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, Denotational Semantics.

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 [2]. The most prominent feature of RDBMS is not reasoning about schema [3], 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 [4, 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 use 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 it has 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 understanding its underlying principles, tools and technologies, it becomes imperative to provide them with a means to conveniently query the existing RDF data. Integrating various databases is often a motivation for adopting RDF [5] and this in turn 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 an 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 [6] 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:** 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 [7], the assumptions made in it also hold in this paper, which we briefly restate here. Algebraic formalization of 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 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 [7], [1] and [9]. Finally, we give a proof of semantics preservation of the translation based on compositional semantics of the two query languages.

2 RETRO Framework: Schema Mapping

ER model and Description logics both model a given domain using classes and relationships. Calvanese el al., in [10] define a translation function that transforms ER model into a ALUNI, DL knowledge base [10] 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 over the given database instance, 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 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 $ABox\cdot size$ do
4: $T \leftarrow ABox[i]$ [Retrieve a triple $T$ from the $ABox$]
5: $p \leftarrow T\cdot predicate$ [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 the users. It is constructed by extracting each of the predicate names from the map and adding attributes \( s \) and \( o \). A relation in the schema stands for relation instance which can be obtained by evaluation of the specific triple pattern over the triple store. These triple patterns belong to the set: \( \{ tp.t.p.s = ?s, \land tp.p \in P \land tp.o = ?o, \forall 1 \leq i \leq n \} \). 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 query mapping algorithm which uses it to unambiguously generate triple patterns.

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 of 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 consider the following RDF dataset \( D \), which is an extension of the dataset provided in [7]. We also provide the relational schema \( S \) generated by the application of the Schema-Mapping algorithm to the dataset \( D \).

\[
S = \{ \text{name}(s, o), \text{age}(s, o), \text{sal}(s, o), \text{webPage}(s, o), \\
\text{dept}(s, o), \text{phone}(s, o), \text{email}(s, o) \}
\]

<table>
<thead>
<tr>
<th>( B_1 ), name, paul</th>
<th>( B_2 ), name, john</th>
<th>( B_3 ), name, george</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_4 ), name, ringo</td>
<td>( B_1 ), age, 30</td>
<td>( B_2 ), age, 40</td>
</tr>
<tr>
<td>( B_3 ), age, 25</td>
<td>( B_4 ), age, 35</td>
<td>( B_1 ), sal, 20,000</td>
</tr>
<tr>
<td>( B_2 ), sal, 30,000</td>
<td>( B_3 ), sal, 40,000</td>
<td>( B_4 ), sal, 50,000</td>
</tr>
<tr>
<td>( B_2 ), dept, Research</td>
<td>( B_3 ), dept, Research</td>
<td>( B_3 ), dept, Admin</td>
</tr>
<tr>
<td>( B_4 ), dept, Admin</td>
<td>( B_1 ), phone, 777-3426</td>
<td>( B_4 ), phone, 888-4537</td>
</tr>
<tr>
<td>( B_2 ), email, <a href="mailto:john@acd.edu">john@acd.edu</a></td>
<td>( B_4 ), email, <a href="mailto:ringo@acd.edu">ringo@acd.edu</a></td>
<td>( B_3 ), webPage, <a href="http://www.george.edu">www.george.edu</a></td>
</tr>
<tr>
<td>( B_4 ), webPage, <a href="http://www.starr.edu">www.starr.edu</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. The RDF dataset \( D \)

In Tables 2 and 3 we give a description of an SQL query on a relational schema \( S \) and give its equivalent SPARQL query over dataset \( D \) and subsequently, we show the result of evaluating the SPARQL query on the 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.
### Table 2: Illustration of SQL-to-SPARQL Translation on Dataset D

