RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation

Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu, and Bhavani Thuraisingham

The University of Texas at Dallas,
800 West Campbell Road, Richardson, TX 75080-3021, USA
{jxr061100,vvk072000,muratk,bxt043000}@utdallas.edu

Abstract. There have been several attempts to make RDBMS and RDF stores inter-operate. The most popular one, D2RQ, has explored one direction i.e. to look at RDBMS through RDF lenses. In this paper we present RETRO, which investigates the reverse direction i.e. to look at RDF through Relational lenses. RETRO generates a relational schema from an RDF store, enabling a user to query RDF data using SQL. A significant advantage of this direction in-addition to interoperability is that it makes numerous relational tools developed over past several decades, available to the RDF stores. In order to provide interoperability between these two DB systems one needs to resolve the heterogeneity between their respective data models and include schema mapping, data mapping and query mapping in the transformation process [1]. However, like D2RQ, RETRO chooses not to physically transform the data and deals only with schema mapping and query mapping. RETRO's schema mapping derives a domain specific relational schema from RDF data and its query mapping transforms an SQL query over the schema into a provably equivalent SPARQL query, which in-turn is executed upon the RDF store. Since RETRO is a read-only framework, its query mapping uses only a relevant and relationally complete subset of SQL. A proof of correctness of this transformation is given based on compositional semantics of the two query languages.

Keywords: Database Interoperability,Query Translation, RDF, RDBMS, Semantic Web, SQL, SPARQL.

1 Introduction

Why not relational databases for Semantic Web?

Since its advent in the 1970's, relational databases have given rise to numerous applications and tools that cater to a wide variety of user requirements. The reasons for the dominance of relational databases for the past several decades are not trivial. It has continually offered the best mix of simplicity, expressivity, performance and robustness in managing generic data. RDBMS has its logical foundations in first-order predicate logic and set theory. It is basically made up of a schema and instance data. The data in a DB is operated upon by relational algebra which can be considered as an abstraction of SQL in a broad sense. The most prominent feature of RDBMS is not reasoning about schema [2], but query answering over the data. Query answering in a database is not logical reasoning, but finite model checking where the model is the given database instance and represents exactly one interpretation [3, Chapter 2]. Consequently, absence of information in a database instance is interpreted as negative information. In other words traditional semantics in RDBMS is characterized as a closed-world semantics. However, over time, application requirements have evolved, giving rise to a need to support reasoning or entailment in query answering. A key example of this being World Wide Web or Semantic Web. Knowledge bases in Semantic Web are based on Description logics which generally are decidable fragments of first-order logic. Unlike RDBMS, query answering in Semantic Web knowledge bases uses reasoning or logical entailment by considering all possible interpretations, which essentially is its open-world semantics. In addition to query answering, nature of data is an additional factor that leads to choice of RDF model over relational model. The data pertaining to RDBMS applications is homogeneous in nature as it usually is a complete description of a particular domain. This makes it amenable to being modeled as entities and relationships with only occasional problems of null values. On the other hand, data in the case of Semantic Web applications is most likely an incomplete instance dataset and such information cannot be modeled as entities and relationships without giving rise to countless null values. Factors such as incompleteness of data, non-finiteness of domain, query answering based on entailment and support for rich expressivity make RDF/OWL a better choice over relational model/RDBMS for Semantic Web applications.

Why RETRO, why bridge the gap in reverse direction?

Owing to the time Relational databases have been around they have developed into a mature technology, dominating the persistence mechanism market with several well-established vendors and standards such as SQL and JDBC, which are well defined and accepted. For the same reasons, experience base of developers for relational databases significantly outnumber RDF specialists. There has been a considerable growth in RDF data and a lot of valuable data now resides in RDF stores. Since a vast majority of RDBMS developers may have little familiarity and limited resources to invest in Semantic Web and understand its underlying principles, tools and technologies, it becomes imperative to provide them with a means to conveniently query the existing RDF data. In-order to build such a tool, one has to step into their shoes and assess their requirements as they think relationally and know little about the expressivity supported by RDF/OWL but are indeed looking forward to use/query the RDF data with minimal effort. A tool like RETRO will enable one to effortlessly reuse existing data visualization, data reporting and data mining tools available for relational data. Additionally, commercial applications that use semantic web technologies are impeded because there are much fewer Business Intelligence tools available for them when compared with tools available for relational data. RETRO will not only help commercial applications use semantic web technologies but also serve to cordially welcome RDBMS users to the brave new world of semantic web. A curious RDBMS user may eventually be instigated to leverage the expressivity supported by SPARQL and get motivated to learn more about it and other semantic web technologies. It may thus eventually lead to higher acceptance of semantic web (SW) and bring more work force into SW research. Integrating various databases is often a motivation for adopting RDF [4] and this in turn also serves as a motivation for RETRO, which addresses aforementioned issues of RDBMS folks such as ease of adaptability, backwards compatibility and painless migration to newer DB/Web technologies (RDF/Semantic Web) by bridging the gap between the two fundamentally different DB systems.

How is the gap bridged to achieve interoperability?

RETRO facilitates interoperability between RDF stores and RDBMS without physically transforming the data. This interoperability is achieved in two steps, namely schema mapping and query mapping. In schema mapping, RETRO aims to provide RDBMS users access to RDF data by creating a relational schema that directly corresponds to the RDF data. The relational schema is obtained by virtually, vertically partitioning [5] RDF data based on predicates. The rationale behind this approach is described in the section on schema mapping. The resulting relational schema consists of relations of degree two, i.e., subject and object of a predicate form the attributes of these relations. The user can now create an SQL query over this relational schema. This SQL query is then converted into a semantically equivalent SPARQL query using RETRO's Query mapping procedure. The SPARQL query is now executed over the RDF store and the results are presented to the user in relational format. Along the following lines we state some assumptions made for query mapping/translation. A query translator is not a query processor and is therefore not responsible for verifying semantic correctness of the source query and assumes it to be correct. Given a semantically correct source query, the translator transforms it from source language to target language by preserving the semantics. Figure 1 shows the schema and query mapping components of RETRO.

Problem Scope: Formal semantics of SPARQL was first discussed in an influential work by Perez et. al in [6] in 2006 and was followed and adapted by W3C standard specification of SPARQL [7] in 2008. In [6], authors present algebraic syntax with compositional semantics. We reuse both of these features of SPARQL as it is the target language of our query translation. In other words, we start with an algebraic syntax and compositional semantics of relational algebra and map it to the algebraic syntax and compositional semantics of SPARQL respectively. Since we reuse the compositional semantics from [6], the assumptions made in it also hold in this paper, which we briefly restate here. Algebraic formalization of

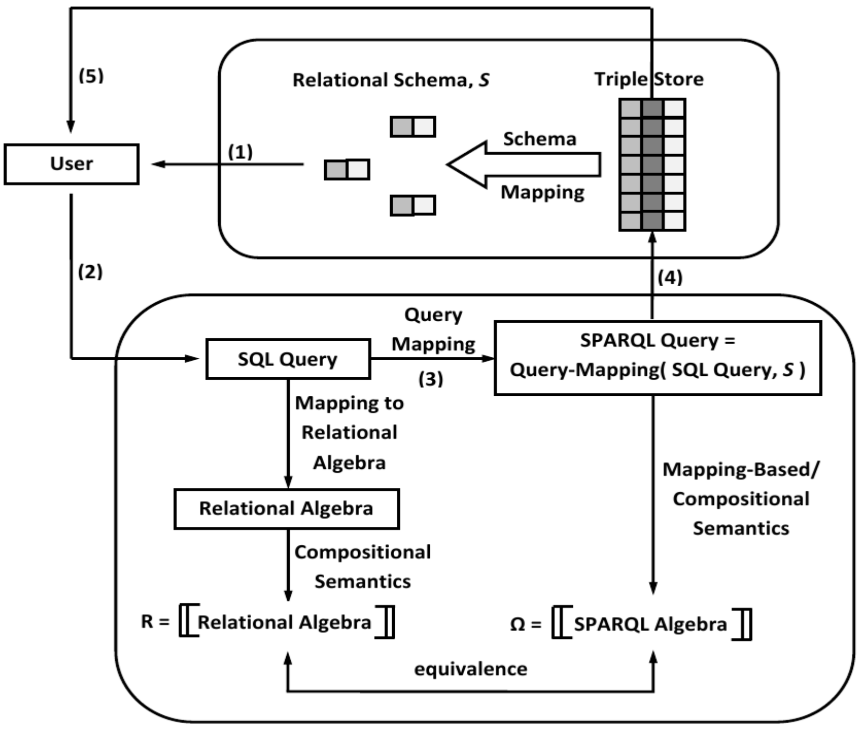

Fig. 1. Schema and Query Mapping

the core fragment of SPARQL is done over simple RDF graphs without RDFS vocabulary and literal rules, and set semantics is used instead of bag semantics as implied in the SPARQL specification.

