8-2011

Benchmarking Bottom-Up and Top-Down Strategies to Sparql-To-Sql Query Translation

Andrii Kashliev
University of Texas-Pan American

BENCHMARKING BOTTOM-UP AND TOP-DOWN STRATEGIES TO SPARQL-TO-SQL QUERY TRANSLATION

A Thesis

by

ANDRII KASHLIEV

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

August 2011

Major Subject: Computer Science

BENCHMARKING BOTTOM-UP AND TOP-DOWN STRATEGIES TO SPARQL-TO-SQL QUERY TRANSLATION

A Thesis
by
ANDRII KASHLIEV

COMMITTEE MEMBERS

Dr. Artem Chebotko
Chair of Committee

Dr. Zhixiang Chen
Committee Member

Dr. Pearl Brazier
Committee Member

Dr. Christine Reilly
Committee Member

August 2011

ABSTRACT

Kashliev, Andrii, Benchmarking Bottom-Up And Top-Down Strategies to SPARQL-to-SQL Query Translation. Master of Science (MS), August, 2011, 40 pp., 2 tables, 7 illustrations, references, 44 titles.

Many researchers have proposed using conventional relational databases to store and query large Semantic Web datasets. The most complex component of this approach is SPARQL-to-SQL query translation. Existing algorithms perform this translation using either bottom-up or top-down strategy and result in semantically equivalent but syntactically different relational queries. Do relational query optimizers always produce identical query execution plans for semantically equivalent bottom-up and top-down queries? Which of the two strategies yields faster SQL queries? To address these questions, this work studies bottom-up and top-down translations of SPARQL queries with nested optional graph patterns. This work presents: (1) A basic graph pattern translation algorithm that yields flat SQL queries, (2) A bottom-up nested optional graph pattern translation algorithm, (3) A top-down nested optional graph pattern translation algorithm, and (4) A performance study featuring SPARQL queries with nested optional graph patterns over RDF databases created in Oracle, DB2, and PostgreSQL.

DEDICATION

The completion of this thesis would not have been possible without the love and support of my family. My mother, Olga Sheliakina, my father, Sergiy Kashlyev, and my sister, Anya, wholeheartedly inspired, motivated and supported me by all means to accomplish this degree. Thank you for your love and patience.

ACKNOWLEDGEMENTS

First, I want to thank God for His love and kindness, for His guidance in this research, and for the privilege He has bestowed upon me to work with Dr. Artem Chebotko. “Therefore I will praise you among the nations, O LORD; I will sing praises to your name.” (Psalms 18:49). Second, I want to thank my parents for their patience and support, both emotional and financial throughout my study at UTPA. Third, I want to thank Dr. Artem Chebotko for allowing me to work with him. Finally, I want to thank all the graduate students that worked with me for all their help. Especially, I want to thank Tony Piazza for all the help and encouragement I received from him during my first year at UTPA.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

We live in a time when the amount of information published on the Web grows at an unprecedented pace. One of the challenges imposed by this growth is that most of the data available on the Web is machine-readable but not machine-understandable. That is, it lacks semantics, or meaning, that could be interpreted by machines and automated agents. 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, the W3C 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.

Many researchers have proposed using conventional relational databases to store and query large Semantic Web datasets [1]. Emerged systems, called *relational RDF databases*, share a common design pattern that uses a schema mapping algorithm to generate a relational database schema, a data mapping algorithm to insert RDF data into the database, and a query mapping algorithm to translate RDF queries into equivalent SQL queries. SPARQL-to-SQL translation is not only the most complex mapping in a relational RDF database, but also very critical to overall querying performance. Existing algorithms translate SPARQL queries to SQL using either bottom-up or top-down strategy and result in semantically equivalent but syntactically different relational queries.

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. These systems can be built using conventional relational databases to store and query large Semantic Web datasets. The most complex component of this approach is SPARQL-to-SQL query translation. Existing algorithms translate SPARQL queries to SQL using either bottom-up or top-down strategy and result in semantically equivalent but syntactically different relational queries. While it can be expected that relational query optimizers produce identical query execution plans for semantically equivalent bottom-up and top-down queries, is this usually the case in practice? And if not, which strategy yields faster

SQL queries? Our motivation is to answer these questions by studying bottom-up and top-down translations of SPARQL queries with nested optional graph patterns.

Research Contributions

Our research contributions are: (1) a basic graph pattern translation algorithm that results in flat SQL, (2) a bottom-up nested optional graph pattern translation algorithm, (3) a top-down nested optional graph pattern translation algorithm, and (4) a performance study featuring SPARQL queries with nested optional graph patterns over RDF databases instantiated in Oracle, DB2, and PostgreSQL.

Organization of this Document

The remaining chapters of this document are organized as follows: Chapter 2 lists a number of preliminary definitions, Chapter 3 presents our algorithms for basic graph pattern translation, bottom-up nested optional graph pattern translation and top-down nested optional graph pattern translation, Chapter 4 reports our performance study, 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. In this chapter we discuss some of the most important terms related to semantic web technology.

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.