<table>
<thead>
<tr>
<th>Description</th>
<th>SQL Query</th>
<th>SPARQL Query</th>
<th>ResultSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross Product of relations email and webPage</td>
<td>$\text{SELECT email.s, email.o, webPage.s, webPage.o}$</td>
<td>$\text{SELECT ?s1 ?o1 ?s2 ?o2}$</td>
<td>$\text{<a href="mailto:john@acd.edu">john@acd.edu</a>}$ $\text{www.george.edu}$ $\text{<a href="mailto:ringo@acd.edu">ringo@acd.edu</a>}$ $\text{www.starr.edu}$</td>
</tr>
<tr>
<td>Find the email id of john</td>
<td>$\text{SELECT email.o}$</td>
<td>$\text{SELECT ?o2}$</td>
<td>$\text{john}$</td>
</tr>
<tr>
<td>Names of all people with age $25$ and salary $&gt;30K$</td>
<td>$\text{SELECT name.s, name.o}$</td>
<td>$\text{SELECT DISTINCT ?s1 ?o1}$</td>
<td>$\text{george}$</td>
</tr>
<tr>
<td>Find the email id of john and salary $&gt;30,000$</td>
<td>$\text{SELECT name.s, name.o}$</td>
<td>$\text{SELECT ?o2}$</td>
<td>$\text{john}$</td>
</tr>
<tr>
<td>Description</td>
<td>SQL Query</td>
<td>SPARQL Query</td>
<td>ResultSet</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Names of all people who either have a phone number or an email id</td>
<td>SELECT name.o FROM name, email WHERE name.s = email.s UNION SELECT name.o FROM name, phone WHERE name.s = phone.s</td>
<td>SELECT ?o1 WHERE { ?s name ?o1</td>
<td>paul \n</td>
</tr>
</tbody>
</table>

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

```
SqlQuery  -->  CrossProductQuery | SPJQuery | UnionQuery | IntersectQuery | ExceptQuery.
CrossProductQuery  -->  SelectClause FromClause.
SPJQuery  -->  SelectClause FromClause WhereClause.
UnionQuery  -->  SqlQuery 'UNION' SqlQuery.
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  -->  Exp Op Exp.
Op  -->  '<' | '>' | '=' | '>=' | '<=' | '<>' | '='.
Lop  -->  'AND' | 'OR' | 'NOT' .
```
3.2 Query Mapping algorithms

The Query-Mapping algorithm is the main algorithm that follows the conceptual query evaluation strategy of SQL. The main 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 procedure could also simply return the corresponding triple patterns but instead returns the map as it allows the next sub-procedure TransSQLWhereClause to index the triple patterns based on SQL table names. The Query-Mapping algorithm next evaluates the WHERE clause of the input SQL query using the sub-procedure TransSQLWhereClause. This sub-procedure takes as input \( JC \), \( BE \) and the map \( TP \). 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 relation names within the conditions of the set \( JC \). For example if relations age and dept have ranks 2 and 3 respectively, a join condition such as “dept.s = age.s” gets transformed to “age.s = dept.s”. This is accomplished in the algorithm using the function th sort(\( JC \), map) which reorders attributes within a join condition. The main complexity of the algorithm lies in generating triple patterns that are semantically equivalent to the join of the tables in the input SQL query. Once the appropriate triple patterns have been generated the next step is to generate the filter conditions which are semantically equivalent to the selection conditions from the SQL query. Finally, the sub-procedure returns the semantically equivalent WHERE clause string.

**Algorithm 2** TransSQLFromClause()

| Input: A set of table names from a SQL FROM clause, \( R \) |
| Output: A map from table names to triple patterns, \( TP \) |

\[
\begin{align*}
  &1: \text{ } TP \leftarrow \emptyset \{ \text{The map } TP \text{ is initially empty}\} \\
  &2: \text{ } \text{for } i \leftarrow 1 \text{ to } R.size \text{ do} \\
  &3: \text{ } tp \leftarrow \{?s, r_i, ?o_i\} \{ \text{A triple pattern is constructed for each } r_i \in R\} \\
  &4: \text{ } TP.put(r_i, tp) \\
  &5: \text{ } \text{end for} \\
  &6: \text{ } \text{return } TP
\end{align*}
\]

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 of the form “relationName.attributeName” from an SQL SELECT clause. During the iteration it uses the relation name part of the SQL SELECT attribute to access the corresponding triple pattern from the map and further it uses the attribute name part of the SQL SELECT attribute to access the desired variable within the this triple pattern.
Algorithm 3 TransSQLWhereClause()