Problem Definition: Given an RDF dataset and a set of relational algebra operators, the goal is firstly to derive a relational schema that accurately represents the RDF data and secondly to map each of the relational algebra operators (applied to operands from relational schema) to a semantically equivalent SPARQL algebra operator or a set of operators. The result set obtained upon SPARQL query evaluation on RDF store is presented to the user as a relation.

Contributions: We describe a means of computing domain specific relational schema from a given RDF data-store. We define a relationally complete [8] core fragment of SQL/relational algebra keeping in mind the read-only aspect of the application. We then describe compositional semantics of this core fragment of relational algebra. Subsequently, we describe compositional semantics of SPARQL based on work done in [6] and [1]. Finally, we give a proof of semantics preservation of the translation based on compositional semantics of the two query languages. The aforementioned abstractions, restrictions and assumptions made for query translation lend to the formulation of a simple proof of correctness, by retaining the core complexity of the problem. To the best of our knowledge, this paper presents the first formal semantics preserving translation from SQL to SPARQL.

2 RETRO Framework: Schema Mapping

The Web ontology language (OWL) is a set of knowledge representation languages, that have logical foundations in Description Logics (DL), which in turn are decidable fragments of first order logic with more efficient decision procedures. A DL knowledge base (KB) consists of two components namely TBox and ABox. TBox essentially is a terminology that describes a domain in terms of concepts and their general properties, whereas an ABox contains assertional knowledge i.e. knowledge that is specific to individuals of domain of discourse. The main inference problem in ABox is instance checking which can be generalized to answering conjunctive queries over DL knowledge bases. A query in a DL KB is a first-order formula and semantics of query answering is again defined as first-order entailment as it considers all possible models of a KB.

A conceptual model such as ER model is a high level, semantic data model that provides concepts that are close to the way RDBMS users perceive data. A relational data model is a logical data model that uses a relation as its central data description construct along with integrity constraints that make data described in a relation more precise. A relation in RDBMS is made up of relation schema and relation state/instance. A relational schema denoted by $R(A_1, A_2, A_3, ...A_n)$ is made up of a relation name R and a set of attributes $A_1, A_2, A_3, ...A_n$, where domain of each attribute is denoted by dom(A_i). Queries in a RDBMS are first-order formulas and are evaluated over a given database instance I and are therefore not first-order entailment but model checking, where the model is the given database instance I.

ER model and Description logics both model a given domain using classes and relationships. Calvanese el al., in [9] define a translation function that transforms ER model into a $\mathcal{ALUNI}$ DL knowledge base [9], and they prove that this transformation is information preserving. Relational databases have ER model as the most popular conceptual data model and can be mapped to a description language that closely corresponds to the constructs of ER model. However, in Semantic Web there are many flavors of description languages with varying expressivity and computational properties that can be used to model a given domain. It is not possible to map or transform every description language to ER model as they may have different expressivity. In our framework we derive a domain specific relational schema from ABox part of the DL knowledge base and do not explicitly consider the TBox. The SCHEMA-MAPPING algorithm traverses the entire set of ABox triples and extracts every unique predicate that is stored separately for later use.

As mentioned earlier, semantics of query answering in relational databases is model checking, whereas in semantic web it is first-order entailment. Since we do not evaluate an SQL query but transform it into an equivalent SPARQL query and evaluate it over an RDF store, we continue using open-world semantics for query answering.

Algorithm 1 SCHEMA-MAPPING()

```
Input: $ABox$
Output: A map, $P$ (relation name $\rightarrow$ rank)
 1: $P \leftarrow \emptyset$ {The map $P$ is initially empty}
 2: $rank \leftarrow 1$
 3: for $i \leftarrow 1$ to $ABox.size$ do
 4:    $T \leftarrow ABox[i]$ {Retrieve a triple $T$ from the $ABox$}
 5:    $p \leftarrow T.predicate$ {Retrieve the predicate $p$ from the triple $T$}
 6:    if $p \notin P$ then
 7:       $P.put(p, rank)$ {Add $p$ to the map $P$ with the current value of $rank$}
 8:       $rank \leftarrow rank + 1$
 9:    end if
10: end for
11: return $P$
```

The map P returned by the SCHEMA-MAPPING algorithm can be used to display the relation schema to users. A relation schema is constructed by extracting each of the predicate names from P and adding attributes s and o. A relation instance, p_I in a schema S for a given predicate p, can be constructed by evaluation of a triple pattern, tp such that $tp.s = ?s_i \wedge tp.p \in P \wedge tp.o = ?o_i$, over the triple store. A pair of values (s_j, o_j) that are bound to variables $(?s_i, ?o_i)$ form a tuple in p_I. The SCHEMA-MAPPING algorithm also ranks the predicates in the order in which they are discovered. The significance of ranking will become evident in the next subsection where it is used to unambiguously generate triple patterns.

We now give an example of a RDF dataset D, which is an extension of the dataset provided in [6], and its corresponding relation schema S which is generated by the SCHEMA-MAPPING algorithm. We also give an example of an instance of relation "name" that can be constructed from D. The schema S will be further used in the next subsection to illustrate a few examples of how a SQL query on schema S is translated into an equivalent SPARQL query. A reader should also note that when the number of predicates in a ABox is very large, we envisage providing a user with an auxiliary schema for the sole purpose of understanding the structure of data to help them with query formulation. However, the implementation of such an auxiliary schema is left as a part of future work.

(B_1, name, paul)	(B_2, name, john)	(B_3, name, george)
(B_4, name, ringo)	(B_1, age, 30)	(B_2, age, 40)
(B_3, age, 25)	(B_4, age, 35)	(B_1, sal, 20,000)
(B_2, sal, 30,000)	(B_3, sal, 40,000)	(B_4, sal, 50,000)
(B_2, dept, Research)	(B_3, dept, Research)	(B_3, dept, Admin)
(B_4, dept, Admin)	(B_1, phone, 777-3426)	(B_4, phone, 888-4537)
(B_2, email, john@acd.edu)	(B_4, email, ringo@acd.edu)	(B_3, webPage, www.george.edu)
(B_4, webPage, www.starr.edu)		

Table 1. The RDF dataset D

```
S = { name(s, o), age(s, o), sal(s, o), webPage(s, o), dept(s, o), phone(s, o), email(s, o) }
```

SELECT ?s ?o
WHERE { ?s name ?o }   =>

name	
s	o
B_1	paul
B_2	john
B_3	george
B_4	ringo

Table 2. A SPARQL query and relation instance obtained from its evaluation on D for predicate "name"

3 RETRO Framework: Query Mapping

In this section we first illustrate the SQL-to-SPARQL translation with some examples and then describe an abstract grammar for SQL which is used to build parse trees for a given SQL query. We then describe a set of algorithms for query mapping: TransSQLFromClause, TransSQLWhereClause, TransSQLSelectClause, Query-Mapping and Merge. In our examples we use the RDF dataset D and its associated relational schema S that were defined in the previous subsection.

