

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
www.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 the past several decades, available to the RDF stores. In order to provide interoperability between these two 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 [5]. 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

^{*} This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-08-1-0260. We thank Dr. Herklotz (AFOSR) for his support.

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 [10], 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 [2, 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 knowledgebases 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 is usually 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 [3] 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 data management 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 [1] 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., the 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 [12] in 2006 and was followed and adapted by the W3C standard specification of SPARQL [13] in 2008. In [12], the authors present algebraic syntax with compositional semantics of SPARQL. 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 seman-

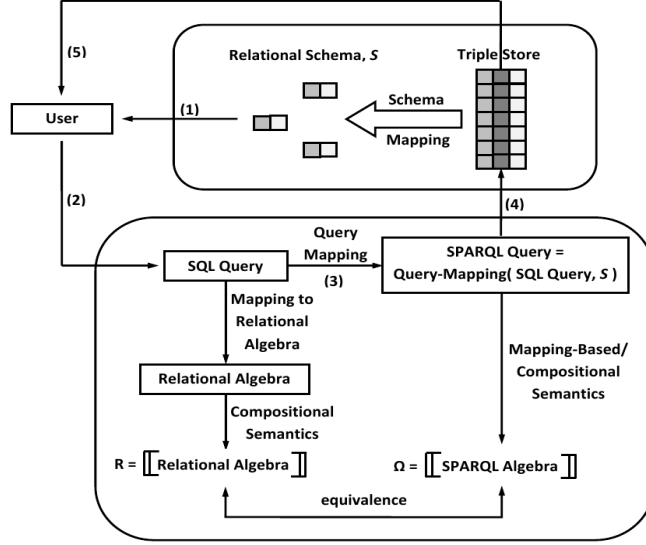


Fig. 1. Schema and Query Mapping

tics of SPARQL respectively. Since we reuse the compositional semantics from [12], the assumptions made in it also hold in this paper, which we briefly restate here. Algebraic formalization of 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 the 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 [6] 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 [12] and [5]. 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 lead 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

ER model and Description logics both model a given domain using classes and relationships. Calvanese et al., in [4] define a translation function that transforms ER model into a $\mathcal{ALN}\mathcal{I}$, DL knowledge base [4] 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, and this is the reason why usual ER to RDF mappings cannot be leveraged to do a reverse transformation in RETRO. In our framework, we derive a domain specific relational schema from the 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. The transformation or derivation of relational schema from RDF store using the schema mapping algorithm is trivially an information preserving translation.

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 Algorithm 1 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 predicate p , can be constructed by the evaluation of a triple pattern, tp , such that $tp.s = ?s_i \wedge tp.p \in P \wedge tp.o = ?o_i$, over a 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 . Algorithm 1 also ranks the predicates in the order in which they are discovered. The significance of ranking will become evident in the next section.

We now give an example of a RDF dataset D , which is an extension of the dataset from [12], and its corresponding relational schema S generated by

Algorithm 1. We also give an example of an instance of relation “name” that can be constructed from D . 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 a schema is left as a part of future work.

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

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

Table 1. The RDF dataset D

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”

We would now like to delve into how RETRO as a tool caters to knowledge evolution and ontology dynamics. Knowledge representation is an important aspect of knowledge. As mentioned earlier, over the past several decades application requirements have evolved and commensurately the ways to represent knowledge have also evolved. However, the gap between the users of the two knowledge representation methods, namely relational and RDF representation, still remains. Often, an analogy is established between databases and DL knowledgebases by comparing the schema of a database to a TBox and a database instance to an ABox. This analogy acts as a stepping stone towards the process of bridging the gap between them. RETRO supports evolution of knowledge representation by bridging the gap between the two knowledge representation paradigms through schema mapping and query mapping. RETRO will additionally facilitate application of existing database technologies for e.g., data analysis

tools used in stream databases [11], to RDF data. This will enable one to capture and analyze the dynamics of RDF data sets using the aforementioned tools, which are much more advanced than the current Semantic Web state of the art.

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 an algorithm that translates a given SQL query into a semantically equivalent SPARQL query. In our examples we use the RDF dataset D and its associated relational schema S that were defined in the previous section.