Input: JC, BE, TP, P
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 ∈ TP do where += tp.subject + " " + tp.predicate + " " + tp.object
4: end for
5: end if
6: JC = SORT(JC, P)
7: for each condition p ∈ JC do
8:   p_1 = p.lOperand; p_2 = p.rOperand;
9:   tp_1 ← TP.get(p_1.relation) {Get the triple pattern for p.lOperand}
10:  tp_2 ← TP.get(p_2.relation) {Get the triple pattern for p.rOperand}
11:  where += tp_1.subject + " " + tp_1.predicate + " " + tp_1.object
12:  {The triple pattern for p.lOperand is always appended as it is to the output}
13: end for
14: for each expression p ∈ BE do
15:   p_1 = p.lOperand; p_2 = p.operator; p_3 = p.rOperand;
16:   tp_1 = TP.get(p_1.relation) {Get the triple pattern for p.lOperand}
17:   if p_1.attribute = "s" then
18:     where += FILTER(" + tp_1.subject + " " + p_2 + " " + p_3 + ")"
19:   else
20:     where += FILTER(" + tp_1.object + " " + p_2 + " " + p_3 + ")"
21:   end if
22: end for
23: return where
Algorithm 4 TRANSSQLSELECTCLAUSE()
Input: A set of SQL SELECT attributes, A, and a map, TP (relation name → triple pattern)
Output: A SPARQL SELECT clause
1: \( q^{\text{out}} = "\) { A SPARQL SELECT clause that is initially blank }
2: for each attribute \( a \in A \) do
3: \( r_{\text{name}} \leftarrow a.\text{relation} \); \( a_{\text{name}} \leftarrow a.\text{attribute} \)
4: \( tp \leftarrow TP.\text{get}(r_{\text{name}}) \) { Get the triple pattern for \( r_{\text{name}} \) }
5: if \( a_{\text{name}} = "s" \) then \( \text{select } + = \text{tp.subject } + \ " \) 
6: else \( \text{select } + = \text{tp.object } + \ " \) end if
7: end for
8: return \( \text{select} \)

Algorithm 5 QUERY-MAPPING()
Input: An SQL query, \( q_\text{in} \), and a map \( P \)
Output: A SPARQL query, \( q_\text{out} \)
1: \( q_\text{out} = "\) { A SPARQL query that is initially blank }
2: \( q_\text{out} = \text{parse}(q_\text{in}) \) { A parse tree obtained by parsing \( q_\text{in} \) }
3: \( q_\text{WHERE} = \text{tree.getSelectClause}() \); \( q_\text{FROM} = \text{tree.getFromClause}() \)
4: \( q_\text{WHERE-JC} = \text{tree.getJoinConditions}() \); \( q_\text{WHERE-BE} = \text{tree.getBooleanExpr}() \)
5: \( q_\text{SELECT} = \text{"SELECT "} \); \( q_\text{WHERE} = \text{"WHERE \{ "} \)
6: \( TP \leftarrow \varnothing \) { The map of triple patterns is initially empty }
7: if \( \text{tree.type} = \text{CrossProductQuery} \) then
8: \( TP = \text{TRANSSQLFROMCLAUSE}(q_\text{FROM}) \)