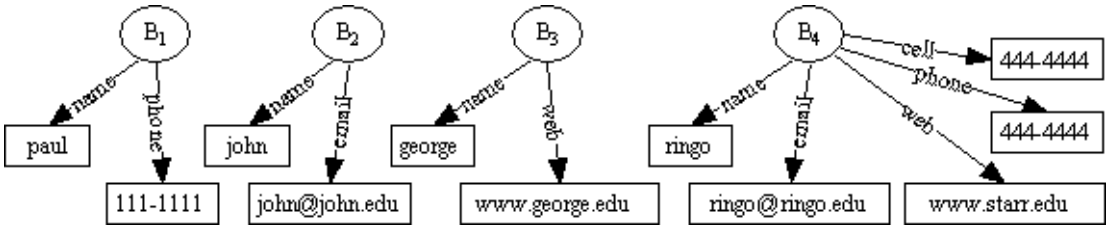


Figure 1: Sample RDF Graph

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 (Basic graph pattern)

A basic graph pattern *bgp* is a set of triple patterns $\{tp_1, tp_2, ..., tp_{n-1}, tp_n\}$, also denoted as tp_1 *AND* tp_2 *AND ... AND* tp_{n-1} *AND* tp_n, where *AND* is a binary operator that corresponds to the conjunction in SPARQL and *n* is the number of triple patterns in *bgp*.

Definition 4 (Nested optional graph pattern)

A nested optional graph pattern *nogp* has the form bgp_1 *OPT*{ bgp_2 *OPT*{ … {bgp_{n-1} *OPT*{ bgp_n}} ... }}, where *OPT* corresponds to the *OPTIONAL* construct in SPARQL, curly braces {}

denote nesting of graph patterns, and $n \geq 3$ represents the number of basic graph patterns in *nogp*.

Definition 5 (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 6 (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 7 (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 8 (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 9 (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 10 (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 7, δ is a mapping as in Definition 8, and τ is a mapping as in Definition 9. 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 11 (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 12 (Mapping α)

Given a set of all possible triple patterns $TP = (IVL) \times (IV) \times (IVL)$ and a set of relations *REL* in a relational RDF database, a mapping α is a many-to-many mapping $\alpha : TP \rightarrow REL$, if given a triple pattern $tp \in TP$, $\alpha(tp)$ is a relation in which all the triples that may match *tp* are stored.

Definition 13 (Mapping β)

Given a set of all possible triple patterns $TP = (IVL) \times (IV) \times (IVL)$, a set of triple $POS = \{sub, pre, obj\}$, and a set of relational attributes *ATR* in a relational RDF database, a mapping β is a many-to-one mapping $\beta : TP \times POS \rightarrow ATR$, if given a triple pattern $tp \in TP$ and a position $pos \in POS$, $\beta(tp, pos)$ is a relational attribute whose value may match *tp* at position *pos*.

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

Definition 14 (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 7, δ is a mapping as in Definition 8, τ is a mapping as in Definition 9 with the generic data types substituted by data types supported by *rdbms*, α is a mapping as in Definition 12, and β is a mapping as in Definition 13.

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 15 (Physical Schema)

We can derive a physical schema based on the mappings in Example 11. 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 determine which tables and attributes will be used.

α(*B*$_1$, *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 (*B*$_1$, *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.

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

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.

In addition, our translation uses three auxiliary functions: (1) a function *alias* that

generates a unique alias for a relation, (2) a function *vars* that returns a set of all variables in a graph pattern, and (3) a function *name* that generates a unique name for a variable in V, such that the generated name conforms to the SQL syntax for relational attribute names (e.g., a variable can be “renamed” by simply removing initial ‘?’ or’$’).

Finally, we assume that a basic graph pattern is comprised of at least one triple pattern and contains at least one variable. [2] and [3] describe how these corner cases can be handled.

CHAPTER III

SPARQL-TO-SQL TRANSLATION

In this chapter we present our basic graph pattern translation algorithm that yields flat SQL queries as well as our bottom-up nested optional graph pattern translation algorithm, and our top-down nested optional graph pattern translation algorithm.

Basic Graph Pattern Translation

A basic graph pattern, which is a set of triple patterns, is the main building block of SPARQL queries. While there exist both bottom-up and top-down strategies that generate equivalent SQL queries with nested subqueries, this section presents a simple algorithm that generates fully flat SQL queries. Therefore, with this algorithm in place, the order of translation becomes unimportant, since only a naive query optimizer does not consider inner join reordering.