In Table 3, we give a description of an SQL query on schema S , its equivalent SPARQL query over dataset D obtained from Algorithm 2 and subsequently, we show the result of evaluating the SPARQL query on D . The SQL queries use the following relational algebra operators: Join, Union and Difference. Additional example queries for other operators can be found in our technical report [14].

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 2) 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 [14].

```

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        --> Attribute Op Attribute | Attribute Op Value.
Value            --> Numeral | String.
Op               --> '<' | '>' | '=' | '>=' | '<=' | '<>'.
Lop              --> 'AND' | 'OR' | 'NOT' .

```

| Description | SQL Query | SPARQL Query | ResultSet | | | | |
|---|--|--|---|-----|----------------|------|-------|
| Find the email address of john | <pre>SELECT email.o FROM name, email WHERE name.s = email.s AND name.o = john</pre> | <pre>SELECT ?o2 WHERE { ?s1 name ?o1 . ?s1 email ?o2 . FILTER (?o1 = john) }</pre> | <table border="1"> <tr><td>?o2</td></tr> <tr><td>john@acd.edu</td></tr> </table> | ?o2 | john@acd.edu | | |
| ?o2 | | | | | | | |
| john@acd.edu | | | | | | | |
| Names of all people who either have a phone number or an email address | <pre>SELECT name.o FROM name, email WHERE name.s = email.s UNION SELECT name.o FROM name, phone WHERE name.s = phone.s</pre> | <pre>SELECT ?o1 WHERE { { ?s1 name ?o1 . ?s1 email ?o2 . } UNION { ?s1 name ?o1 . ?s1 phone ?o3 . } }</pre> | <table border="1"> <tr><td>?o1</td></tr> <tr><td>paul</td></tr> <tr><td>john</td></tr> <tr><td>ringo</td></tr> </table> | ?o1 | paul | john | ringo |
| ?o1 | | | | | | | |
| paul | | | | | | | |
| john | | | | | | | |
| ringo | | | | | | | |
| Find the resources that work for the Research department but not the Admin department | <pre>SELECT dept.s FROM dept WHERE dept.o = Research EXCEPT SELECT dept.s FROM dept WHERE dept.o = Admin</pre> | <pre>SELECT ?s1 WHERE { { ?s1 dept ?o1 . FILTER (?o1 = Research) } OPTIONAL { ?s1 dept ?o2 . FILTER (?o2 = Admin) } FILTER (!bound(?o2)) }</pre> | <table border="1"> <tr><td>?s1</td></tr> <tr><td>B₂</td></tr> </table> | ?s1 | B ₂ | | |
| ?s1 | | | | | | | |
| B ₂ | | | | | | | |

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

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. Algorithm 2 is used to construct a SPARQL query and is based on the conceptual query evaluation strategy of SQL. It takes as input an SQL query string and a map P generated from Algorithm 1, and returns an equivalent SPARQL query string. The SQL query string is first used to generate a parse tree using function `parse(SQLQuery)`. We then extract SQL SELECT, FROM, WHERE-JC (Join Conditions of the form: *Attr Op Attr*) and WHERE-

BE (Boolean Expressions of the form: *Attr Op Value*) clause strings from the parse tree. The procedure then performs a query translation based on the type of a query. Algorithm 2 makes use of several sub-procedures to construct a SPARQL query. We briefly explain each of these sub-procedures below. An interested reader can find detailed explanations of these sub-procedures in our technical report [14].