9: \( q_\text{WHERE} = \text{TRANSSQLWHERECLAUSE}(q_\text{WHERE-JC}, \text{WHERE-BE}, q_\text{WHERE}) \)
10: \( q_\text{SELECT} = \text{TRANSSQLSELECTCLAUSE}(q_\text{SELECT}, TP) \)
11: \( q_\text{out} = q_\text{SELECT} + q_\text{WHERE} + " \)
12: else if \( \text{tree.type} = \text{SPJQuery} \) then
13: \( TP = \text{TRANSSQLFROMCLAUSE}(q_\text{FROM}) \)
14: \( q_\text{WHERE} = \text{TRANSSQLWHERECLAUSE}(q_\text{WHERE-JC}, q_\text{WHERE-BE}, TP, P) \)
15: \( q_\text{SELECT} = \text{TRANSSQLSELECTCLAUSE}(q_\text{SELECT}, TP) \)
16: \( q_\text{out} = q_\text{SELECT} + q_\text{WHERE} + " \)
17: else if \( \text{tree.type} = \text{UnionQuery} \) then
18: \( q_1 = \text{tree.leftSubTree}() \); \( q_2 = \text{tree.rightSubTree}() \)
19: \( q_1^{\text{out}} = \text{QUERY-MAPPING}(q_1) \); \( q_2^{\text{out}} = \text{QUERY-MAPPING}(q_2) \)
20: \( q_\text{out} = \text{Merge}(q_1^{\text{out}}, q_2^{\text{out}}, "\text{UNION}\)"
21: else if \( \text{tree.type} = \text{IntersectQuery} \) then
22: \( q_1 = \text{tree.leftSubTree}() \); \( q_2 = \text{tree.rightSubTree}() \)
23: \( q_1^{\text{out}} = \text{QUERY-MAPPING}(q_1) \); \( q_2^{\text{out}} = \text{QUERY-MAPPING}(q_2) \)
24: \( q_\text{out} = \text{Merge}(q_1^{\text{out}}, q_2^{\text{out}}, "\text{INTERSECT}\)"
25: else if \( \text{tree.type} = \text{ExceptQuery} \) then
26: \( q_1 = \text{tree.leftSubTree}() \); \( q_2 = \text{tree.rightSubTree}() \)
27: \( q_1^{\text{out}} = \text{QUERY-MAPPING}(q_1) \); \( q_2^{\text{out}} = \text{QUERY-MAPPING}(q_2) \)
28: \( q_\text{out} = \text{Merge}(q_1^{\text{out}}, q_2^{\text{out}}, "\text{EXCEPT}\)"
29: end if
30: return \( q_\text{out} \)
The main procedure Query-Mapping takes as input an SQL query string and a map P generated from Schema-Mapping algorithm and returns a SPARQL query string. The SQL query string is used to generate its parse tree using function parse(SQLQuery). We then extract SQL SELECT clause, SQL FROM clause, SQL WHERE-JC clause (Join Conditions) and SQL 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. If the type of the query is SPJQuery then we first call the sub-procedure TransSQLFromClause. The result generated is a map TP which is then used in the call to procedure TransSQLWhereClause with WHERE-JC and WHERE-BE conditions additionally provided. It returns a SPARQL WHERE clause string which is appended to the result of the evaluation of TransSQLSelectClause. Since SQL UNION, INTERSECT and EXCEPT queries are 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.