In Tables 3 and 4 we give a description of an SQL query on relational schema S, its equivalent SPARQL query over dataset D obtained from our Query-Mapping algorithm and its associated subprocedures and subsequently, we show the result of evaluating the SPARQL query on dataset D. The SQL queries in the tables use the following relational algebra operators: Cross product, Join, Union, Intersection and Difference.

3.1 Abstract Grammar for SQL

As shown in the abstract grammar below, an input SQL query for the translation can be a simple Cross product query, Select-Project-Join query or queries with Union, Intersection or Difference (Except) operators. Additionally, the Query-Mapping algorithm (Algorithm 5) supports equi-join and does not consider conditional join because of the nature of RDF data. Conditional joins are meaningful in relational data, however, since the relational schema is derived from RDF data and joins over the derived schema are usually performed over attributes which are URI's, conditional joins do not come into picture.

For example, if one needs to perform a join of two relations, name and age from schema S on attribute s, and since domain of s is URI's, only an equi-join is meaningful while a conditional join is not. We also do not yet support self-join and will consider it in our future work if our investigation reveals a need for it.

```
SqlQuery          --> CrossProductQuery | SPJQuery | UnionQuery |
                      IntersectQuery | ExceptQuery.
CrossProductQuery --> SelectClause FromClause.
SPJQuery          --> SelectClause FromClause WhereClause.
UnionQuery        --> SqlQuery 'UNION' SqlQuery.
```

Description	SQL Query	SPARQL Query	ResultSet

Cross Product of relations email and webPage	SELECT email.s, email.o,
	webPage.s, webPage.o
	FROM email, webPage
		SELECT ?s1 ?o1 ?s2 ?o2
		WHERE { ?s1 email ?o1 .
		?s2 webPage ?o2 }
			?s1 ?o1 ?s2 ?o2
			B_2 john@acd.edu B_3 www.george.edu
			B_2 john@acd.edu B_4 www.starr.edu
			B_4 ringo@acd.edu B_3 www.george.edu
			B_4 ringo@acd.edu B_4 www.starr.edu

Find the email address of john	SELECT email.o
	FROM name, email
	WHERE name.s = email.s
	AND name.o = john
		SELECT ?o2
		WHERE { ?s1 name ?o1 .
		?s1 email ?o2 .
		FILTER (?o1 = john) }
			?o2
			john@acd.edu

Names of all people with age ≤ 25 and salary > 30K	SELECT name.s, name.o
	FROM name, age
	WHERE age.o $\leq$ 25 AND name.s = age.s
	INTERSECT
	SELECT name.s, name.o
	FROM name, sal
	WHERE sal.o > 30,000 AND name.s = sal.s
		SELECT ?s1 ?o1
		WHERE { { ?s1 name ?o1 .
		?s1 age ?o2 .
		FILTER (?o2 $\leq$ 25) }
		{ ?s1 name ?o1 .
		?s1 sal ?o3 .
		FILTER (?o3 > 30,000) } }
			?s1 ?o1
			B_3 george

Table 3. Illustration of SQL-to-SPARQL translation on dataset D

Description	SQL Query	SPARQL Query	ResultSet

Names of all people who either have a phone number or an email address	SELECT name.o
	FROM name, email
	WHERE
	name.s = email.s
	UNION
	SELECT name.o
	FROM name, phone
	WHERE
	name.s = phone.s
		SELECT ?o1
		WHERE {
		{ ?s1 name ?o1 .
		?s1 email ?o2 . }
		UNION
		{ ?s1 name ?o1 .
		?s1 phone ?o3 . }
		}
			?o1
			paul
			john
			ringo

Find the resources that work for the Research department but not the Admin department	SELECT dept.s
	FROM dept
	WHERE
	dept.o = Research
	EXCEPT
	SELECT dept.s
	FROM dept
	WHERE
	dept.o = Admin
		SELECT ?s1
		WHERE { {
		?s1 dept ?o1 .
		FILTER
		(?o1 = Research) }
		OPTIONAL {
		?s1 dept ?o2 .
		FILTER
		(?o2 = Admin) }
		FILTER
		(!bound(?o2)) }
			?s1
			B_2

Table 4. Illustration of SQL-to-SPARQL translation on dataset D

```
IntersectQuery     --> SqlQuery 'INTERSECT' SqlQuery.
ExceptQuery        --> SqlQuery 'EXCEPT' SqlQuery.
SelectClause       --> 'SELECT' AttributeList.
FromClause         --> 'FROM' TableList.
WhereClause        --> 'WHERE' ExpressionList.
AttributeList      --> Attribute',' AttributeList | Attribute.
Attribute          --> String.
TableList          --> Table','TableList | Table.
Table              --> String.
ExpressionList     --> Expression Lop ExpressionList | Expression.
Expression         --> Attribute op Attribute | Attribute op Value.
Value              --> Numeral | String.
Op                 --> '<' | '>' | '=' | '>=' | '<=' | '<>'.
Lop                --> 'AND' | 'OR' | 'NOT' .
```

3.2 Query Mapping algorithms

The conceptual evaluation strategy of an SQL query first computes the cross-product of the tables in the FROM list, then deletes the rows in the cross-product that fail the qualification conditions and finally deletes all the columns that do not appear in the SELECT list. Additionally, if DISTINCT is specified, duplicate rows are eliminated. The Query-Mapping algorithm is used to convert a given SQL query into a semantically equivalent SPARQL query and follows the conceptual query evaluation strategy of SQL. This algorithm in-turn calls the TransSQLFromClause sub-procedure which takes as input, a set of table names from a SQL FROM clause, R and returns a map from table names to triple patterns, TP. This sub-procedure generates a triple pattern, $?s_i\ r_i\ ?o_i$, for every table, r_i, that is present in a SQL FROM clause (line 3). Further, each generated triple pattern is added to map TP indexed by its corresponding table name (line 4). This procedure could also simply return the corresponding triple patterns but instead returns a map as it allows the next sub-procedure TRANSSQLWHERECLAUSE to index the triple patterns based on table names in a SQL FROM clause.

Algorithm 2 TRANSSQLFROMCLAUSE()

```
Input: A set of table names from a SQL FROM clause, $R$
Output: A map from table names to triple patterns, $TP$
1: $TP \leftarrow \varnothing$ {The map $TP$ is initially empty}
2: for $i \leftarrow 1$ to $R.size$ do
3:     $tp \leftarrow \{?s_i\ r_i\ ?o_i\}$ {A triple pattern is constructed for each $r_i \in R$}
4:     $TP.put(r_i, tp)$
5: end for
6: return $TP$
```

The QUERY-MAPPING algorithm then evaluates the WHERE clause of the input SQL query using the sub-procedure TRANSSQLWHERECLAUSE. This sub-procedure takes as input JC, BE, the map TP generated by sub-procedure TRANSSQLFROMCLAUSE and map P generated from Algorithm 1. The variable JC denotes a set of join conditions, where a condition is of the form: *Attr Op Attr* and BE denotes a set of boolean expressions where an expression is of the form: *Attr Op Value*. In this sub-procedure we use the aforementioned ranking of the schema relations to sort the join conditions of the set JC (line 5). This step is necessary to ensure that unambiguous triple patterns are generated for join conditions in a SPARQL query. We present the following example to show the necessity of ranking relations: Consider relations *name*, *age* and *dept* with ranks 1, 2 and 3 respectively and a query containing only join conditions "*name.s* = *age.s* AND *dept.s* = *age.s*". If we do not rank relations, we get the following SPARQL WHERE clause using TRANSSQLWHERECLAUSE:

```
{ ?s1 name ?o1 .
  ?s3 age  ?o2 .
  ?s3 dept ?o3 }
```

However, in sub-procedure TRANSSQLWHERECLAUSE, as part of the function $SORT(JC, P)$, the second join condition would be sorted to become *age.s* = *dept.s*, since the rank of *dept* (3) was originally > the rank of *age* (2). The sorted join conditions would now lead to the correct WHERE clause:

```
{ ?s1 name ?o1 .
  ?s1 age  ?o2 .
  ?s1 dept ?o3 }
```

The ranking of relations used in join re-ordering ensures generation of unique SPARQL WHERE clause. The process of ranking relations and sorting join conditions is merely an implementation detail and has little conceptual significance. The TRANSSQLWHERECLAUSE then generates a part of the SPARQL WHERE clause by translating join conditions in a given SQL query into equivalent triple patterns (lines 6-24). Once the appropriate triple patterns have been generated for the join conditions, the next step is to generate the filter conditions which are semantically equivalent to the selection conditions from the SQL query (lines 25-33). Finally, the sub-procedure returns the semantically equivalent WHERE clause string (line 34). The main complexity of this sub-procedure lies in generating triple patterns that are semantically equivalent to the join of the tables in the input SQL query.