Algorithm 2 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:  $tree = \text{parse}(q_{in})$  { A parse tree obtained by parsing  $q_{in}$  }
3:  $q_{in}^{SELECT} = \text{tree.getSelectClause}(); q_{in}^{FROM} = \text{tree.getFromClause}();$ 
4:  $q_{in}^{WHERE-JC} = \text{tree.getJoinConditions}(); q_{in}^{WHERE-BE} = \text{tree.getBooleanExpr}();$ 
5:  $q_{out}^{SELECT} = "SELECT "$   $q_{out}^{WHERE} = "WHERE { "$ 
6:  $TP \leftarrow \emptyset$  { The map of triple patterns is initially empty }
7: if  $tree.type = \text{CrossProductQuery}$  then
8:    $TP = \text{TRANSQlFROMCLAUSe}(q_{in}^{FROM})$ 
9:    $q_{out}^{WHERE} += \text{TRANSQlWHERECLAUSe}(q_{in}^{WHERE-JC}, q_{in}^{WHERE-BE}, TP, P)$ 
10:   $q_{out}^{SELECT} += \text{TRANSQlSELECTCLAUSe}(q_{in}^{SELECT}, TP)$ 
11:   $q_{out} = q_{out}^{SELECT} + q_{out}^{WHERE} + " }$ 
12: else if  $tree.type = \text{SPJQuery}$  then
13:   $TP = \text{TRANSQlFROMCLAUSe}(q_{in}^{FROM})$ 
14:   $q_{out}^{WHERE} += \text{TRANSQlWHERECLAUSe}(q_{in}^{WHERE-JC}, q_{in}^{WHERE-BE}, TP, P)$ 
15:   $TP = \text{extractTP}(q_{out}^{WHERE})$ 
16:   $q_{out}^{SELECT} += \text{TRANSQlSELECTCLAUSe}(q_{in}^{SELECT}, TP)$ 
17:   $q_{out} = q_{out}^{SELECT} + q_{out}^{WHERE} + " }$ 
18: else if  $tree.type = \text{UnionQuery}$  then
19:   $q_1 = \text{tree.leftSubTree}(), q_2 = \text{tree.rightSubTree}()$ 
20:   $q_1^{out} = \text{QUERY-MAPPING}(q_1), q_2^{out} = \text{QUERY-MAPPING}(q_2)$ 
21:   $q_{out} = \text{MERGE}(q_1^{out}, q_2^{out}, "UNION")$ 
22: else if  $tree.type = \text{IntersectQuery}$  then
23:   $q_1 = \text{tree.leftSubTree}(), q_2 = \text{tree.rightSubTree}()$ 
24:   $q_1^{out} = \text{QUERY-MAPPING}(q_1), q_2^{out} = \text{QUERY-MAPPING}(q_2)$ 
25:   $q_{out} = \text{MERGE}(q_1^{out}, q_2^{out}, "INTERSECT")$ 
26: else if  $tree.type = \text{ExceptQuery}$  then
27:   $q_1 = \text{tree.leftSubTree}(), q_2 = \text{tree.rightSubTree}()$ 
28:   $q_1^{out} = \text{QUERY-MAPPING}(q_1), q_2^{out} = \text{QUERY-MAPPING}(q_2)$ 
29:   $q_{out} = \text{MERGE}(q_1^{out}, q_2^{out}, "EXCEPT")$ 
30: end if
31: return  $q_{out}$ 

```

TRANSQlFROMCLAUSe: This sub-procedure 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 . For each table name that is present in a FROM clause, the sub-procedure generates a triple pattern of the form, $?s_i r_i ?o_i$, where r_i denotes

a given table name. A generated triple pattern is then added to map TP indexed by its corresponding table name.

TRANS_SQL_WHERE_CLAUSE: This sub-procedure takes sets of join conditions JC and boolean expressions BE , a map TP from **TRANS_SQL_FROM_CLAUSE** and map P generated from Algorithm 1 as inputs and returns a SPARQL WHERE clause string. In this sub-procedure, we use the aforementioned ranking of relations to sort the set JC to ensure that a consistent join ordering is obtained. We present the following example to show the necessity of ranking relations: Consider relations $name$, age and $dept$ with ranks 1, 2 and 3 and a query containing only join conditions “ $name.s = age.s$ AND $dept.s = age.s$ ”. If we do not rank relations, we would get the SPARQL WHERE clause: $\{?s_1 name ?o_1 . ?s_3 age ?o_2 . ?s_3 dept ?o_3\}$. However, using a sorting procedure, the second join condition would become $age.s = dept.s$, since the rank of $dept$ (3) was originally $>$ the rank of age (2). The sorted join conditions now lead to the correct WHERE clause: $\{?s_1 name ?o_1 . ?s_1 age ?o_2 . ?s_1 dept ?o_3\}$. Thus, the ranking of relations used in join re-ordering ensures generation of a consistent SPARQL WHERE clause. Note that the process of ranking relations and sorting join conditions is merely an implementation detail and has little conceptual significance. The sub-procedure then uses the sorted join conditions to generate a part of the SPARQL WHERE clause string. This is followed by generating filter conditions using triple patterns in TP and boolean expressions present in a SQL query.