The sub-procedure Merge, merges the two input sub-query strings to generate 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. It then extracts and stores the triple patterns from the WHERE clauses of the two sub-queries in variables $W_1$ and $W_2$. Finally, it extracts and stores the boolean conditions from the FILTER expressions of each of the sub-queries into variables $F_1$ and $F_2$. 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. We repeat the same procedure to generate the WHERE clause for INTERSECT query and do an additional renaming of the variables. The renaming is done only to the variables of the set of triple patterns ($W_2$) that belong to the second sub-query. If these variables do not 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 belonging to $W_2$ that do not belong to the set $V_1$. 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 $V_1$. The sub-procedure implicitly checks that only variable names are renamed and does not manipulate literal values.
Algorithm 6 \textsc{Merge()} \\
\textbf{Input}: \(q_1, q_2, Q\_TYPE\) \\
\textbf{Output}: A SPARQL query \(q_{out}\) \\
1: \(q_{out} = "\) \{A SPARQL query that is initially blank\} \\
2: \(V_1 = q_1.\text{extractSelectClause}(); V_2 = q_2.\text{extractSelectClause}();\) \\
3: \(W_1 = q_1.\text{extractTriplePatterns}(); W_2 = q_2.\text{extractTriplePatterns}();\) \\
4: \(F_1 = q_1.\text{extractFilter}(); F_2 = q_2.\text{extractFilter}();\) \\
5: \textbf{if} \(Q\_TYPE = "\text{UNION}\) \textbf{then} \\
6: \(q_{out} + = "\text{SELECT DISTINCT}"\) \\
7: \textbf{for} \(i \leftarrow 1 \text{ to } V_1.\text{size} \textbf{do} \ q_{out} + = V_i[i]; \textbf{end for}\) \\
8: \(q_{out} + = "\text{WHERE} \{ \text{"}\) \\
9: \textbf{for} \(i \leftarrow 1 \text{ to } W_1.\text{size} \textbf{do} \ q_{out} + = W_i[i]; \textbf{end for}\) \\
10: \(q_{out} + = "\text{UNION} \{ "\) \\
11: \textbf{for} \(i \leftarrow 1 \text{ to } F_1.\text{size} \textbf{do} \ q_{out} + = F_i[i]; \textbf{end for}\) \\
12: \(q_{out} + = "\} \text{"}\) \\
13: \textbf{else if} \(Q\_TYPE = "\text{INTERSECT}\) \textbf{then} \\
14: \(q_{out} + = "\text{SELECT DISTINCT}"\) \\
15: \textbf{for} \(i \leftarrow 1 \text{ to } V_1.\text{size} \textbf{do} \ q_{out} + = V_i[i]; \textbf{end for}\) \\
16: \(q_{out} + = "\text{WHERE} \{ \text{"}\) \\
17: \textbf{for} \(i \leftarrow 1 \text{ to } W_1.\text{size} \textbf{do} \ q_{out} + = W_i[i]; \textbf{end for}\) \\
18: \(q_{out} + = "\text{OPTIONAL} \{ "\) \\
19: \textbf{for} \(i \leftarrow 1 \text{ to } F_1.\text{size} \textbf{do} \ q_{out} + = F_i[i]; \textbf{end for}\) \\
20: \(q_{out} + = "\} \text{"}\) \\
21: \textbf{else if} \(Q\_TYPE = "\text{EXCEPT}\) \textbf{then} \\
22: \(q_{out} + = "\text{SELECT}"\) \\
23: \textbf{for} \(i \leftarrow 1 \text{ to } V_1.\text{size} \textbf{do} \ q_{out} + = V_i[i]; \textbf{end for}\) \\
24: \(q_{out} + = "\text{WHERE} \{ \text{"}\) \\
25: \textbf{if} \(W_2[i].\text{subject} \notin V_1 \textbf{then} W_2[i].\text{subject} = \text{uniqVar}(W_2[i].\text{subject}) \textbf{end if}\) \\
26: \textbf{if} \(W_2[i].\text{object} \notin V_1 \textbf{then} W_2[i].\text{object} = \text{uniqVar}(W_2[i].\text{object}) \textbf{end if}\) \\
27: \(q_{out} + = W_2[i];\) \\
28: \textbf{end for}\) \\
29: \(q_{out} + = "\});\) \\
30: \textbf{else if} \(Q\_TYPE = "\text{EXCEPT}\) \textbf{then} \\
33: \(q_{out} + = "\text{WHERE} \{ \text{"}\) \\
34: \textbf{for} \(i \leftarrow 1 \text{ to } V_1.\text{size} \textbf{do} \ q_{out} + = V_i[i]; \textbf{end for}\) \\
35: \(q_{out} + = "\text{OPTIONAL} \{ "\) \\
36: \textbf{for} \(i \leftarrow 1 \text{ to } F_1.\text{size} \textbf{do} \ q_{out} + = F_i[i]; \textbf{end for}\) \\
37: \(q_{out} + = "\} \text{"}\) \\
38: \textbf{for} \(i \leftarrow 1 \text{ to } W_2.\text{size} \textbf{do}\) \\
39: \(\textbf{if} \ W_2[i].\text{subject} \notin V_1 \textbf{then} W_2[i].\text{subject} = \text{uniqVar}(W_2[i].\text{subject}) \textbf{end if}\) \\
40: \(\textbf{if} \ W_2[i].\text{object} \notin V_1 \textbf{then} W_2[i].\text{object} = \text{uniqVar}(W_2[i].\text{object}) \textbf{end if}\) \\
41: \(q_{out} + = W_2[i];\) \\
42: \textbf{end for}\) \\
43: \(q_{out} + = F_2[i];\) \\
44: \textbf{end if}\) \\
45: \textbf{else if} \(Q\_TYPE = "\text{EXCEPT}\) \textbf{then} \\
48: \(q_{out} + = F_2[i];\) \\
49: \textbf{end if}\) \\
50: \textbf{return} \(q_{out}\)
4 Semantics Preserving SQL-To-SPARQL Translation