The TRANSSQLSELECTCLAUSE sub-procedure takes as input a set of SQL SELECT attributes, A, and a map, TP and returns a SPARQL SELECT clause string which is semantically equivalent to the corresponding SQL query's SELECT clause. It generates the variables for SPARQL SELECT clause by iterating over the set of attributes, A, where an attribute $a_i \in A$ is of the form: "relationName.attributeName". During an iteration the relation name part of an attribute a_i is used to access the corresponding triple pattern from TP (line 3) and the attribute name part of a_i is used to access the desired variable within this triple pattern (line 2). Then, based on whether an attribute is a subject or an object, the appropriate variable is appended to the list of variables in the SPARQL SELECT clause (lines 5-6).

Algorithm 3 TransSQLWhereClause()

Input: Join conditions JC, Boolean expressions BE, TP (relation name $\rightarrow$ triple pattern), P (relation name $\rightarrow$ rank)
Output: A SPARQL WHERE clause

```
 1: $where =$ "" {A SPARQL WHERE clause that is initially blank}
    {Return a SPARQL WHERE clause containing triple patterns as it is for a blank SQL WHERE clause}
 2: if ($JC.isEmpty()$ AND $BE.isEmpty()$) then
 3:    for each $tp \in TP$ do $where += tp.subject +$ " " $+ tp.predicate +$ " " $+ tp.object$ end for
 4: end if
 5: $JC = SORT(JC, P)$ {$JC$ denotes a set of join conditions where a condition is of the form: $Attr\ Op\ Attr$}
 6: for each condition $p \in JC$ do
 7:    $p_i^1 = p.lOperand$; $p_i^2 = p.rOperand$; {A $p_i$ has a relation name, $relation$ and an attribute name, $attribute$}
 8:    $tp_1 \leftarrow TP.get(p_i^1.relation)$ {Get the triple pattern for $p.lOperand$}
 9:    $tp_2 \leftarrow TP.get(p_i^2.relation)$ {Get the triple pattern for $p.rOperand$}
10:    $where += tp_1.subject +$ " " $+ tp_1.predicate +$ " " $+ tp_1.object$
       {The triple pattern for $p.lOperand$ is always appended as it is to the output}
       {The triple pattern for $p.rOperand$ depends on the two joining variables}
11:    if $p_i^1.attribute =$ "$s$" and $p_i^2.attribute =$ "$s$" then
12:       $where += tp_1.subject +$ " " $+ tp_2.predicate +$ " " $+ tp_2.object$
13:       $tp \leftarrow \{tp_1.subject\ tp_2.predicate\ tp_2.object\}$
14:       $TP.put(p_i^2.relation, tp)$
15:    else if $p_i^1.attribute =$ "$s$" and $p_i^2.attribute =$ "$o$" then
16:       $where += tp_2.subject +$ " " $+ tp_2.predicate +$ " " $+ tp_1.subject$
17:       $tp \leftarrow \{tp_2.subject\ tp_2.predicate\ tp_1.subject\}$
18:       $TP.put(p_i^2.relation, tp)$
19:    else if $p_i^1.attribute =$ "$o$" and $p_i^2.attribute =$ "$s$" then
20:       $where += tp_1.object +$ " " $+ tp_2.predicate +$ " " $+ tp_2.object$
21:       $tp \leftarrow \{tp_1.object\ tp_2.predicate\ tp_2.object\}$
22:       $TP.put(p_i^2.relation, tp)$
23:    end if
24: end for
    {$BE$ denotes a set of boolean expressions where an expression is of the form: $Attr\ Op\ Value$}
25: for each expression $p \in BE$ do
26:    $p_i^1 = p.lOperand$; $p_i^2 = p.rOperand$;
27:    $tp_1 = TP.get(p_i^1.relation)$ {Get the triple pattern for $p.lOperand$}
28:    if $p_i^1.attribute =$ "$s$" then
29:       $where +=$ "$FILTER($" $+ tp_1.subject +$ " " $+ p.operator +$ " " $+ p_i^2 +$ " )"
30:    else
31:       $where +=$ "$FILTER($" $+ tp_1.object +$ " " $+ p.operator +$ " " $+ p_i^2 +$ " )"
32:    end if
33: end for
34: return $where$
```

Algorithm 4 TransSQLSelectClause()

Input: A set of SQL SELECT attributes, A, and a map, TP (relation name $\rightarrow$ triple pattern)
Output: A SPARQL SELECT clause

```
1: $select =$ "" {A SPARQL SELECT clause that is initially blank}
2: for each attribute $a \in A$ do
3:    $r_{name} \leftarrow a.relation$; $a_{name} = a.attribute$ {Get the relation and attribute for each $a \in A$}
4:    $tp \leftarrow TP.get(r_{name})$ {Get the triple pattern for $r_{name}$}
5:    if $a_{name} =$ "$s$" then $select += tp.subject +$ " "
6:    else $select += tp.object +$ " " end if
7: end for
8: return $select$
```

The main procedure, Query-Mapping, takes as input an SQL query string and a map P generated from the Schema-Mapping algorithm, and returns a SPARQL query string. The SQL query string is used to generate its corresponding parse tree using function parse(SQLQuery). We then extract SQL SELECT, FROM, WHERE-JC clause (Join Conditions) and WHERE-BE (Boolean Expressions) clause strings from the parse tree. If the query does not contain any of the parts, the corresponding strings remain initialized to empty. For example a cross product query will have null values for WHERE-JC and WHERE-BE. The procedure now checks the parse tree to determine the type of the query and does the corresponding translation. As described earlier, the Query-Mapping algorithm follows the conceptual query evaluation strategy of SQL. Therefore, this algorithm makes use of the previously defined TransSQLFromClause, TransSQLWhereClause and TransSQLSelectClause sub-procedures to perform a query translation. Since SQL UNION, INTERSECT and EXCEPT queries can be considered as being composed of two separate sub-queries, we recursively use the Query-Mapping procedure to evaluate each sub-query. The results of evaluation of sub-queries are merged using the Merge sub-procedure that generates the final SPARQL query string.

We now give a simple example that demonstrates how a SQL SPJQuery will be translated to a semantically equivalent SPARQL query. The Query-Mapping algorithm first calls the sub-procedure TransSQLFromClause. The result generated is a map TP from relation names to triple patterns, which is then used in the procedure TransSQLWhereClause with WHERE-JC and WHERE-BE conditions additionally provided. This sub-procedure returns a SPARQL WHERE clause string which is appended to the result of the evaluation of TransSQLSelectClause.