TRANS_SQL_SELECT_CLAUSE: This sub-procedure takes as input a set of SQL SELECT attributes, A , and map TP , and returns a SPARQL SELECT clause string. It generates the variables for SPARQL SELECT by iterating over the set 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 and the attribute name part of a_i is used to access the desired variable within this triple pattern.

MERGE: This sub-procedure is used to merge two generated SPARQL sub-query strings for SQL queries containing Union, Intersection or Difference operators. The sub-procedure first extracts the necessary components of both SPARQL sub-queries (SELECT clause, triple patterns and filter expressions in WHERE clause). Next, the two sub-queries are combined into a merged query using various operations depending on the operator in the original SQL query. These operations include concatenation of triple patterns of both sub-queries, variable renaming to ensure consistency and introduction of specific keywords.

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 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.
- 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. An Operation is defined in terms of its functionality and a description of its mapping.
 - Functionality: For a function f , its functionality is an expression from domain and co-domain denoted by $f : D_1 \times D_2 \times \dots \times D_n \rightarrow A$, which implies f needs an argument from domain D_1 and one argument from D_2, \dots , and one from D_n to produce an answer belonging to domain A .
 - Description: A description of an operation's mapping is generally given as an equational definition. However, a set, a graph, a table or a 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 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 is 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 and 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 pattern expressions.

$gp \in \text{graphpatterns}$, where gp represents the syntax domain of graph patterns
 $gp ::= tp \mid gp \text{ FILTER } expr \mid PROJECT_{v_1, v_2, \dots, v_n} gp \mid gp \text{ UNION } gp \mid gp \text{ AND } gp \mid gp \text{ OPT } gp$

Semantic Algebra for the domain of mapping sets

Semantic Domain: Mapping Set, Ω .

Operations defined over the semantic domain Ω are:

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:
 - $\sigma_{expr} \Omega = \{\mu \mid \mu \in \Omega \wedge \mu \models expr\}$
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 $\prod_{v_1, v_2, \dots, v_n} : \Omega \rightarrow \Omega$ and description by:
 - $\prod_{v_1, v_2, \dots, v_n} \Omega = \{\mu \mid \mu \in \Omega \mid \mu \in \Omega\}$
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:
 - $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
4. The Join operator's functionality is given by $\bowtie : \Omega \times \Omega \rightarrow \Omega$, description by:
 - $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$
5. Difference operator's functionality is $\setminus : \Omega \times \Omega \rightarrow \Omega$, description is:
 - $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \text{for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\}$
6. The Optional operator's functionality is $\bowtie : \Omega \times \Omega \rightarrow \Omega$, description is:
 - $\Omega_1 \bowtie \Omega_2 = \{(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}$

Valuation functions

$\llbracket \cdot \rrbracket : gp \rightarrow \Omega$; The valuation function $\llbracket \cdot \rrbracket$, 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.