An important principle of compositional semantics is that 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 is commonly specified as BNF grammar 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. 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. 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).

4.1 Syntax and Semantics of SPARQL

In this section we 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 \(\mu\) 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 \(\mu\). Domain of \(\mu\), \(\text{dom}(\mu)\) is the subset of \(V\) where \(\mu\) is defined and \(\Omega\) is a set of mappings \(\mu\). Two mappings \(\mu_1, \mu_2\) are compatible when for all \(x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)\), it is the case that \(\mu_1(x) = \mu_2(x)\).

Abstract Syntax for SPARQL

\(gp \in \text{graphpatterns}\)
\(gp := tp \mid gp \text{ FILTER expr} \mid \text{PROJECT}_{v_1,v_2,\ldots,v_n} \text{ gp} \mid \text{gp UNION} \text{ gp} \mid \text{gp AND} \text{ gp} \mid \text{gp OPT} \text{ gp}\)

Semantic Algebra for the domain of mapping sets

Domain Mapping Set: \(\Omega \equiv \{[tp] = \{\mu|\text{dom}(\mu) = \text{var}(tp) \land \mu(tp) \in G\}\}

1. \(\sigma_{\text{expr}}\Omega = \{\mu|\mu \in \Omega \land \mu \vdash \text{expr}\}\)
2. \(\Pi_{v_1,v_2,\ldots,v_n}\Omega = \{\mu|v_1,v_2,\ldots,v_n|\mu \in \Omega\}\)
3. \(\Omega_1 \cup \Omega_2 = \{\mu|\mu \in \Omega_1 \lor \mu \in \Omega_2\}\)
4. \(\Omega_1 \otimes \Omega_2 = \{\mu_1 \otimes \mu_2|\mu_1 \in \Omega_1, \mu_2 \in \Omega_2\text{ are compatible mappings}\}\)
5. \(\Omega_1 \setminus \Omega_2 = \{\mu|\mu \in \Omega_1\text{ for all }\mu' \in \Omega_2, \mu \text{ and } \mu'\text{ are not compatible}\}\)
6. $\Omega_1 \mathcal{R} \Omega_2 = \{(\Omega_1 \mathcal{R} \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}$

Valuation functions
\[
\mathcal{[[\cdot]]}: \mathcal{E} \rightarrow \Omega; \quad \text{The valuation function } \mathcal{[[\cdot]]}, \text{ maps an element of } \mathcal{E} \text{ to } \Omega
\]
- $\mathcal{[[t\mathcal{P}]]} = \{\mu|\text{dom}(\mu) = \text{var}(t\mathcal{P}) \land \mu(t\mathcal{P}) \in G\}$
- $\mathcal{[[\mathcal{G} \mathcal{F}\mathcal{I}\mathcal{T} \mathcal{R} \mathcal{E}] \mathcal{F}]]}$ = $\mathcal{[[\mathcal{G} \mathcal{F}]]}$
- $\mathcal{[[\mathcal{P} \mathcal{R} \mathcal{O} \mathcal{J} \mathcal{E} \mathcal{T} \mathcal{C} \mathcal{T} \mathcal{R} \mathcal{I} \mathcal{Y} \mathcal{C} \mathcal{L} \mathcal{U} \mathcal{C} \mathcal{L} \mathcal{E}] \mathcal{F}]]}$ = $\mathcal{[[\mathcal{P} \mathcal{R} \mathcal{O} \mathcal{J} \mathcal{E} \mathcal{T} \mathcal{C} \mathcal{T} \mathcal{R} \mathcal{I} \mathcal{Y} \mathcal{C} \mathcal{L} \mathcal{E}] \mathcal{F}]]}$
- $\mathcal{[[\mathcal{G} \mathcal{P} \mathcal{N} \mathcal{I} \mathcal{O} \mathcal{N} \mathcal{E} \mathcal{N} \mathcal{N} \mathcal{E}] \mathcal{F}]]}$ = $\mathcal{[[\mathcal{G} \mathcal{P} \mathcal{N} \mathcal{I} \mathcal{O} \mathcal{N} \mathcal{E} \mathcal{N} \mathcal{N} \mathcal{E}] \mathcal{F}]]}$
- $\mathcal{[[\mathcal{G} \mathcal{P} \mathcal{N} \mathcal{I} \mathcal{O} \mathcal{N} \mathcal{E} \mathcal{N} \mathcal{N} \mathcal{E}] \mathcal{F}]]}$ = $\mathcal{[[\mathcal{G} \mathcal{P} \mathcal{N} \mathcal{I} \mathcal{O} \mathcal{N} \mathcal{E} \mathcal{N} \mathcal{N} \mathcal{E}] \mathcal{F}]]}$
- $\mathcal{[[\mathcal{G} \mathcal{P} \mathcal{N} \mathcal{I} \mathcal{O} \mathcal{N} \mathcal{E} \mathcal{N} \mathcal{N} \mathcal{E}] \mathcal{F}]]}$ = $\mathcal{[[\mathcal{G} \mathcal{P} \mathcal{N} \mathcal{I} \mathcal{O} \mathcal{N} \mathcal{E} \mathcal{N} \mathcal{N} \mathcal{E}] \mathcal{F}]]}$