Our basic graph pattern translation function BGPtoFlatSQL is shown in Figure 2. It translates a SPARQL basic graph pattern *bgp* that consists of a set of triple patterns tp_1, tp_2, ..., tp_n into an equivalent flat SQL query that can be executed by a relational database engine. BGPtoFlatSQL constructs *from*, *where*, and *select* clauses of an SQL query as follows. For each triple pattern tp_i in *bgp*, a unique table alias a_i is assigned and table $\alpha(tp_i)$ with this alias is appended to the *from* clause. The algorithm then computes an inverted index on all variables in *bgp*, such that each distinct variable is associated with

```
1: function BGPtoFlatSQL
2: input: basic graph pattern bgp; mappings α and β; functions alias,
   vars, and name
3: output: flat SQL query
4: Let bgp = {tp_1, tp_2, ..., tp_{n-1}, tp_n} and n ≥ 1
5: select = ""; from = ""; where = ""
6: Assign a unique alias a_i = alias() to each tp_i ∈ bgp
7: //Construct the SQL From clause:
8: for each tp_i ∈ bgp do
9:     from += "$α(tp_i) $a_i, "
10: end for
11: //Construct an inverted index (hash) h on variables in bgp:
12: for each tp_i ∈ bgp do
13:     for each variable ?v found in tp_i do
14:         h(?v) = h(?v) ∪ {"$a_i.$β(tp_i, position of ?v in tp_i)"}
15:     end for
16: end for
17: //Construct the SQL Where clause:
18: for each tp_i ∈ bgp do
19:     for each instance or literal l (l ∈ IL) found in tp_i do
20:         where += "$a_i.$β(tp_i,position of l in tp_i) = '$l' And "
21:     end for
22: end for
23: for each variable ?v ∈ vars(bgp) and |h(?v)| > 1 do
24:     Let x ∈ h(?v)
25:     for each y ∈ h(?v) and y ≠ x do
26:         where += "$x = $y And "
27:     end for
28: end for
29: //Construct the SQL Select clause:
30: for each variable ?v ∈ vars(bgp) do
31:     Let x ∈ h(?v)
32:     select += "$x As $name(?v), "
33: end for
34: return "Select $select From $from Where $where"
35: end function
```

Figure 2: Function BGPtoFlatSQL

attributes in the respective tables from the *from* clause. The corresponding attribute names for variables are computed using mapping *β*. The *where* clause is first constructed to ensure that any non-variables in *bgp* are restricted to their values (e.g., literals or identifiers). In particular, each relational attribute that corresponds to a literal or IRI in *bgp* must contain this literal or IRI value. The inverted index is then used to append join conditions into the *where* clause, such that all attributes that correspond to the same variable must be equal. Finally, the *select* clause is generated to include attributes that correspond to every distinct variable in *bgp*, with attributes being renamed using auxiliary function *name()*. A flat SQL query constructed with computed

select, *from*, and *where* clauses is returned.

This algorithm is used by our bottom-up and top-down translation algorithms for nested optional graph patterns described in the following sections.

Bottom-Up Nested Optional Graph Pattern Translation

The bottom-up approach to SPARQL-to-SQL query translation is well-studied in the literature [2] and implemented in many relational RDF databases. This section presents an algorithm that implements one of our translation rules described in [2]. It should be noted that, while this work assumes that nested *OPTIONAL* clauses contain basic graph patterns, which is sufficient for our study, in the general case, other graph patterns, such as sequential optional graph patterns and alternative graph patterns, are possible. The algorithm uses the translation rule for the general case with an additional simplification that eliminates the call of the `Coalesce` function for some attributes in projection lists. The use of `Coalesce` is redundant with only basic graph patterns assumed in *OPTIONAL* clauses; however, other simplifications on join conditions are not applied.

Our bottom-up translation function NOGPtoSQL-BU is shown in Figure 3. It visits each basic graph pattern in a SPARQL nested optional graph pattern *nogp* starting from bgp_n and going up to bgp_1. Each basic graph pattern is translated to SQL using function BGPtoFlatSQL producing a flat SQL query. During the first loop iteration, the translation of bgp_n is assigned to variable *sql* and the translation of bgp_{n-1} is assigned to variables sql_i. A new SQL query that computes a left outer join between virtual relations

```
1: function NOGPtoSQL-BU
2: input: nested optional graph pattern nogp; mappings α and β;
   functions alias, vars, and name
3: output: bottom-up SQL query
4: Let nogp = bgp_1 OPT{ bgp_2 OPT{ ··· {bgp_{n-1} OPT{ bgp_n}}
   ···}} and n ≥ 3
5: //Construct a bottom-up SQL query:
6: sql = BGPtoFlatSQL(bgp_n, α, β, alias, vars, name)
7: for i = n − 1; i ≤ 1; i = i − 1 do
8:    //Construct the SQL From clause:
9:    sql_i = BGPtoFlatSQL(bgp_i, α, β, alias, vars, name)
10:   a_1 = alias(); a_2 = alias()
11:   from = "($sql_i) $a_1 Left Outer Join ($sql) $a_2"
12:   //Construct a join condition:
13:   cond = "True "
14:   for each relational attribute ra that appears in the Select clause
      of both sql_i and sql do
15:      cond += "And ( $a_1.$ra = $a_2.$ra Or $a_1.$ra Is
         Null Or $a_2.$ra Is Null )"
16:   end for
17:   //Construct the SQL Select clause:
18:   select = ""
19:   for each relational attribute ra that appears in the Select clause
      of sql_i do
20:      select += "$a_1.$ra As $ra,"
21:   end for
22:   for each relational attribute ra that appears in the Select clause
      of sql but not sql_i do
23:      select += "$a_2.$ra As $ra,"
24:   end for
25:   sql = "Select $select From $from On ($cond)"
26: end for
27: return sql
28: end function
```