Algorithm 5 Query-Mapping()

```
Input: An SQL query, $q_{in}$, and a map $P$
Output: A SPARQL query, $q_{out}$
 1: $q_{out}$ =“” {A SPARQL query that is initially blank}
 2: $tree$ = parse($q_{in}$) { A parse tree obtained by parsing $q_{in}$}
 3: $q_{in}^{SELECT} = tree.getSelectClause()$; $q_{in}^{FROM} = tree.getFromClause()$;
 4: $q_{in}^{WHERE-JC} = tree.getJoinConditions()$ $q_{in}^{WHERE-BE} = tree.getBooleanExpr()$
 5: $q_{out}^{SELECT}$ =“$SELECT$ ” $q_{out}^{WHERE}$ =“$WHERE$ { ”
 6: $TP \leftarrow \varnothing$ { The map of triple patterns is initially empty}
 7: if $tree.type$ = CrossProductQuery then
 8:     $TP$ = TransSQLFromClause($q_{in}^{FROM}$)
 9:     $q_{out}^{WHERE}$ += TransSQLWhereClause($q_{in}^{WHERE-JC}, q_{in}^{WHERE-BE}, TP, P$)
10:     $q_{out}^{SELECT}$ += TransSQLSelectClause($q_{in}^{SELECT}, TP$)
11:     $q_{out} = q_{out}^{SELECT} + q_{out}^{WHERE}$+“ }”
12: else if $tree.type$ = SPJQuery then
13:     $TP$ = TransSQLFromClause($q_{in}^{FROM}$))
14:     $q_{out}^{WHERE}$ = TransSQLWhereClause($q_{in}^{WHERE-JC}, q_{in}^{WHERE-BE}, TP, P$)
15:     $TP = extractTP(q_{out}^{WHERE})$
16:     $q_{out}^{SELECT}$ += TransSQLSelectClause($q_{in}^{SELECT}, TP$)
17:     $q_{out} = q_{out}^{SELECT} + q_{out}^{WHERE}$+“ }”
18: else if $tree.type$ = UnionQuery then
19:     $q_1 = tree.leftSubTree(), q_2 = tree.rightSubTree()$
20:     $q_1^{out}$ = Query-Mapping($q_1$), $q_2^{out}$ = Query-Mapping($q_2$)
21:     $q_{out} = Merge(q_1^{out}, q_2^{out}$, “UNION”)
22: else if $tree.type$ = IntersectQuery then
23:     $q_1 = tree.leftSubTree(), q_2 = tree.rightSubTree()$
24:     $q_1^{out}$ = Query-Mapping($q_1$), $q_2^{out}$ = Query-Mapping($q_2$)
25:     $q_{out} = Merge(q_1^{out}, q_2^{out}$, “INTERSECT”)
26: else if $tree.type$ = ExceptQuery then
27:     $q_1 = tree.leftSubTree(), q_2 = tree.rightSubTree()$
28:     $q_1^{out}$ = Query-Mapping($q_1$), $q_2^{out}$ = Query-Mapping($q_2$)
29:     $q_{out} = Merge(q_1^{out}, q_2^{out}$, “EXCEPT”)
30: end if
31: return $q_{out}$
```

Algorithm 6 MERGE()

```
Input: $q_1, q_2, Q\_TYPE$
Output: A SPARQL query $q_{out}$
 1: $q_{out} =$ “” {A SPARQL query that is initially blank}
 2: $V_1 = q_1.extractSelectClause();\ V_2 = q_2.extractSelectClause();$
 3: $W_1 = q_1.extractTriplePatterns();\ W_2 = q_2.extractTriplePatterns();$
 4: $F_1 = q_1.extractFilter();\ F_2 = q_2.extractFilter();$
 5: if $Q\_TYPE =$ “UNION” then
 6:    $q_{out} +=$ “$SELECT$ ”
 7:    for $i \leftarrow 1$ to $V_1.size$ do $q_{out} += V_1[i]$; end for
 8:    $q_{out} +=$ “$WHERE$ { { ”
 9:    for $i \leftarrow 1$ to $W_1.size$ do $q_{out} += W_1[i]$; end for
10:    for $i \leftarrow 1$ to $F_1.size$ do $q_{out} += F_1[i]$; end for
11:    $q_{out} +=$ “ } $UNION$ { ”
12:    for $i \leftarrow 1$ to $W_2.size$ do $q_{out} += W_2[i]$; end for
13:    for $i \leftarrow 1$ to $F_2.size$ do $q_{out} += F_2[i]$; end for
14:    $q_{out} +=$ “ }} ”
15: else if $Q\_TYPE =$ “INTERSECT” then
16:    $q_{out} +=$ “$SELECT$ ”
17:    for $i \leftarrow 1$ to $V_1.size$ do $q_{out} += V_1[i]$; end for
18:    $q_{out} +=$ “$WHERE$ { { ”
19:    for $i \leftarrow 1$ to $W_1.size$ do $q_{out} += W_1[i]$; end for
20:    for $i \leftarrow 1$ to $F_1.size$ do $q_{out} += F_1[i]$; end for
21:    $q_{out} +=$ “ } { ”
22:    for $i \leftarrow 1$ to $W_2.size$ do
23:       if $W_2[i].subject \notin V_1$ then $W_2[i].subject = uniqueVar(W_2[i].subject)$ end if
24:       if $W_2[i].object \notin V_1$ then $W_2[i].object = uniqueVar(W_2[i].object)$ end if
25:       $q_{out} += W_2[i]$;
26:    end for
27:    for $i \leftarrow 1$ to $F_2.size$ do
28:       if $F_2[i].var \notin V_1$ then $F_2[i].var = uniqueVar(F_2[i].var)$ end if
29:       $q_{out} += F_2[i]$;
30:    end for
31:    $q_{out} +=$ “ }} ”
32: else if $Q\_TYPE =$ “EXCEPT” then
33:    $q_{out} +=$ “$SELECT$ ”
34:    for $i \leftarrow 1$ to $V_1.size$ do $q_{out} += V_1[i]$; end for
35:    $q_{out} +=$ “$WHERE$ { { ”
36:    for $i \leftarrow 1$ to $W_1.size$ do $q_{out} += W_1[i]$; end for
37:    for $i \leftarrow 1$ to $F_1.size$ do $q_{out} += F_1[i]$; end for
38:    $q_{out} +=$ “ } $OPTIONAL$ { ”
39:    for $i \leftarrow 1$ to $W_2.size$ do
40:       if $W_2[i].subject \notin V_1$ then $W_2[i].subject = uniqueVar(W_2[i].subject)$ end if
41:       if $W_2[i].object \notin V_1$ then $W_2[i].object = uniqueVar(W_2[i].object)$ end if
42:       if $W_2[i].object \in V_1$ then $W_2[i].object = uniqueVar(W_2[i].object)$ end if
43:       $q_{out} += W_2[i]$;
44:    end for
45:    for $i \leftarrow 1$ to $F_2.size$ do
46:       $F_2[i].var = uniqueVar(F_2[i].var)$
47:       $q_{out} += F_2[i]$;
48:    end for
49: end if
50: return $q_{out}$
```

The sub-procedure MERGE, merges two input sub-query strings and generates a meaningful SPARQL query. It first extracts the SELECT clauses from each of the sub-queries and then stores these in variables V_1 and V_2 respectively (line 2). It then extracts and stores the triple patterns from the WHERE clauses of the two sub-queries in variables W_1 and W_2 (line 3). Finally, it extracts and stores the boolean conditions from the FILTER expressions of each of the sub-queries into variables F_1 and F_2 (line 4). SQL requires the sub-queries (relations) to be union compatible in order to perform UNION, INTERSECTION or DIFFERENCE. Hence, the SPARQL SELECT clause for each of the queries (UNION, INTERSECT, EXCEPT) will contain only variables from V_1. The merging operation to generate a SPARQL WHERE clause for UNION is simple. The set of triple patterns from the first sub-query are concatenated with the set of triple patterns from the second sub-query with keyword UNION in between (lines 5-14). We repeat the same procedure to generate the WHERE clause for an INTERSECT query (lines 16-21) and do an additional renaming of variables (lines 22-30). The renaming is done only to variables in the set of triple patterns (W_2) and filter conditions (F_2) that belong to the second sub-query. If these variables do belong to the set V_1 then they remain the same else they will be uniquely renamed using function $uniqueVar$. The function $uniqueVar$ when given a SPARQL variable as input returns as output a new variable name that has not been used earlier and additionally, if it is called multiple times using the same variable name it returns the same result each time. In order to generate the WHERE clause of the EXCEPT query we again rename the variables of the triple patterns in W_2 that do not belong to the set V_1 (lines 40-41). In addition, we also rename variables from triple patterns in W_2 if they are present as objects and they are also part of the set V_1 (line 42). Finally, all variables that belong to the set of filter expressions, F_2, are also renamed (lines 45-48). The sub-procedure implicitly checks that only variable names are renamed and does not manipulate literal values.