Given a mapping $\mu$ and expression $\mathcal{E}$, the evauluation of $\mu \models \mathcal{E}$ is done according to [7].

4.2 Syntax and Compositional Semantics of Relational Algebra
The close relationship between SQL and Relational Algebra is the basis for query optimization in a RDBMS 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
$E \in \mathcal{E}$
\[
E \rightarrow \sigma_{\mathcal{C}on\mathcal{d}}E \mid \prod_{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n} E | E_1 \cup E_2 | E_1 \mathcal{X} E_2 | E_1 \times E_2 | E_1 - E_2 | E_1 \cap E_2
\]

Semantic Algebra for the domain of relations
Domain Relation: $R = \{t|\xi(t) \equiv \xi(R)\}$ ; Set of all tuples such that $t$ and $R$ are defined over the same schema.
1. $\sigma_{\mathcal{C}on\mathcal{d}}R = \{t|R(t) \land \mathcal{C}on\mathcal{d}(t)\}$ ; where $\mathcal{C}on\mathcal{d}(t)$ implies $\mathcal{C}on\mathcal{d}$ is satisfied by $t$
2. $\prod_{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n} R = \{t.A_1, t.A_2, \ldots, t.A_n|\mathcal{R}(t)\}$
3. $R_1 \cup R_2 = \{t|\mathcal{R}_1(t) \lor \mathcal{R}_2(t)\}$
4. $R_1 \mathcal{X} R_2 = \{t|\mathcal{R}_1(t_1) \land \mathcal{R}_2(t_2) \rightarrow t.A_1 = t_1.A_1 \land t.A_2 = t_1.A_2 \land \ldots \land t.A_n = t_1.A_n \land t.B_1 = t_2.B_1 \land \ldots \land t.B_n = t_2.B_n \land \mathcal{C}on\mathcal{d}(t)\}$
5. $R_1 \times R_2 = \{t|\forall t_1, \forall t_2 (R_1(t_1) \land R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \land t.A_2 = t_1.A_2 \land \ldots \land t.A_n = t_1.A_n \land t.B_1 = t_2.B_1 \land \ldots \land t.B_n = t_2.B_n)\}$
6. $R_1 - R_2 = \{t|R_1(t) \land \forall t_2 (R_2(t_2) \rightarrow t \neq t_2)\}$
7. $R_1 \cap R_2 = \{t|R_1(t) \land R_2(t)\}$

Valuation functions
\[
\mathcal{[[\mathcal{E}]]}, \mathcal{E} \rightarrow R; \quad \text{The valuation function } \mathcal{[[\mathcal{E}]]}, \mathcal{E} \rightarrow R; \quad \text{maps an element of } \mathcal{E} \text{ to } R
\]
- $\mathcal{[[\sigma_{\mathcal{C}on\mathcal{d}}E]]} = \sigma_{\mathcal{C}on\mathcal{d}}\mathcal{[[E]]}$
- $\prod_{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n} \mathcal{[[E]]} = \prod_{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n} \mathcal{[[E]]}$
- $\mathcal{[[E_1 \cup E_2]]} = \mathcal{[[E_1]]} \cup \mathcal{[[E_2]]}$
Intersection Projection

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
  \[ R_A: \sigma_{\text{Cond}} R = \{ t | R(t) \land \text{Cond}(t) \} \]
  \[ S_A: \text{Filter}_{\text{expr}} \Omega = \{ \mu | \mu \in \Omega \land \text{expr}(\mu) \} \]
  If \( \Omega \equiv [tp] \land \text{Cond} \equiv \text{expr} \land R \in S \) such that \( tp = ?s \land R ?o \) then \( R_A \equiv S_A \).