Figure 3: Function NOGPtoSQL-BU

sql$_i$ and *sql* is constructed. This query contains: " ($*sql*$_i$) $*a*$_1$ `Left Outer Join` ($*sql*) $a2" in its `From` clause, where a_1 and a_2 are unique aliases; a join condition "$*a*$_1$.$*ra* = $*a*$_2$.$*ra* `Or` $*a*$_1$.$*ra* `Is Null Or` $*a*$_2$.$*ra* `Is Null`" in its `On` clause, which requires common relational attributes in a_1 and a_2 to be equal or one of them to be `Null`; and a projection list in its `Select` clause of all attributes in a_1 and all other unique attributes in a_2. This newly constructed query is assigned to variable *sql*, overwriting its previous value. The following loop iteration repeats the procedure but with a new value of *sql* as previously described and a new value of *sql*$_i$ that now holds the translation of *bgp*$_{n-2}$. After the final iteration, a value of *sql* represents a fully generated query and is returned.

Top-Down Nested Optional Graph Pattern Translation

One of the first top-down SPARQL-to-SQL query translations found in the literature is described in our unpublished report [3]. This section summarizes our solution for the case when only basic graph patterns are used in *OPTIONAL* clauses.

Our top-down translation function NOGPtoSQL-TD is shown in Figure 4.

The logic of this algorithm is similar to the logic described for NOGPtoSQL-BU. One obvious difference is that function NOGPtoSQL-TD visits each basic graph pattern in a SPARQL nested optional graph pattern *nogp* starting from bgp_1 and going down to bgp_n.

1: **function** NOGPtoSQL-TD
2: **input:** well-designed nested optional graph pattern $nogp$; mappings α and β; functions $alias$, $vars$, and $name$
3: **output:** top-down SQL query
4: Let $nogp = bgp_1\ OPT\{\ bgp_2\ OPT\{\ \cdots\ \{bgp_{n-1}\ OPT\{\ bgp_n\}\}\ \cdots\}\}$ and $n \geq 3$
5: //Construct a top-down SQL query:
6: sql = BGPtoFlatSQL($bgp_1, \alpha, \beta, alias, vars, name$)
7: **for** $i = 2;\ i \leq n;\ i = i + 1$ **do**
8: //Construct the SQL `From` clause:
9: sql_i = BGPtoFlatSQL($bgp_i, \alpha, \beta, alias, vars, name$)
10: $a_1 = alias()$; $a_2 = alias()$
11: $from$ = "($sql) $a_1 Left Outer Join ($sql_i) $a_2"
12: //Construct a join condition:
13: Let v be a relational attribute that (1) appears in the `Select` clause of sql, (2) $v = name(?v)$ corresponds to a variable $?v$, and (3) variable $?v \in vars(bgp_{i-1}) - (vars(bgp_1) \cup vars(bgp_2) \cup \cdots \cup vars(bgp_{i-2}))$ first occurs in bgp_{i-1} but not in $bgp_1, bgp_2, \cdots, bgp_{i-2}$. If sql has no attribute that satisfies these conditions, a dummy attribute must be introduced as discussed in [4].
14: $cond$ = "$v Is Not Null"
15: **for** each relational attribute ra that appears in the `Select` clause of both sql and sql_i **do**
16: $cond$ += " And $a_1.$ra = $a_2.$ra"
17: **end for**
18: //Construct the SQL `Select` clause:
19: $select$ = ""
20: **for** each relational attribute ra that appears in the `Select` clause of sql **do**
21: $select$ += "$a_1.$ra As $ra,"
22: **end for**
23: **for** each relational attribute ra that appears in the `Select` clause of sql_i but not sql **do**
24: $select$ += "$a_2.$ra As $ra,"
25: **end for**
26: sql = "Select $select From $from On ($cond)"
27: **end for**
28: **return** sql
29: **end function**

Figure 4: Function NOGPtoSQL-TD

The other difference lies in how a join condition is generated. It encodes the following semantics: before a nested optional graph pattern can succeed, all containing optional graph patterns must have succeeded. Therefore, a join condition must check that a basic graph pattern in a containing *OPTIONAL* clause has a solution. This is achieved via a `Not Null` check on a relational attribute with special properties: this attribute must appear in the `Select` clause of *sql*, since the translation of the containing graph pattern is part of *sql*, and it must correspond to a variable that first occurred in a basic graph pattern of the containing *OPTIONAL* clause and not in any preceding basic graph pattern. If such an attribute is not readily available, a new attribute for a "dummy" variable can be introduced in a basic graph pattern to perform the check. Further details on this solution can be found in [3].

CHAPTER IV

PERFORMANCE STUDY

This chapter reports our query performance study conducted using the WordNet dataset and test SPARQL queries that were translated to SQL using the proposed bottom-up and top-down query translation algorithms and evaluated in three relational database management systems.