- $\llbracket tp \rrbracket = \{\mu \mid \text{dom}(\mu) = \text{var}(tp) \wedge \mu(tp) \in G\}$
- $\llbracket gp \text{ FILTER } expr \rrbracket = \sigma_{expr} \llbracket gp \rrbracket$
- $\llbracket PROJECT_{v_1, v_2, \dots, v_n} gp \rrbracket = \prod_{v_1, v_2, \dots, v_n} \llbracket gp \rrbracket$
- $\llbracket gp_1 \text{ UNION } gp_2 \rrbracket = \llbracket gp_1 \rrbracket \cup \llbracket gp_2 \rrbracket$
- $\llbracket gp_1 \text{ AND } gp_2 \rrbracket = \llbracket gp_1 \rrbracket \bowtie \llbracket gp_2 \rrbracket$
- $\llbracket gp_1 \text{ OPT } gp_2 \rrbracket = \llbracket gp_1 \rrbracket \bowtie \llbracket gp_2 \rrbracket$

The meaning of expression $\llbracket gp_1 \text{ UNION } gp_2 \rrbracket$ is computed by first evaluating $\llbracket gp_1 \rrbracket$ to get Ω_1 and $\llbracket gp_2 \rrbracket$ 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 $expr$ ($\mu \models expr$), iff

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

4.2 Compositional Semantics of Relational Algebra

The close relationship between SQL and Relational Algebra is the basis for query optimization in a RDBMS [15], 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 [7] 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.

$$E \rightarrow \sigma_{Cond}E \mid \prod_{A_1, A_2, \dots, A_n} E \mid E_1 \cup E_2 \mid E_1 \bowtie_{Cond} E_2 \mid E_1 - E_2 \mid E_1 \cap E_2 \mid E_1 \times E_2$$

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 the same schema or set of attributes.

Operations defined over the semantic domain R are:

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:
 - $\sigma_{Cond}R = \{t \mid R(t) \wedge Cond(t)\}$ where $R(t)$ denotes tuple t belongs to relation R .
2. The Project operator's functionality is $\prod_{A_1, A_2, \dots, A_n} : R \rightarrow R$, description is:
 - $\prod_{A_1, A_2, \dots, A_n} R = \{t.A_1, t.A_2, \dots, t.A_n \mid R(t)\}$
3. Union operator's functionality is $\cup : R \times R \rightarrow R$ and description is:
 - $R_1 \cup R_2 = \{t \mid (R_1(t) \vee R_2(t)) \wedge (\xi(R_1) \equiv \xi(R_2))\}$
 - A 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. The Join operator's functionality is $\bowtie_{Cond} : R_1 \times R_2 \rightarrow R$ and description is:
 - $R_1 \bowtie_{Cond} R_2 = \{t \mid \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge \dots \wedge t.A_n = t_1.A_n \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2 \wedge \dots \wedge t.B_n = t_2.B_n \wedge Cond(t))\}$
5. Difference operator's functionality is $- : R \times R \rightarrow R$ and description is:
 - $R_1 - R_2 = \{t \mid R_1(t) \wedge \forall t_2 (R_2(t_2) \rightarrow t \neq t_2)\}$
6. Intersection operator's functionality is $\cap : R \times R \rightarrow R$ and description is:
 - $R_1 \cap R_2 = \{t \mid (R_1(t) \wedge R_2(t)) \wedge (\xi(R_1) \equiv \xi(R_2))\}$
7. Cross product operator's functionality is $\times : R \times R \rightarrow R$ and description is:
 - $R_1 \times R_2 = \{t \mid \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge \dots \wedge t.A_n = t_1.A_n \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2 \wedge \dots \wedge t.B_n = t_2.B_n)\}$
 - where A_1, A_2, \dots, A_n are attributes of relation R_1 and B_1, B_2, \dots, B_n are attributes of R_2 .

Valuation functions

$\llbracket \cdot \rrbracket_r: E \rightarrow R$; The valuation function $\llbracket \cdot \rrbracket_r$, maps an element of syntax domain E to an element of semantic domain R .

- $\llbracket \sigma_{Cond} E \rrbracket_r = \sigma_{Cond} \llbracket E \rrbracket_r$
- $\llbracket \prod_{A_1, A_2, \dots, A_n} E \rrbracket_r = \prod_{A_1, A_2, \dots, A_n} \llbracket E \rrbracket_r$
- $\llbracket E_1 \cup E_2 \rrbracket_r = \llbracket E_1 \rrbracket_r \cup \llbracket E_2 \rrbracket_r$
