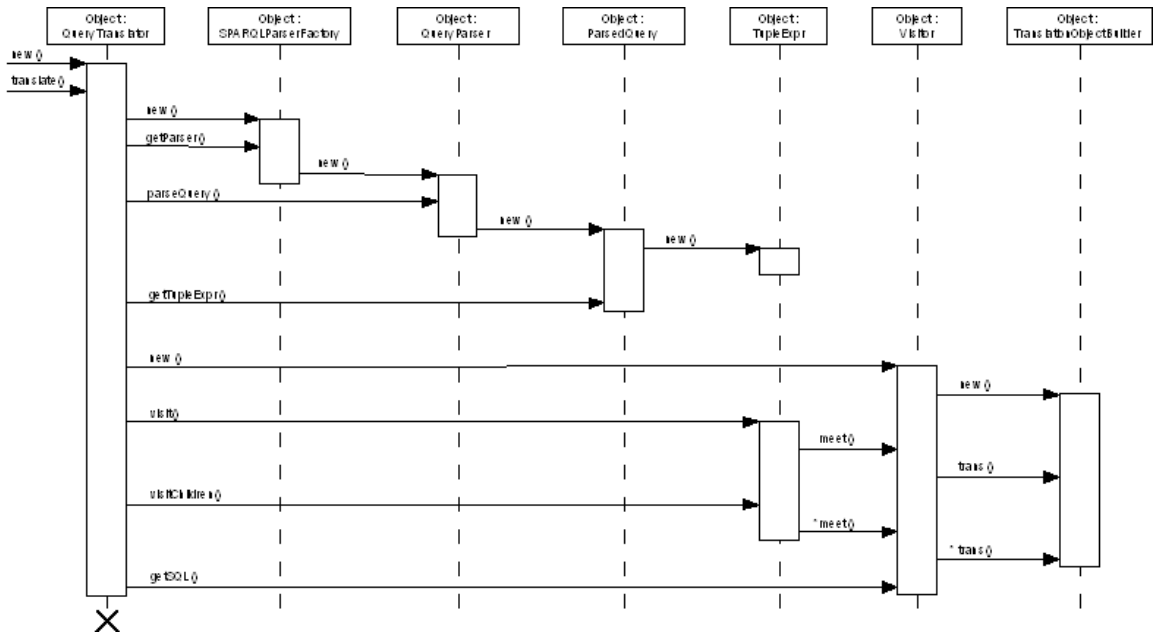

Figure 11: Query Mapping Sequence Diagram

The initial implementation of query translation in S2ST does not generate the simplest possible SQL. A. Chebotko et al. provide a number of simplifications that can be used to generate simpler and more efficient SQL queries. We expect to implement these simplifications in S2ST in the near future. Once a query has been translated, it must be executed against the relational database associated with the specified physical schema. This is done using a JDBC Statement object. Once the query is executed by the RDBMS,

the results are returned as a JDBC ResultSet object. The results are then encoded in XML to facilitate easy transformation into any of several different formats including CSV, HTML and PDF.

CHAPTER V

RELATED WORK

There has been considerable research done in the area of Semantic Web data. In this chapter, we review the research that is most closely related to the work we have done here. In recent years, several RDBMS-based RDF stores (see [4] for a survey) have been developed to support large-scale Semantic Web applications. The conflict between the graph RDF [22,23] data model and the target relational data model of such systems requires providing a way to deal with various mappings between the two data models, such as schema mapping, data mapping, and query mapping (a.k.a. query translation). Schema mapping is used to generate a relational database schema that can store RDF data. Schema mapping strategies employed by existing RDF stores fall into four categories:

Schema-oblivious (also called generic or vertical): A single relation, e.g., *Triple(s,p,o)*, is used to store RDF triples, such that attribute *s* stores the subject of a triple, *p* stores its predicate, and *o* stores its object. Schema-oblivious RDF stores include Jena [26, 27], Sesame [5], 3store [12,13], KAON [21], RStar [14], and OpenLink Virtuoso [10]. This approach has no concerns of RDF schema or ontology evolution, since it employs a generic database representation.