4 Semantics Preserving SQL-To-SPARQL Translation

An important principle of compositional semantics is that the semantics should be compositional i.e. the meaning of a program expression should be built out of the meanings of its sub-expressions. We present compositional semantics in terms of abstract syntax definition of the language, semantic algebras and the valuation functions.

- Abstract Syntax: It is commonly specified as BNF grammer and it maps expressions in the language to parse trees. A Syntax domain is the term used to denote a collection of values with common syntactic structure.
- Semantic Algebra: A Semantic algebra consists of a semantic domain accompanied by a corresponding set of operations. A semantic domain is a set of elements grouped together because they share some common property, for e.g. set of natural numbers, set of tuples in a relation etc. The operations are functions that need arguments from the domain to produce answers, for e.g. selection, projection, join, are meaningful operations over the domain of relations. Operations are defined in terms of operation's functionality and a description of operation's mapping.
    - Functionality: For a function f its functionality is an expression from domain and co-domain denoted by $f : D_1 \times D_2 \times ... \times D_n \rightarrow A$, which implies f needs an argument from domain D_1 and one argument from D_2 ,..., and one from D_n to produce an answer belonging to domain A.
    - Description: Description of operation's mapping is generally given as equational definition.However, set graph,table or diagram can also be used.In this paper we use equational definitions.
- Valuation Function: A Valuation function is a collection of functions, one for each for each syntax domain. They map elements of syntax domain (Abstract syntax) to elements of semantic domain (Semantic algebra),for e.g. a valuation function can be used to map a query string(syntax domain) to its result relation(semantic domain).

4.1 Compositional Semantics of SPARQL

In this section we first describe the related formal notation and then give abstract syntax, semantic algebra of mapping sets and valuation functions as the compositional semantics for SPARQL.

- RDF terms: Denoted by T, it comprises of pairwise disjoint infinite sets of I,B and L (IRI's, Blank nodes and Literals respectively).

- Triple: A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where s is a subject, p a predicate, and o is an object.
- Triple Pattern: A triple pattern, tp, is a triple of the form $(sp, pp, op) \in (I \cup V \cup L) \times (I \cup V) \times (I \cup V \cup L)$, where V is an infinite set of variables that is pairwise disjoint from the sets I, B, and L; and *sp, pp,* and *op* are a subject pattern, predicate pattern, and object pattern respectively.
- A mapping μ is a partial function $\mu : V \rightarrow T$. Given a triple pattern t, $\mu(t)$ denotes the triple obtained by replacing the variables in t according to μ. Domain of μ, $dom(\mu)$ is the subset of V where μ is defined and Ω is a set of mappings μ. Two mappings μ_1, μ_2 are compatible when for all $x \in dom(\mu_1) \cap dom(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$.

Abstract Syntax for SPARQL
The abstract syntax shown below describes the structure of graph patterns expressions.
$gp \in graphpatterns$, where gp represents the syntax domain of graph patterns
$gp ::= tp \mid gp\ FILTER\ expr \mid PROJECT_{v_1,v_2,..,v_n}\ gp \mid gp\ UNION\ gp \mid gp\ AND\ gp \mid gp\ OPT\ gp$

Semantic Algebra for the domain of mapping sets
Semantic Domain: Mapping Set, Ω.
Operations defined over the semantic domain Ω.

1. The σ_{expr} operator takes a mapping set as an input and retains only the mappings that satisfy the expression expr and returns the resulting mapping set. Its functionality is denoted by $\sigma_{expr} : \Omega \rightarrow \Omega$ and description by:
   - $\sigma_{expr}\Omega = \{\mu | \mu \in \Omega \wedge \mu \vDash expr\}$
2. The $\prod_{vars}$ operator takes an input mapping set and retains only the variables belonging to the set $vars$ in each mapping and returns the resulting mapping set. Its functionality is given by $\prod_{v_1,v_2,..,v_n} : \Omega \rightarrow \Omega$ and description by:
   - $\prod_{v_1,v_2,..,v_n} \Omega = \{\mu_{|v_1,v_2,..,v_n} | \mu \in \Omega\}$
3. The Union operator $\cup$, takes as input two mapping sets and computes the set union and returns the resulting mapping set. Its functionality is given by $\cup : \Omega \times \Omega \rightarrow \Omega$ and description by:
   - $\Omega_1 \cup \Omega_2 = \{\mu | \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
4. The Join operator's functionality is given by $\bowtie$: $\Omega \times \Omega \rightarrow \Omega$, description by:
   - $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible mappings$\}$
5. Difference operator's functionality is $\smallsetminus : \Omega \times \Omega \rightarrow \Omega$, description is:
   - $\Omega_1 \smallsetminus \Omega_2 = \{\mu \in \Omega_1 |$ for all $\mu\prime \in \Omega_2, \mu$ and $\mu\prime$ are not compatible $\}$
6. The Optional operator's functionality is $⟕ : \Omega \times \Omega \rightarrow \Omega$, description is:
   - $\Omega_1 ⟕ \Omega_2 = \{(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2)\}$

Valuation functions
$[\![]\!]$: $gp \rightarrow \Omega$; The valuation function $[\![]\!]$, maps an element of gp to Ω, i.e it maps the language's abstract syntax structures to meanings drawn from semantic domains. The domain of a valuation function is a set of derivation trees of the language and is defined structurally. It determines meanings of its subtrees and combines them into meaning of the entire tree.

- $[\![tp]\!] = \{\mu | dom(\mu) = var(tp) \wedge \mu(tp) \in G\}$
- $[\![gp\ FILTER\ expr]\!] = \sigma_{expr}\ [\![gp]\!]$
- $[\![PROJECT_{v_1,v_2,..,v_n} gp]\!] = \prod_{v_1,v_2,..,v_n}\ [\![gp]\!]$
- $[\![gp_1\ UNION\ gp_2]\!] = [\![gp_1]\!] \cup [\![gp_2]\!]$
- $[\![gp_1\ AND\ gp_2]\!] = [\![gp_1]\!] \bowtie [\![gp_2]\!]$
- $[\![gp_1\ OPT\ gp_2]\!] = [\![gp_1]\!] ⟕ [\![gp_2]\!]$

The meaning of expression $[\![gp_1\ UNION\ gp_2]\!]$is computed by first evaluating $[\![gp_1]\!]$to get Ω_1 and $[\![gp_2]\!]$to get Ω_2 and then computing $\Omega_1 \cup \Omega_2$ using the mapping set's semantic algebra operation union defined earlier.
Given a mapping μ and expression *expr*, μ satisfies *expr* ($\mu \vDash expr$), iff

(i) *expr* is *bound*$(?X)$ and $?X \in dom(\mu)$;
(ii) *expr* is $?X\ op\ c$, $?X \in dom(\mu)$ and $\mu(?X)\ op\ c$;
(iii) *expr* is $?X\ op\ ?Y$, $?X, ?Y \in dom(\mu)$, and $\mu(?X)\ op\ \mu(?Y)$,
where $op \in \{< \mid \leq \mid \geq \mid > \mid =\}$;
(iv) *expr* is $(\neg expr_1)$ and it is not the case that $\mu \vDash expr_1$;
(v) *expr* is $(expr_1 \wedge expr_2)$ and $\mu \vDash expr_1$ and $\mu \vDash expr_2$;
(vi) *expr* is $(expr_1 \vee expr_2)$ and $\mu \vDash expr_1$ or $\mu \vDash expr_2$;

4.2 Compositional Semantics of Relational Algebra

The close relationship between SQL and Relational Algebra is the basis for query optimization in a RDBMS [10] and in our work it forms the basis for query translation. In order to simplify the proof we use relational algebra based syntax and define its compositional semantics by using tuple relational calculus formulas since they closely mirror the mapping based semantics of SPARQL. The proof of equivalence between relational algebra and relational calculus presented in [11] enables us to define the compositional semantics in this manner.