Experimental Setup

The experiments were conducted on a server with two 2GHz Intel Xeon E5504 Nehalem CPUs, 32GB RAM and 6TB disk array running Ubuntu 9.02 Jaunty x64. Three different database management systems, namely Oracle 10.2 Express Edition, DB2 9.7 Express-C and PostgreSQL 8.3.12, were installed on the server.

Our algorithms were implemented in Java 6 within the S2ST system; generic schema and data mapping algorithms supported by S2ST were used to generate identical database schemas in Oracle, DB2 and PostgreSQL, and to store the RDF dataset into the databases, respectively.

Dataset and Test Queries

The OWL representation of WordNet was chosen for our experiments. WordNet is a lexical database for the English language, which organizes English words into synonym sets according to part of speech (e.g. noun, verb, etc.) and enumerates linguistic relations between

these sets. In the *WordNet.OWL*, each part of speech is modeled as an *owl:Class*, and each linguistic relation is modeled as an *owl:ObjectProperty*, *owl:DatatypeProperty*, *owl:TransitiveProperty*, or *owl:SymmetricProperty*. The simplified WordNet ontology is illustrated in Figure 5.

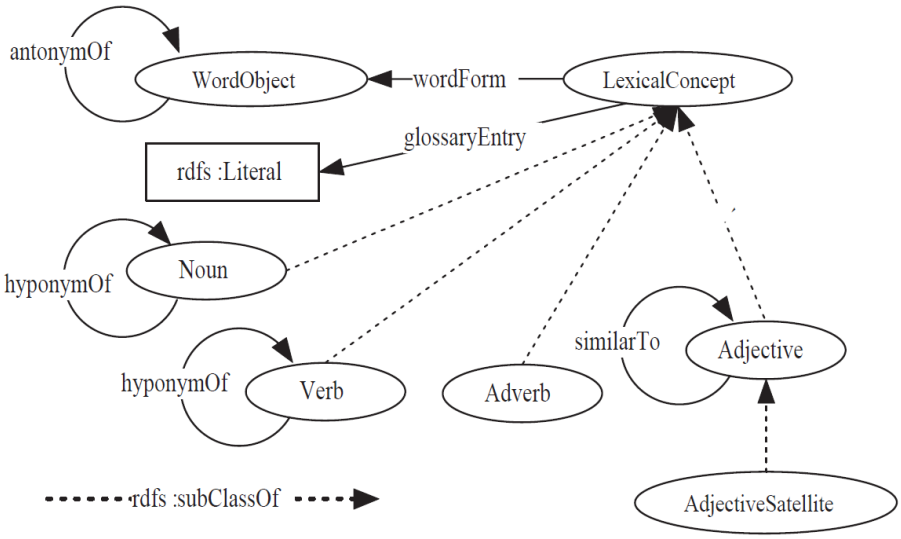


Figure 5: Classes and properties of the WordNet ontology

Table I
PROPERTIES AND RESOURCES IN WORDNET 1.2

Property	Count
type	251,726
wordForm	195,802
glossaryEntry	111,223
hyponymOf	90,267
similarTo	22,494
antonymOf	7,115
Others	36,225
Total	714,852

Resource	Count
WordObject	140,470
Noun	75,804
Verb	13,214
AdjectiveSatellite	11,231
Adjective	7,345
Adverb	3,629
Others	33
Total	251,726

The figure does not include some classes (e.g., *wn:Nouns_and_Verbs*) and properties (e.g., *wn:mMeronym*) that are not essential for the understanding of the dataset and the experiments. The relevant statistics for the WordNet dataset is shown in Table I. For example, *WordNet.OWL* contains 251,726 triples involving `rdf:type` as the predicate, and 140,470 of them have

wn:WordObject as the object.

Table II shows 22 SPARQL queries over the WordNet dataset that were carefully selected for our experiments.

Table II
TEST SPARQL QUERIES

Q#	SPARQL
Q1	W{?a rdf:type :Adjective O{?a :wordForm ?c O{?a :glossaryEntry ?b}}}
Q2	W{?a rdf:type :Adjective O{?a :glossaryEntry ?b O{?a :wordForm ?c}}}
Q3	W{?a :wordForm ?c O{?a rdf:type :Adjective O{?a :glossaryEntry ?b}}}
Q4	W{?a :glossaryEntry ?b O{?a rdf:type :Adjective O{?a :wordForm ?c}}}
Q5	W{?a :wordForm ?c O{?a :glossaryEntry ?b O{?a rdf:type :Adjective}}}
Q6	W{?a :glossaryEntry ?b O{?a :wordForm ?c O{?a rdf:type :Adjective}}}
Q7	W{?n1 :hyponymOf ?n2 O{?n2 :hyponymOf ?n3 O{?n3 :hyponymOf ?n4}}}
Q8	W{?n1 :hyponymOf ?n2 O{?n2 :hyponymOf ?n3 O{?n3 :hyponymOf ?n4 O{?n4 :hyponymOf ?n5 O{?n5 :hyponymOf ?n6 O{?n6 :hyponymOf ?n7}}}}}}
Q1′-Q6′	Same as respective queries Q1 - Q6 but with one variable in the first triple pattern (the W clause) restricted to a URI or literal
Q1″-Q6″	Same as respective queries Q1 - Q6 but with one variable in the second triple pattern (the first O clause) restricted to a URI or literal
Q7′	Same as Q7 but with ?n1 and ?n4 restricted to URIs
Q8′	Same as Q8 but with ?n1 and ?n7 restricted to URIs