- $\llbracket E_1 \bowtie_{Cond} E_2 \rrbracket_r = \llbracket E_1 \rrbracket_r \bowtie_{Cond} \llbracket E_2 \rrbracket_r$
- $\llbracket E_1 - E_2 \rrbracket_r = \llbracket E_1 \rrbracket_r - \llbracket E_2 \rrbracket_r$
- $\llbracket E_1 \cap E_2 \rrbracket_r = \llbracket E_1 \rrbracket_r \cap \llbracket E_2 \rrbracket_r$
- $\llbracket E_1 \times E_2 \rrbracket_r = \llbracket E_1 \rrbracket_r \times \llbracket E_2 \rrbracket_r$

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 $\Omega \equiv \llbracket tp \rrbracket$, $Cond \equiv expr$ 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 $\Omega \equiv \llbracket tp \rrbracket$ and $R \in S$ such that $tp = ?s R ?o$ ensure the data equivalence and $RA \equiv SA$ stand for operator equivalence.
- **Projection** If $\Omega \equiv \llbracket tp \rrbracket$, $\{v_i, A_i \equiv v_i, 1 \leq i \leq n\}$ and $R \in S$ such that $tp = ?s R ?o$ then $RA \equiv SA$.
 $RA: \prod_{A_1, A_2, \dots, A_n} R = \{t.A_1, t.A_2, \dots, t.A_n | R(t)\}$
 $SA: PROJECT_{v_1, v_2, \dots, v_n} \Omega = \{\mu | v_1, v_2, \dots, v_n | \mu \in \Omega\}$
- **Union** If $\Omega_1 \equiv \llbracket gp_1 \rrbracket$, $\Omega_2 \equiv \llbracket gp_2 \rrbracket$, $gp_1 = tp_1 AND tp_2$, $gp_2 = tp_1 AND tp_3$, and $R_1, R_2, R_3 \in S$ such that $tp_1 = ?s_1 R_1 ?o_1$, $tp_2 = ?s_1 R_2 ?o_2$, $tp_3 = ?s_1 R_3 ?o_3$ 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 $RA \equiv SA$.
 $RA: R_x \cup R_y = \{t | (R_x(t) \vee R_y(t)) \wedge (\xi(R_x) \equiv \xi(R_y))\}$
 $SA: \prod_{?s_1 ?o_1} (\Omega_1 \cup \Omega_2) = \{\mu | s_1, o_1 | \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
- **Join** If $\Omega_1 \equiv \llbracket tp_1 \rrbracket$, $\Omega_2 \equiv \llbracket tp_2 \rrbracket$, $Cond \equiv expr$, and $R_1, R_2 \in S$ such that $tp_1 = ?s_1 R_1 ?o_1$, $tp_2 = ?s_1 R_2 ?o_2$ then $RA \equiv SA$.
 $RA: R_1 \bowtie_{Cond} R_2 = \{t | \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2 \wedge Cond(t))\}$
 $SA1: \Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$
 $SA2: FILTER_{expr} \Omega = \{\mu | \mu \in \Omega \wedge expr(\mu)\}$
 $SA3: FILTER_{expr} (\Omega_1 \bowtie \Omega_2)$
 Semantics of $SA3$ is composed of semantics of $SA1$ and $SA2$.

- **Difference** If $\Omega_1 \equiv \llbracket gp_1 \rrbracket$, $\Omega_2 \equiv \llbracket gp_2 \rrbracket$, $R \in S$ such that $tp_1 = ?s_1 R ?o_1$, $tp_2 = ?s_1 R ?o_2$, $gp_1 = FILTER_{?o_1 Op} \text{“value1”}(tp_1)$, $gp_2 = FILTER_{?o_1 Op} \text{“value2”}(tp_2)$, $R_1 = \prod_{R.s} (\sigma_{R.o Op} \text{“value1”}(R))$, $R_2 = \prod_{R.s} (\sigma_{R.o Op} \text{“value2”}(R))$, $expr \equiv \neg bound(?o_2)$, then $RA \equiv SA3$.
 $RA: R_1 - R_2 = \{t | R_1(t) \wedge \forall t_2 (R_2(t_2) \wedge t \neq t_2) \wedge (\xi(R_1) \equiv \xi(R_2))\}$
 $SA1: \Omega_1 \bowtie \Omega_2 = \{\Omega_1 \bowtie \Omega_2\} \cup \{\Omega_1 \setminus \Omega_2\}$
 $SA2: FILTER_{expr} \Omega = \{\mu | \mu \in \Omega \wedge expr(\mu)\}$
 $SA3: \prod_{?s_1} (FILTER_{expr} (\Omega_1 \bowtie \Omega_2))$
 Semantics of $SA3$ is composed of semantics of $SA1$ and $SA2$.
- **Intersection** If $\Omega_1 \equiv \llbracket gp_1 \rrbracket$, $\Omega_2 \equiv \llbracket gp_2 \rrbracket$, $gp_1 = tp_1 AND tp_2$, $gp_2 = tp_1 AND tp_3$, and $R_1, R_2, R_3 \in S$ such that $tp_1 = ?s_1 R_1 ?o_1$, $tp_2 = ?s_1 R_2 ?o_2$, $tp_3 = ?s_1 R_3 ?o_3$ 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 $RA \equiv SA$.
 $RA: R_x \cap R_y = \{t | (R_x(t) \wedge R_y(t)) \wedge (\xi(R_x) \equiv \xi(R_y))\}$
 $SA: \prod_{?s_1, ?o_1} (\Omega_1 \bowtie \Omega_2) = \{(\mu_1 \cup \mu_2)_{|s_1, o_1} | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$