Abstract Syntax for Relational Algebra The abstract syntax describes the structure of relational algebra(RA) expressions.
$E \in Expression$, where E represents the syntax domain of RA expressions.
$E \rightarrow \sigma_{Cond}E | \prod_{A_1,A_2,..,A_n} E | E_1 \cup E_2 | E_1 \bowtie_{Cond} E_2 | E_1 \times E_2 | E_1 - E_2 | E_1 \cap E_2$
Semantic Algebra for the domain of relations
Semantic Domain: Relation, R ; A relation is a set of all tuples such that they are all defined over same schema or set of attributes.
Operations defined over the semantic domain R.

1. σ_{Cond} operator takes a relation as an input and retains only the tuples that satisfy the expression $Cond$ and returns the resulting relation. $Cond$ is a boolean expression defined in the abstract syntax of SQL. Its functionality is denoted by $\sigma_{Cond} : R \rightarrow R$ and description by:
   – $\sigma_{Cond}R = \{t | R(t) \wedge Cond(t)\}$ where R(t) denotes tuple t belongs to relation R.
2. The Project operator's functionality is $\prod_{A_1,A_2,..,A_n} : R \rightarrow R$, description is:
   – $\prod_{A_1,A_2,..,A_n} R = \{t.A_1, t.A_2, .., t.A_n | R(t)\}$
3. Union operator's functionality is $\cup : R \times R \rightarrow R$ and description is:
   – $R_1 \cup R_2 = \{t | (R_1(t) \vee R_2(t)) \wedge (\xi(R_1) \equiv \xi(R_2))\}$
   Union operation can be performed only over union compatible relations i.e. their schemas should be identical and it is denoted by the condition $(\xi(R_1) \equiv \xi(R_2))$ where $(\xi(R)$ stands for schema of a relation R.
4. Intersection operator's functionality is $\cap : R \times R \rightarrow R$ and description is:
   – $R_1 \cap R_2 = \{t | (R_1(t) \wedge R_2(t)) \wedge (\xi(R_1) \equiv \xi(R_2))\}$
5. Cross product operator's functionality is $\times : R \times R \rightarrow R$ and description is:
   – $R_1 \times R_2 = \{t | \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge ... \wedge t.A_n = t_1.A_n \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2 \wedge ... \wedge t.B_n = t_2.B_n)\}$
   where $A_1, A_2, .., A_n$ are attributes of relation R_1 and $B_1, B_2, .., B_n$ are attributes of R_2.
6. The Join operator's functionality is $\bowtie_{Cond}$: $R_1 \times R_2 \rightarrow R$ and description is:
   – $R_1 \bowtie_{Cond} R_2 = \{t | \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge ... \wedge t.A_n = t_1.A_n \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2 \wedge ... \wedge t.B_n = t_2.B_n \wedge Cond(t))\}$
7. Difference operator's functionality is $- : R \times R \rightarrow R$ and description is:
   – $R_1 - R_2 = \{t | R_1(t) \wedge \forall t_2 (R_2(t_2) \rightarrow t \neq t_2)\}$

Valuation functions
$[\![]\!]_r$: $E \rightarrow R$; The valuation function $[\![]\!]_r$, maps an element of syntax domain E to element of semantic domain R.

– $[\![\sigma_{Cond}E]\!]_r = \sigma_{Cond}[\![E]\!]_r$
– $[\![\prod_{A_1,A_2,..,A_n} E]\!]_r = \prod_{A_1,A_2,..,A_n}[\![E]\!]_r$
– $[\![E_1 \cup E_2]\!]_r = [\![E_1]\!]_r \cup [\![E_2]\!]_r$
– $[\![E_1 \bowtie_{Cond} E_2]\!]_r = [\![E_1]\!]_r \bowtie_{Cond} [\![E_2]\!]_r$
– $[\![E_1 \times E_2]\!]_r = [\![E_1]\!]_r \times [\![E_2]\!]_r$
– $[\![E_1 - E_2]\!]_r = [\![E_1]\!]_r - [\![E_2]\!]_r$
– $[\![E_1 \cap E_2]\!]_r = [\![E_1]\!]_r \cap [\![E_2]\!]_r$

Theorem 1. *Given an RDF dataset D, an SQL query, $sql \in SqlQuery$, with an equivalent relational algebra expression $RA \in E$ over a relational schema S derived from D, the translation of sql to an equivalent SPARQL query, sparql(or an equivalent sparql algebra expression SA), over dataset D, is a semantics preserving translation.*

Proof. A query is made up of operators and operands and two given queries are equivalent if they are made up of semantically equivalent operators applied on equivalent data in the same sequence. Having defined the compositional semantics for Relational Algebra and SPARQL, proving the semantic equivalence boils down to equating their respective semantic algebras.