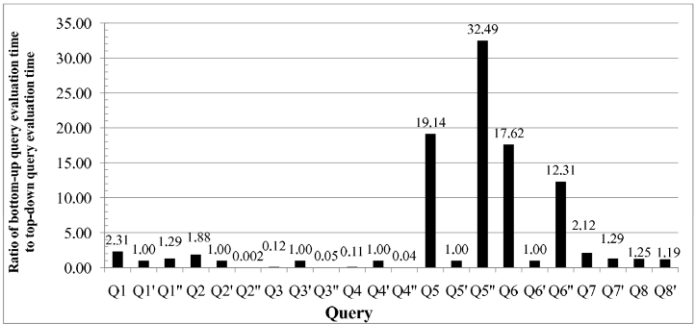
In the table, W stands for *WHERE* and O stands for *OPTIONAL*; the SPARQL SELECT clause is omitted for brevity, and the projection includes all distinct variables of a query. Queries Q1-Q6 are constructed as all possible permutations of the three triple patterns occurring outside and inside *OPTIONAL* clauses. These queries have one nested *OPTIONAL* clause. Queries Q1′-Q6′ and Q1″-Q6″ are obtained from respective queries Q1-Q6 by restricting variable values in the first and second triple patterns, respectively. The rationale for such restrictions is to reduce cardinalities of intermediate relations resulting from first left outer joins in the queries. In particular, in terms of the intermediate relation size, Q1′-Q6′ favor the top-down approach and

Q1″-Q6″ favor the bottom-up approach. We chose not to restrict variable values in the third triple pattern of the nested *OPTIONAL* clause in any of queries Q1-Q6 because the relation that results after matching the third triple pattern is always used as the right operand of a left outer join and therefore can only marginally influence the join result for the given dataset and queries. Finally, queries Q7, Q8, Q7′, and Q8′ are interesting because they only include triple patterns of the same form with same predicate and variables as subject and object patterns. From the viewpoint of bottom-up and top-down translations, these queries are "symmetric".

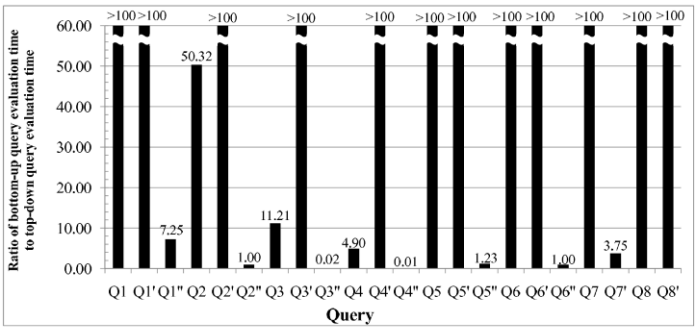
Bottom-Up and Top-Down Query Performance

The S2ST system was used to generate database schemas with property relations [2] and load *WordNet.OWL* into Oracle, DB2 and PostgreSQL. The test SPARQL queries were translated to SQL using algorithms NOGPtoSQL-BU and NOGPtoSQL-TD. The resulting SQL queries were evaluated by RDBMSs. To prevent an unintentional comparison of the three RDBMSs, Figure 6 reports the ratio of a bottom-up query evaluation time to a top-down query evaluation time for each test query. In the figure, if *ratio* > 1, a top-down query was faster; if *ratio* < 1, a bottom-up query was faster; and if *ratio* = 1, both top-down and bottom-up queries showed the same execution time.

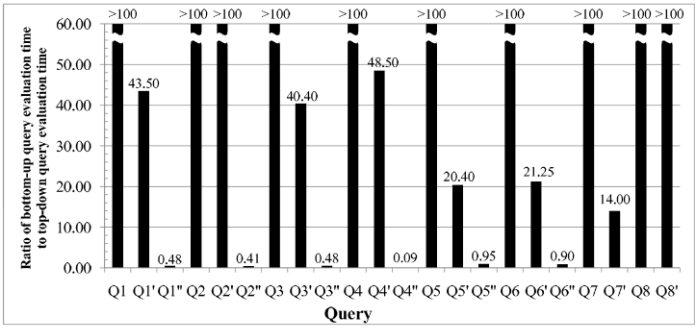
Our *first observation* was that bottom-up and top-down queries generally showed different execution times. This observation gave the definite "No" answer to question "*While it can be expected that relational query optimizers produce identical query execution plans for semantically equivalent bottom-up and top-down queries, is this usually the case in practice*?" in the case of SPARQL queries with nested optional graph patterns.



