

Semantics Preserving SPARQL-to-SQL Translation

(Technical Report TR-DB-112007-CLF, November 2007)

Artem Chebotko, Shiyong Lu, and Farshad Fotouhi
Department of Computer Science
Wayne State University
5143 Cass Avenue, Detroit, Michigan 48202, USA
{artem, shiyong, fotouhi}@wayne.edu

Abstract

Most existing RDF stores, which serve as metadata repositories on the Semantic Web, use an RDBMS as a backend to manage RDF data. This motivates us to study the problem of translating SPARQL queries into equivalent SQL queries, which further can be optimized and evaluated by the relational query engine and their results can be returned as SPARQL query solutions. The main contributions of our research are: (i) We formalize a relational algebra based semantics of SPARQL, which bridges an important gap between the Semantic Web and relational databases, and prove that our semantics is equivalent to the mapping-based semantics of SPARQL; (ii) Based on this semantics, we propose the first provably semantics preserving SPARQL-to-SQL translation for SPARQL triple patterns, basic graph patterns, optional graph patterns, union graph patterns, and value constraints; (iii) Our translation algorithm is generic and can be directly applied to existing RDBMS-based RDF stores; and (iv) We extend our defined semantics and translation to support the bag semantics of a SPARQL query solution and outline a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries.

1 Introduction

The Semantic Web [9] has recently gained tremendous momentum due to its great potential for providing a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. Semantic annotations for various heterogeneous resources on the Web are represented in Resource Description Framework (RDF) [2, 4], the standard language for annotating resources on the Web, and searched using the query language for RDF, called SPARQL [5], that has been proposed by the World Wide Web Consortium (W3C) and has recently achieved the candidate recommendation status. Essentially, RDF data is a collection of statements, called *triples*, of the form (s, p, o) , where s is called *subject*, p is called *predicate*, and o is called *object*, and each triple states the relation between a subject and an object. Such a collection of triples can be viewed as a directed graph, in which nodes represent subjects and objects, and edges represent predicates connecting from subject nodes to object nodes. To query RDF data, SPARQL allows the specification of triple and graph patterns to be matched over RDF graphs.

Explosive growth of RDF data on the Web drives the need for novel database systems, called *RDF stores*, that can efficiently store and query large RDF datasets. Most existing RDF stores, including Jena [43, 31], Sesame [12], 3store [25, 26], KAON [42], RStar [30], OpenLink Virtuoso [21], DLDB [33], RDFSuite [7, 41], DBOWL [32], PARKA [40], and ProvRDF [14], use a relational database management system (RDBMS) as a backend to manage RDF data. The main advantage of the RDBMS-based approach is that a mature and vigorous relational query engine with transactional processing support can be reused to provide major functionalities for RDF stores. The main challenge of this approach is that one needs to resolve the conflict between the graph RDF data model and the target relational data model. This usually requires various mappings, such as schema mapping, data mapping, and query mapping, to be performed between the two data models. One of the most difficult problems in this approach is the translation of SPARQL queries into

equivalent relational algebra expressions and SQL queries, which can be further optimized and evaluated by the relational query engine and their results can be returned as SPARQL query solutions.

A few early works on the SPARQL-to-SQL translation include [26], [18], and [15]. Harris and Shadbolt [26] shows how SPARQL basic graph pattern expressions, as well as simple optional graph patterns, can be translated into relational algebra expressions. Cyganiak [18] presents an insightful work on relational algebra for SPARQL and outlines rules establishing equivalence between this algebra and SQL. Our work [15] presents algorithms for the SPARQL basic and optional graph pattern translation into SQL and addresses some of the open problems discussed in [18]. All the three papers use the simplest relational database schema that essentially assumes one relation with three attributes to store triples of the form (s, p, o) . An influential work of Perez et al. [35, 34] presents the compositional mapping-based semantics of SPARQL, which is later adopted by the W3C SPARQL specification [5]. The new semantics, which greatly affects the semantics of optional graph patterns, requires existing SPARQL-to-SQL translation works to revisit their implementations.

In this paper, we present one of the most comprehensive research results on the SPARQL-to-SQL translation with the following main contributions:

1. We define a relational algebra based semantics of SPARQL as a function *eval*, which bridges an important gap between the Semantic Web and relational databases. We prove that *eval* is equivalent to the mapping-based semantics of SPARQL under interpretation function λ , which is used to establish the equivalence relationship between two SPARQL solution representations: a relational representation and a mapping-based representation.
2. We propose a SPARQL-to-SQL translation as a function *trans* for core SPARQL constructs and prove that *trans* is semantics preserving with respect to the relational algebra based semantics of SPARQL under interpretation function ϕ , which is used to establish the equivalence relationship between a relation produced by the relational algebra based SPARQL semantics *eval* and a relation produced by the evaluation of a *trans*-generated SQL query; *eval* and *trans* may produce relations with different relational attribute names due to the SQL naming constraints. *trans* supports the translation of SPARQL triple patterns, basic graph patterns, optional graph patterns, union graph patterns, and value constraints; *trans* is the first translation that fully supports union graph patterns.
3. The SPARQL-to-SQL translation *trans* is generic, i.e., supports both schema-oblivious and schema-aware database representations of existing RDBMS-based RDF stores, which is achieved by full separation of the translation from the relational database schema design. We checked that *trans* can be implemented in at least 11 existing RDF stores, including Jena, Sesame, 3store, KAON, RStar, OpenLink Virtuoso, DLDB, RDFSuite, DBOWL, PARKA, and ProvRDF.
4. Finally, we extend *eval* and *trans* to support the bag semantics of a SPARQL query solution and outline a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries.