- **Selection** If $\Omega \equiv [\![tp\ ]\!]$, *Cond* and *expr* are equivalent boolean conditions and $R \in S$ such that $tp =$?s R ?o then $RA \equiv SA$.
  RA: $\sigma_{Cond} R = \{t | R(t) \wedge Cond(t)\}$
  SA: $Filter_{expr} \Omega = \{\mu | \mu \in \Omega \wedge expr(\mu)\}$
  The conditions $\Omega \equiv [\![tp\ ]\!]$and $R \in S$ such that $tp =$?s R ?o ensure the data equivalence and RA $\equiv$ SA stand for operator equivalence.
- **Projection** If $\Omega \equiv [\![tp\ ]\!]$, $\{\forall i, A_i \equiv v_i, 1 \leq i \leq n\}$ and $R \in S$ such that $tp =$?s R ?o then $RA \equiv SA$.
  RA: $\prod_{A_1,A_2,..,A_n} R = \{t.A_1, t.A_2, .., t.A_n | R(t)\}$
  SA: $Project_{v_1,v_2,..,v_n} \Omega = \{\mu_{|v_1,v_2,..,v_n} | \mu \in \Omega\}$
  The conditions representing data equivalence are $\Omega \equiv [\![tp\ ]\!]$, and $R \in S$ such that $tp =$?s R ?o and operator equivalence is represented by algebraic expressions RA and SA and $\{\forall i, A_i \equiv v_i, 1 \leq i \leq n\}$.
- **Cross product** If $\Omega_1 \equiv [\![tp_1\ ]\!]$,$\Omega_2 \equiv [\![tp_2\ ]\!]$and $R_1, R_2 \in S$ such that $tp_1 =?s_1R_1?o_1$, $tp_2 =?s_2R_2?o_2$ then $RA \equiv SA$.
  RA: $R_1 \times R_2 = \{t | \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2)\}$
  SA: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible mappings$\}$
  The conditions that ensure data equivalence are $\Omega_1 \equiv [\![tp_1\ ]\!]$,$\Omega_2 \equiv [\![tp_2\ ]\!]$and $R_1, R_2 \in S$ such that $tp_1 =?s_1R_1?o_1$, $tp_2 =?s_2R_2?o_2$ and the operator equivalence is shown by algebraic expressions RA and SA.
- **Join** If $\Omega_1 \equiv [\![tp_1\ ]\!]$,$\Omega_2 \equiv [\![tp_2\ ]\!]$, *Cond* and *expr* are equivalent boolean conditions, and $R_1, R_2 \in S$ such that $tp_1 =?s_1R_1?o_1$, $tp_2 =?s_1R_2?o_2$ then $RA \equiv SA3$.
  RA: $R_1 \bowtie_{Cond} R_2 = \{t | \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2 \wedge Cond(t))\}$
  SA1: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible mappings$\}$
  SA2: $Filter_{expr} \Omega = \{\mu | \mu \in \Omega \wedge expr(\mu)\}$
  SA3: $Filter_{expr}(\Omega_1 \bowtie \Omega_2)$
  Semantics of *SA3* is composed of semantics of *SA1* and *SA2*. The data equivalence is ensured by conditions $\Omega_1 \equiv [\![tp_1\ ]\!]$,$\Omega_2 \equiv [\![tp_2\ ]\!]$, and $R_1, R_2 \in S$ such that $tp_1 =?s_1R_1?o_1$, $tp_2 =?s_1R_2?o_2$ and operation equivalence is shown by euivalence of RA and $SA3$ and by equivalence of boolean expressions *Cond* and *expr*.
- **Union** If $\Omega_1 \equiv [\![gp_1\ ]\!]$, $\Omega_2 \equiv [\![gp_2\ ]\!]$, $gp_1 = tp_1\ AND\ tp_2$, $gp_2 = tp_1\ AND\ tp_3$, and $R_1, R_2, R_3 \in S$ such that $tp_1 =?s_1R_1?o_1$, $tp_2 =?s_1R_2?o_2$, $tp_3 =?s_1R_3?o_3$ and $R_x = \prod_{R_1.s,R_1.o}(R_1 \bowtie R_2)$ and $R_y = \prod_{R_1.s,R_1.o}(R_1 \bowtie R_3)$, then $RA \equiv SA$.
  RA: $R_x \cup R_y = \{t | (R_x(t) \vee R_y(t)) \wedge (\xi(R_x) \equiv \xi(R_y))\}$
  SA: $\prod_{?s_1,?o_1}(\Omega_1 \cup \Omega_2) = \{\mu_{|s1,o1} | \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
- **Intersection** If $\Omega_1 \equiv [\![gp_1\ ]\!]$, $\Omega_2 \equiv [\![gp_2\ ]\!]$, $gp_1 = tp_1\ AND\ tp_2$, $gp_2 = tp_1\ AND\ tp_3$, and $R_1, R_2, R_3 \in S$ such that $tp_1 =?s_1R_1?o_1$, $tp_2 =?s_1R_2?o_2$, $tp_3 =?s_1R_3?o_3$ and $R_x = \prod_{R_1.s,R_1.o}(R_1 \bowtie R_2)$ and $R_y = \prod_{R_1.s,R_1.o}(R_1 \bowtie R_3)$, then $RA \equiv SA$.
  RA: $R_x \cap R_y = \{t | (R_x(t) \wedge R_y(t)) \wedge (\xi(R_x) \equiv \xi(R_y))\}$
  SA: $\prod_{?s_1,?o_1}(\Omega_1 \bowtie \Omega_2) = \{(\mu_1 \cup \mu_2)_{|s1,o1} | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ are compatible mappings$\}$
- **Difference** If $\Omega_1 \equiv [\![gp_1\ ]\!]$, $\Omega_2 \equiv [\![gp_2\ ]\!]$, $R \in S$ such that $tp_1 =?s_1R?o_1$, $tp_2 =?s_1R?o_2$, $gp_1 = FILTER_{?o_1\ Op\ \text{“}Value1''}(tp_1)$, $gp_2 = FILTER_{?o_1\ Op\ \text{“}Value2''}(tp_2)$, $R_1 = \prod_{R.s}(\sigma_{R.o\ Op\ \text{“}Value1''}(R))$, $R_2 = \prod_{R.s}(\sigma_{R.o\ Op\ \text{“}Value2''}(R))$, $expr \equiv \neg bound(?o_2)$, then $RA \equiv SA3$.
  RA: $R_1 - R_2 = \{t | R_1(t) \wedge \forall t_2 (R_2(t_2) \wedge t \neq t_2) \wedge (\xi(R_1) \equiv \xi(R_2))\}$
  SA1: $\Omega_1 ⟕ \Omega_2 = \{\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}$
  SA2: $Filter_{expr} \Omega = \{\mu | \mu \in \Omega \wedge expr(\mu)\}$
  SA3: $\prod_{?s_1}(Filter_{expr}(\Omega_1 ⟕ \Omega_2))$
  Semantics of *SA3* is composed of semantics of *SA1* and *SA2*.

Since, all the operators in relational algebra are proved equivalent to corresponding operators in SPARQL, it can be concluded that $\forall sql \in SqlQuery$, where RA is relational algebra expression of *sql*, $[\![RA]\!]_r \equiv [\![sparql\ ]\!]$

5 Related work

There have been several attempts to make RDBMS and RDF stores inter-operate. The most popular one, D2RQ [12], has explored one direction i.e they construct an RDF schema from Relational data. A similar effort, W3C RDB2RDF WG [13], aims to generate RDF triples from one or more Relational tables without loss of information. In [14], the author describes a transformation from SPARQL to relational algebra and attempts to make existing work on query planning and optmization available to SPARQL implementors. RETRO provides the translation in reverse direction and proves the translation to be semantics preserving by reusing the work done on compositional semantics of SPARQL in [6] and [1]. Compositional semantics of SPARQL discussed in [6] is reused and extended by authors in [1] by providing additional operators such as $FILTER_{expr}$ and $SELECT(v_1, v_2..v_n)$. However, unlike [6] and [1], we explicitly define compositional semantics in terms of valuation functions, which map elements of syntax domain to elements of semantic domain and use it towards the proof of semantics preservation of the query translation.

6 Conclusion and Future Work

In this paper we have provided interoperability between RDF Stores and RDBMS by firstly deriving a relational schema from the RDF store and secondly providing a means to translate an SQL query to semantically equivalent SPARQL query. In future versions of RETRO we plan to provide an algorithm for deriving a user friendly relational schema and extend the query mapping algorithm with additional and advanced SQL operators such as aggregation. Another promising direction for the future work is to leverage the query optimization available for SQL queries to generate efficient, equivalent SPARQL queries.

References

1. Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving sparql-to-sql translation. *Data Knowl. Eng.*, 68(10):973–1000, 2009.
2. Boris Motik, Ian Horrocks, and Ulrike Sattler. Bridging the gap between owl and relational databases. *J. Web Sem.*, 7(2):74–89, 2009.
3. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, 2003.
4. Christian Bizer and Richard Cyganiak. D2rq lessons learned. `http://www.w3.org/2007/03/RdfRDB/papers/d2rq-positionpaper/`.
5. Daniel J. Abadi, Adam Marcus 0002, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.
6. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International Semantic Web Conference*, pages 30–43, 2006.
7. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.
8. E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.
9. Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *J. Artif. Intell. Res. (JAIR)*, 11:199–240, 1999.
10. Raghu Ramakrishnan, Johannes Gehrke, Raghu Ramakrishnan, and Johannes Gehrke. *Database Management Systems.* McGraw-Hill Science/Engineering/Math, 3 edition, August 2002.
11. E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.
12. C. Bizer and A. Seaborne. D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs. Poster at 3rd International Semantic Web Conference, 2004.
13. A. Malhotra. W3C RDB2RDF Incubator Group Report. `http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/`, 2009.
14. Richard Cyganiak. A relational algebra for SPARQL. 2005.