Schema-aware (also called specific or binary): This approach usually employs an RDF schema or ontology to generate so called property relations and class relations. A property relation, e.g., Property(s, o), is created for each property in an ontology and stores subjects s and objects o related by this property. A class relation, e.g., Class(i), is created for each class in an ontology and stores instances i of this class. An extension to the idea of property relations is a clustered property relation [25], e.g., Clustered(s, o_1, o_2, ... , o_n), which stores subjects s and objects o1, o2, ..., on related by n distinct properties (e.g., < s p1 o1 >, < s p2 o2 >, etc.). In [7], along with property and class relations, class-subject and class-object relations are introduced. A class-subject relation, e.g., ClassSubject(i, p, o) stores triples whose subjects are instances of a particular class in an ontology. Similarly, a class-object relation, e.g., ClassObject(s, p, i), stores triples whose objects are instances of a particular class. Such relations are useful for queries that retrieve all information about an instance (subject or object) of a particular class. Representatives of schema-aware RDF stores are Jena [25–27], DLDB [16], RDFSuite [3,20], DBOWL [15], PARKA [18], and RDFPROV [6,7]. Schema evolution for this approach is quite straightforward: the addition or deletion of a class/property in an ontology requires the addition or deletion of a relation (or relational tuples) in the database. More information on ontology evolution can be found in [19] and [11]. The schema-aware approach is in general yields better query performance than the schema oblivious approach as has been shown in several experimental studies [2, 3, 7, 20]. In addition, the use of a

column-oriented DBMS, in conjunction with vertical partitioning of relations, has shown further improvements in query performance [1].

Data-driven: This approach uses RDF data to generate database schema. For example, in [9], a database schema is generated based on patterns found in RDF data using data mining techniques. RDF store RDFBroker [17] implements signature relations, which are conceptually similar to clustered property relations, but are generated based on RDF data rather than RDF Schema information. In general, relations generated by the schema-aware approach can also be supported by the data-driven approach (e.g., property relations in Sesame [5] are created when their instances are first seen in an RDF document during data mapping). RDFBroker [17] reports improved in-memory query performance over Sesame and Jena for some test queries. Schema evolution for the data-driven approach, if supported, might be expensive.

Hybrid: This approach uses the mix of features of the previous approaches. An example of the hybrid database schema (resulted from schema-oblivious and schema-aware approaches) is presented in [20], where a schema-oblivious database representation, e.g., Triple(s, p, o), is partitioned into multiple relations based on the data type of object o, and a binary relation, e.g., Class(i, c), is introduced to store instances i of classes c. [20] reports comparable query performance of the hybrid and schema-aware approaches.

Data mapping is used to shred RDF triples into relational tuples and insert them into the database. Data mapping algorithms employed by existing RDF stores are usually

fairly straightforward, such that RDF triples are inserted into a single relation as in the schema-oblivious approach, or into one or multiple relations as in the other approaches. Several data mapping strategies and algorithms are presented in [7].

Query mapping is used to translate a SPARQL query into an equivalent SQL query, which is evaluated by the relational engine and the result is returned as a SPARQL query solution. This is one of the most difficult mappings in RDBMS-based RDF stores.

CHAPTER VI

CONCLUSION AND FUTURE WORK

In this paper, we established the theoretical foundations of a Relational RDF Database Management System (RRDBMS), we described an SOA architecture for exposing the services provided by an RRDBMS, and we detailed the implementation of S2ST, the first and only RRDBMS that supports multiple relational database management systems, user-customizable schema mappings, schema-independent data mapping, and semantics-preserving query translation.

We plan to extend our research in a number of areas including:

- Enhancements to our schema generation user interface to provide recommendations for creating optimal schemas based on ontology and amount of data being stored/queried
- Simplifications to improve the performance of generated SQL queries
- Access control that provides fine-grained security for all data and user accessible objects
- RSS feeds for tracking specific database activity
- Virtual machine images containing a fully configured S2ST system

REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web data management using vertical partitioning. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, pages 411–422, 2007.

[2] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *Proc. of the Internatonal Conference on Very Large Data Bases (VLDB)*, pages 149–158, 2001.

[3] S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. On storing voluminous RDF descriptions: The case of Web portal catalogs. In *Proc. of the International Workshop on the Web and Databases (WebDB)*, pages 43–48, 2001.

[4] D. Beckett and J. Grant. SWAD-Europe Deliverable 10.2: Mapping SemanticWeb data with RDBMSs. Technical report, 2003. Available from http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report.

[5] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 54–68, 2002.

[6] Chebotko, X. Fei, C. Lin, S. Lu, and F. Fotouhi. Storing and querying scientific workflow provenance metadata using an RDBMS. In *Proc. of the IEEE International Workshop on Scientific Workflows and Business Workflow Standards in e-Science*, pages 611–618, 2007.

[7] Chebotko, X. Fei, S. Lu, and F. Fotouhi. Scientific workflow provenance metadata management using an RDBMS-based RDF store. Technical Report TR-DB-092007-CFLF, Wayne State University, September 2007. Available from http://www.cs.wayne.edu/~artem/main/research/TR-DB-092007-CFLF.pdf.