(a) over an RDF database instantiated in Oracle



(b) over an RDF database instantiated in DB2



(c) over an RDF database instantiated in PostgreSQL

Figure 6: Bottom-up and top-down query performance

Our *second observation* was that different database management systems showed quite different and sometimes even "contradicting" query evaluation ratios. For example, Oracle showed much less contrast between bottom-up and top-down approaches than DB2 and PostgreSQL. Some queries, such as Q1″, Q3, Q4, Q5″, and Q6″, showed different classes of ratios (> 1, < 1, and = 1) in different databases. For example, for Q6″, the bottom-up approach

was slower than the top-down approach in Oracle, equivalent to the top-down approach in DB2, and faster than the top-down approach in PostgreSQL.

Our *third observation* was that selectivities of participating triple patterns and their occurrence in a SPARQL query had a significant impact on which SPARQL-to-SQL translation strategy won, which could be explained by a similar effect of cardinalities of join participating relations and intermediate relations on corresponding top-down and bottom-up SQL queries. In particular, top-down queries Q1 and Q2 were consistently faster in all experiments, given that the first triple pattern *?a rdf:type :Adjective* yielded the smallest result set of 7,345 triples (the other two triple patterns yielded over 10 times larger results), and therefore the intermediate relation in the top-down queries was also small and over 10 times smaller than the intermediate relation in the corresponding bottom-up queries. When *?a rdf:type :Adjective* occurred in the first *OPTIONAL* clause of Q3 and Q4, the situation was opposite: the intermediate relation in the bottom-up queries was over 10 times smaller than the intermediate relation in the corresponding top-down queries. However, while all three systems showed that the ratios decreased when compared to Q1 and Q2, only Oracle showed the advantage of the bottom-up approach, and DB2 and PostgreSQL still ran top-down queries faster. Moving *?a rdf:type :Adective* to the nested *OPTIONAL* clause in Q5 and Q6 did not favor one or the other translation strategy since the last triple pattern did not influence the size of an intermediate relation. Top-down queries Q5 and Q6 were consistently faster in all experiments. Next, restricting selectivities of the first triple pattern in Q1′-Q6′ to 1 or 2 triples, which was favorable for the top-down approach, showed that the top-down queries were faster or as fast as the corresponding bottom-up queries. Interestingly, Oracle showed identical performance for both top-down and bottom-up queries Q1′-Q6′. Finally, Q1″-Q6″, which restricted selectivites of the second triple pattern and favored the

bottom-up approach, showed a consistent performance pattern only for PostgreSQL, where bottom-up queries were faster. For Oracle and DB2, some queries showed a similar pattern: top-down queries Q1″ and Q5″ were faster and bottom-up queries Q3″ and Q4″ were faster; in addition, both bottom-up and top-down Q6″ showed identical times in DB2, top-down Q6″ was faster in Oracle, bottom-up query Q2″ was significantly faster (the smallest ratio in our experiments) in Oracle but as fast as top-down query Q2″ *in* DB2.

Our *fourth observation* was that "symmetric" queries Q7 and Q8 (and similarly Q7′ and Q8′), which are neutral to both top-down and bottom-up translation strategies, showed better performance of the top-down queries. The ratios were significantly larger for DB2 and PostgreSQL, while only from 1.19 to 2.12 times larger in Oracle. These "symmetric" queries showed that, in a general (with no particular bias towards one or the other translation strategy) case, the top-down approach is superior to the bottom-up approach.

Our last, *fifth observation* was that a choice of a translation strategy could have a tremendous impact on a resulting query performance. In one case of Q2″ for Oracle, the bottom-up query was over 600 times faster than the top-down query. In 12 other cases (all occurred in experiments with DB2 and PostgreSQL), the ratios were greater than 1,000 in the favor of top-down queries.

Summary

The performance study gives the answers to the two questions raised in this work. For the first question, our results imply that, in a general case, a relational RDF database designer cannot rely on a relational query optimizer to produce identical or close to identical query execution plans for semantically equivalent SQL queries resulted from bottom-up and top-down

translations of SPARQL queries. To answer the second question, neither of the two approaches is universally better than its sibling. The performance of queries produced by the bottom-up and top-down translation strategies depends on many factors, including selectivities of triple patterns, their order and location in a SPARQL query, and even a relational engine that evaluates translated queries. A number of important observations are made that suggest directions for choosing the best translation strategy for a particular query by a SPARQL query optimizer; the choice can have a tremendous impact on query performance.

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 [5,6] 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 [7, 8], Sesame [9], 3store [10,11], KAON [12], RStar [13], and OpenLink Virtuoso [14]. 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 [15], 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 p_1 o_1 >, < s p_2 o_2 >, etc.). In [16], 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 [17–19], DLDB [20], RDFSuite [21,22], DBOWL [23], PARKA [24], and RDFPROV [25,26]. 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 [27] and [28]. The schema-aware approach is in general yields better query performance than the schema oblivious approach as has been shown in several experimental studies [29, 21, 16, 22]. In addition, the use of a column-oriented DBMS, in conjunction with vertical partitioning of relations, has shown further improvements in query performance [30].