- Projection
  \[ R_A: \Pi_{A_1, A_2, \ldots, A_n} R = \{ t.A_1, t.A_2, \ldots, t.A_n | R(t) \} \]
  \[ S_A: \text{Project}_{v_1, v_2, \ldots, v_n} \Omega = \{ \mu[v_1, v_2, \ldots, v_n] | \mu \in \Omega \} \]
  If \( \Omega \equiv [tp] \land A_1, A_2, \ldots, A_n = v_1, v_2, \ldots, v_n \) and \( R \in S \) such that \( tp = ?s \land R ?o \) then \( R_A \equiv S_A \).

- Cross product
  \[ R_A: R_1 \times R_2 = \{ t | \forall t_1, t_2 (R_1(t_1) \land R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \land t.A_2 = t_1.A_2 \land t.B_1 = t_2.B_1 \land t.B_2 = t_2.B_2) \} \]
  \[ S_A: \Omega_1 \times \Omega_2 = \{ \mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings} \} \]
  If \( \Omega_1 \equiv [tp_1] \land \Omega_2 \equiv [tp_2] \land R_1, R_2 \in S \) such that \( tp_1 = ?s_1 \land R_1 ?o_1, tp_2 = ?s_2 \land R_2 ?o_2 \) then \( R_A \equiv S_A \).

- Join
  \[ R_A: R_1 \bowtie \text{Cond} \bowtie R_2 = \{ t | \forall t_1, t_2 (R_1(t_1) \land R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \land t.A_2 = t_1.A_2 \land t.B_1 = t_2.B_1 \land t.B_2 = t_2.B_2 \land \text{Cond}(t)) \} \]
  \[ S_A: \Omega_1 \bowtie \Omega_2 = \{ \mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings} \} \]
  If \( \Omega_1 \equiv [tp_1] \land \Omega_2 \equiv [tp_2] \land \text{Cond} \equiv \text{expr} \land R_1, R_2 \in S \) such that \( tp_1 = ?s_1 \land R_1 ?o_1, tp_2 = ?s_2 \land R_2 ?o_2 \) then \( R_A \equiv S_A \land S.A. \)

- Union
  \[ R_A: R_1 \cup R_2 = \{ t | (R_1(t) \lor R_2(t)) \land (\xi(R_1) \equiv \xi(R_2)) \} \]
  \[ S_A: \Omega_1 \cup \Omega_2 = \{ \mu | \mu \in \Omega_1 \lor \mu \in \Omega_2 \} \]
  If \( \Omega_1 \equiv [gp_1] \land \Omega_2 \equiv [gp_2] \land R_1 \equiv [gp_1] \land R_2 \equiv [gp_2] \), then \( R_A \equiv S_A \).

- Intersection
  \[ R_A: R_1 \cap R_2 = \{ t | (R_1(t) \land R_2(t)) \land (\xi(R_1) \equiv \xi(R_2)) \} \]
  \[ S_A: \Omega_1 \cap \Omega_2 = \{ \mu | \mu \in \Omega_1 \land \mu \in \Omega_2 \} \]
  If \( \Omega_1 \equiv [gp_1] \land \Omega_2 \equiv [gp_2] \land R_1 \equiv [gp_1] \land R_2 \equiv [gp_2] \), then \( R_A \equiv S_A \).
Difference

RA: $R_1 \setminus R_2 = \{ t \mid \exists t_1(t) \land \forall t_2(t_2) \land t_2 \neq t \} \land (\xi(R_1) \equiv \xi(R_2))$

SA: $\Omega_1 \setminus \Omega_2 = \{ (\Omega_1 \land \Omega_2) \cup (\Omega_1 \land \Omega_2) \}$

SA: $\text{Filter}_{\text{expr}} \Omega = \{ \mu \mid \mu \in \Omega \land \text{expr}(\mu) \}$

If $\Omega_1 \equiv \left[ tp_1 \right], \quad \Omega_2 \equiv \left[ tp_2 \right], \quad R_1, R_2 \in S$ such that $tp_1 = ?s_1 R_1 ?o_1, \quad tp_2 = ?s_1 R_2 ?o_2, \quad \text{expr} \equiv \neg \text{bound}(?o_2)$, then RA $\equiv$ SA.

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

5 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