[8] Chebotko, S. Lu, and F. Fotouhi. Semantics preserving SPARQL-to-SQL translation. *Data & Knowledge Engineering (DKE)*, 68(10):973–1000, 2009.

[9] L. Ding, K. Wilkinson, C. Sayers, and H. Kuno. Application specific schema design for storing large RDF datasets. *In Proc. of the International Workshop on Practical and Scalable Semantic Systems (PSSS)*, 2003.

[10] O. Erling. Implementing a SPARQL compliant RDF triple store using a SQL-ORDBMS. Technical report, OpenLink Software Virtuoso, 2001. Available from http://virtuoso.openlinksw.com/wiki/main/Main/VOSRDFWP.

[11] G. Flouris, D.Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou. Ontology change: classification and survey. *Knowledge Engineering Review*, 23(2), 2008.

[12] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *Proc. of the International Workshop on Practical and Scalable Semantic Systems (PSSS)*, pages 1–15, 2003.

[13] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *Proc. of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, pages 235–244, 2005.

[14] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proc. of the International Conference on Information and Knowledge Management (CIKM)*, pages 484–491, 2004.

[15] S. Narayanan, T. M. Kurc, and J. H. Saltz. DBOWL: Towards extensional queries on a billion statements using relational databases. Technical Report OSUBMI TR 2006 n03, Ohio State University, 2006. Available from http://bmi.osu.edu/resources/techreports/osubmi.tr.2006.n3.pdf.

[16] Z. Pan and J. Heflin. DLDB: Extending relational databases to support Semantic Web queries. In *Proc. of the International Workshop on Practical and Scalable Semantic Web Systems (PSSS)*, pages 109–113, 2003.

[17] M. Sintek and M. Kiesel. RDFBroker: A signature-based high-performance RDF store. In *Proc. of the European Semantic Web Conference (ESWC)*, pages 363–377, 2006.

[18] K. Stoffel, M. G. Taylor, and J. A. Hendler. Efficient management of very large ontologies. In *Proc. of the American Association for Artificial Intelligence Conference (AAAI)*, pages 442–447, 1997.

[19] L. Stojanovic. Methods and Tools for Ontology Evolution. Ph.D. Dissertation, University of Karlsruhe, Germany, 2004. Available from http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1241.

[20] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 685–701, 2005.

[21] R. Volz, D. Oberle, B. Motik, and S. Staab. KAON SERVER - a Semantic Web management system. In *Proc. of the International World Wide Web Conference (WWW), Alternate Tracks - Practice and Experience*, 2003.

[22] W3C. RDF Primer.W3C Recommendation, 10 February 2004. F. Manola and E. Miller (Eds.). 2004. Available from http://www.w3.org/TR/rdf-primer/.

[23] W3C. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. G. Klyne, J. J. Carroll, and B. McBride (Eds.). 2004. Available from http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[24] W3C. SPARQL Query Language for RDF. W3C Recommendation, 15 January 2008. E. Prud'hommeaux and A. Seaborne (Eds.). 2008. Available from http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[25] K. Wilkinson. Jena property table implementation. In *Proc. of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2006.

[26] K.Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. of the International Workshop on Semantic Web and Databases (SWDB)*, pages 131–150, 2003.

[27] K. Wilkinson, C. Sayers, H. A. Kuno, D. Reynolds, and L. Ding. Supporting scalable, persistent Semantic Web applications. *IEEE Data Eng. Bull.*, 26(4):33–39, 2003.

BIOGRAPHICAL SKETCH

Anthony T. Piazza earned a Master of Science in Computer Science from the University of Texas – Pan American in December 2009. He earned a Bachelor of Science in Electrical Engineering from Texas A&M University – Kingsville in 1987. He has over 22 years of experience in software development including design, development, testing, training and consulting. He holds a number of industry certifications. He has done consulting work for many companies, including; eBay, BMC Software, Wright Express, Texas Department of Family and Protective Services, Wisconsin Electric Power, Metavante, Trane Corporation, J.C. Penney Company, Concordia Publishing House, Missouri Chamber of Commerce, Edward Jones, Ralston Purina and IBM. He has also delivered technical training courses for many companies, including; SRA International, BlueCross BlueShield of North Dakota, IBM, U.S. Department of Treasury, eLoyalty, Chicago Tribune, Cash America International, Discover Financial Services, TDS Telecom, Highmark Blue Cross Blue Shield and the City of Phoenix (Arizona). His permanent mailing address is 1308 Palm Drive, Kingsville, TX 78363.