- **Cross product** If $\Omega_1 \equiv \llbracket tp_1 \rrbracket$, $\Omega_2 \equiv \llbracket tp_2 \rrbracket$ and $R_1, R_2 \in S$ such that $tp_1 = ?s_1 R_1 ?o_1$, $tp_2 = ?s_2 R_2 ?o_2$ then $RA \equiv SA$.
 $RA: R_1 \times R_2 = \{t | \forall t_1, \forall t_2 (R_1(t_1) \wedge R_2(t_2)) \rightarrow t.A_1 = t_1.A_1 \wedge t.A_2 = t_1.A_2 \wedge t.B_1 = t_2.B_1 \wedge t.B_2 = t_2.B_2\}$
 $SA: \Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 | \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}$

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 , $\llbracket RA \rrbracket_r \equiv \llbracket sparql \rrbracket$. Further explanation of the proof can be found in [14].

5 Related work

There have been several attempts to make RDBMS and RDF stores interoperate. The most popular one, D2RQ [3], has explored one direction i.e they construct an RDF schema from Relational data. A similar effort, W3C RDB2RDF WG [9], aims to generate RDF triples from one or more Relational tables without loss of information. In [8], the author describes a transformation from SPARQL to relational algebra and attempts to make existing work on query planning and optimization available to SPARQL implementors. RETRO provides the translation in the reverse direction and proves the translation to be semantics preserving by reusing the work done on compositional semantics of SPARQL in [12] and [5]. Compositional semantics of SPARQL discussed in [12] is reused and extended by authors in [5] by providing additional operators such as $FILTER_{expr}$ and $SELECT(v_1, v_2..v_n)$. However, unlike [12] and [5], we explicitly define compositional semantics in terms of valuation functions, which map the elements of syntax domain to elements of the 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 a 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 future work is to leverage the query optimization available for SQL queries to generate efficient, equivalent SPARQL queries.

References

1. Abadi, D.J., 0002, A.M., Madden, S., Hollenbach, K.J.: Scalable Semantic Web Data Management Using Vertical Partitioning. In: VLDB. pp. 411–422 (2007)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
3. Bizer, C., Cyganiak, R.: D2RQ Lessons Learned. <http://www.w3.org/2007/03/RdfRDB/papers/d2rq-positionpaper/>
4. Calvanese, D., Lenzerini, M., Nardi, D.: Unifying Class-Based Representation Formalisms. *J. Artif. Intell. Res. (JAIR)* 11, 199–240 (1999)
5. Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving SPARQL-to-SQL translation. *Data Knowl. Eng.* 68(10), 973–1000 (2009)
6. Codd, E.F.: 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)
7. Codd, E.F.: 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)
8. Cyganiak, R.: A relational algebra for SPARQL (2005), <http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html>
9. Malhotra, A.: W3C RDB2RDF Incubator Group Report. <http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/> (2009)
10. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. *J. Web Sem.* 7(2), 74–89 (2009)
11. Parker, D.S.: Stream data analysis in Prolog, pp. 249–312. MIT Press, Cambridge, MA, USA (1990), <http://dl.acm.org/citation.cfm?id=103477.103548>
12. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. In: International Semantic Web Conference. pp. 30–43 (2006)
13. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008)
14. Rachapalli, J., Khadilkar, V., Kantarcioglu, M., Thuraisingham, B.: RETRO: A Framework for Semantics Preserving SQL-to-SPARQL Translation. <http://www.utdallas.edu/~vvk072000/Research/Technical-Report/technical-report.html> (2011)
15. Ramakrishnan, R., Gehrke, J., Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill Science/Engineering/Math, 3 edn. (Aug 2002), <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0072465638>