Data-driven: This approach uses RDF data to generate database schema. For example, in

[31], a database schema is generated based on patterns found in RDF data using data mining techniques. RDF store RDFBroker [32] 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 [9] are created when their instances are first seen in an RDF document during data mapping). RDFBroker [32] 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 [22], 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. [22] 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 [16].

One of the most complex mappings in relational RDF databases is the SPARQL-to-SQL query mapping or translation [2], [33], [34], [3], [35], [36]. Existing algorithms translate

SPARQL queries to SQL using either bottom-up or top-down strategy and result in semantically equivalent but syntactically different relational queries. To our best knowledge, this work is the first to compare bottom-up and top-down query translations in the context of complex nested optional graph patterns. The importance of such a comparison is twofold: it gives insights to the query optimization problem of choosing a "good" translation strategy for a particular query and motivates future research on a potentially hubrid translation strategy where both bottom-up and top-down approaches are employed. While we present this work in the context of relational RDF databases, its insights are also beneficial for query optimization in non-relational RDF databases, such as emerging Hadoop and HBase based RDF data management systems in the cloud environment [37], [38].

Other related works on RDF query optimization that are complementary to our research include containment and minimization of RDF/S query patterns [39]. SPARQL query rewriting [40], and various RDF data indexing techniques [41], [42], [43], [44].

CHAPTER VI

CONCLUSION AND FUTURE WORK

In this thesis, we studied the bottom-up and top-down SPARQL-to-SQL translation strategies and compared them empirically in the context of SPARQL queries with nested optional graph patterns. We presented a basic graph pattern translation algorithm that results in flat SQL queries, making the case that the order of translation is not important for such graph patterns. We proposed bottom-up and top-down nested graph pattern translation algorithms and compared their resulting SQL queries in Oracle, DB2, and PostgreSQL. Our performance study suggested that the choice between bottom-up and top-down translation algorithms can have dramatic performance implications on the resulting SQL queries. This choice is dependant on many factors, including selectivities of triple patterns, their order and location in a SPARQL query, and even a relational engine that evaluates translated queries. In the future, we will research a formal framework for optimizing SPARQL queries and defining heuristics for choosing a “good” translation strategy for a SPARQL query.

REFERENCES

[1] A. Chebotko and S. Lu, *Querying the Semantic Web: An Efficient Approach Using Relational Databases.* LAP Lambert Academic Publishing, 2009.

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

[3] A. Chebotko, S. Lu, H.M. Jamil, and F. Fotouhi, "Semantics preserving SPARQL-to-SQL query translation for optional graph patterns," Wayne State University, Tech. Rep. TR-DB-052006-CLJF, May 2006, available from http://www.cs.wayne.edu/~artem/main/research/TR-DB-052006-CLJF.pdf.

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

[6] 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/.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems (TODS)*, pages 16:1-16:45, 2009.

[34] R. Cyganiak. A relational algebra for SPARQL. *Hewlett-Packard Laboratories*, Tech. Rep. HPL-2005-170, 2005, available from http://www.hpl.hp.com/techreports/2004/HPL-2005-170.html.

[35] F. Zemke. Converting SPARQL to SQL. Tech. Rep., October 2006, available from http://lists.w3.org/Archives/Public/public-rdf-dawg/2006OctDec/att-0058/sparql-to-sql.pdf

[36] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. *In Proc. of SSWS*, pages 235-244, 2005.

[37] M. F. Husain, L. Khan, M. Kantarcioglu, and B. M. Thuraisingham. Data intensive query processing for large RDF graphs using cloud computing tools. *In Proc. of CLOUD*, pages 1-10, 2010.

[38] C. Franke, S. Moring, A. Chebotko, J. Abraham, and P. Brazier. Distributed Semantic Web data management in HBase and MySQL Cluster. *In Proc. of CLOUD*, 2011.

[39] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of RDF/S query patterns. *In Proc. of ISWC*, pages 607-623, 2005.

[40] O. Hartig and R. Heese. The SPARQL query graph model for query optimization. *In Proc. of ESWC*, pages 564-578, 2007.

[41] A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. *In Proc. of LA-WEB*, pages 71-80, 2005.

[42] O. Udrea, A. Pugliese, and V. S. Subrahmanian. GRIN: A graph based RDF index. *In Proc. of AAAI*, pages 1465-1470, 2007.

[43] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for Semantic Web data management. *Proc. of PVLDB*, pages 1008-1019, 2008.

[44] G. H. L. Fletcher and P. W. Beck. Scalable indexing of RDF graphs for efficient join processing. *In Proc. of CIKM*, pages 1513-1516, 2009.

BIOGRAPHICAL SKETCH

Andrii Kashliev earned a Master of Science in Computer Science from the University of Texas – Pan American in August 2011. He earned a Bachelor of Science and a Master of Science in Electrical Engineering from Kyiv Polytechnic Institute in 2007 and 2009 respectively. His permanent mailing address is Bakinskaja 37, Apt. 373, Kyiv 04086 Ukraine.