The first two contributions are illustrated in Figure 1, where the dashed arrow represents the mapping-based semantics of SPARQL defined in [35], the dotted arrows represent this paper contribution to the definition of relational algebra based semantics of SPARQL, and the solid arrows represent this paper contribution to the definition of the SPARQL-to-SQL translation. The leftmost \Leftrightarrow arrow represents the equivalence between the two semantics definitions, and the rightmost \Leftrightarrow arrow represents that the translation is semantics preserving with respect to the relational algebra based semantics of SPARQL.

Organization. The rest of the paper is organized as follows. Section 2 discusses related work on storing and querying RDF data using an RDBMS-based RDF stores. Section 3 defines our relational algebra based semantics of SPARQL. Section 4 presents our semantics preserving SPARQL-to-SQL translation. Section 5 deals with the extension of the semantics and translation to support the bag semantics of a SPARQL query solution. Section 6 outlines our simplifications to the translation to generate simpler and more efficient SQL queries. Finally, Section 7 concludes the paper and discusses possible future work.

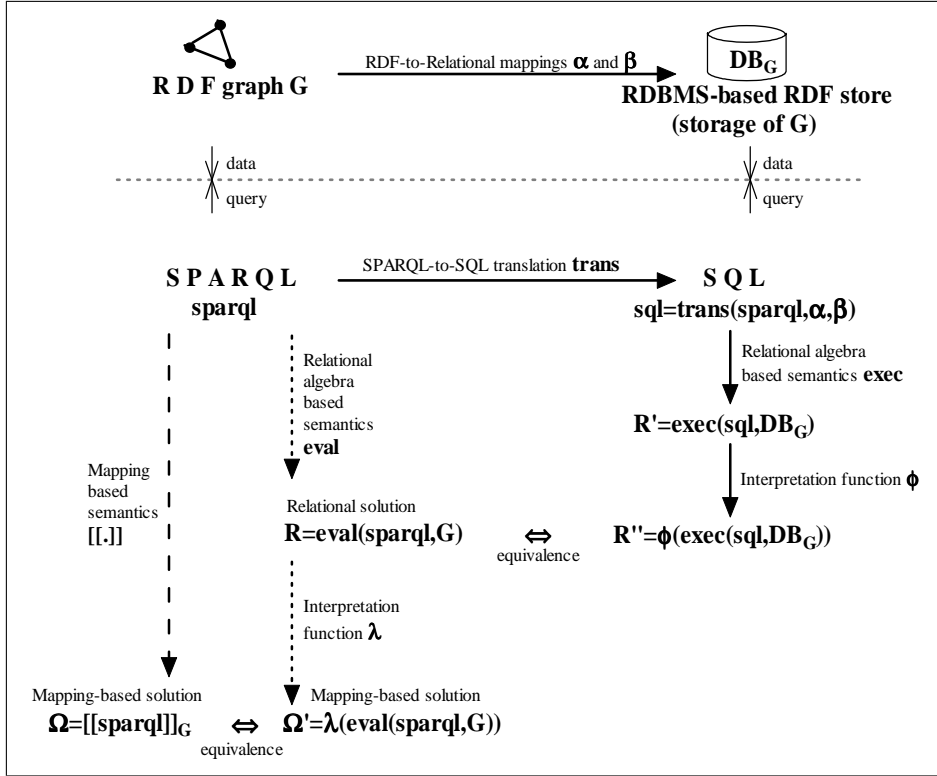


Figure 1: Our contributions in the definitions of the SPARQL semantics and SPARQL-to-SQL translation

2 Related Work

In recent years, a number of RDBMS-based RDF stores (see [8] for a survey) have been developed to support large-scale Semantic Web applications. To resolve the conflict between the graph RDF data model and the target relational data model, such systems require to deal with various mappings between the two data models, such as schema mapping, data mapping, and query mapping. First, the schema mapping is used to generate a relational database schema that can store RDF data. Second, the data mapping is used to shred RDF triples into relational tuples and insert them into the database. Finally, the query mapping is used to translate a SPARQL query into an equivalent SQL query, which is evaluated by the relational engine and its result is returned as a SPARQL query solution. In addition, RDF stores have to support inference of new RDF triples based on RDFS [3] or OWL [1] ontologies. In the following, we give more details on advances in RDF store design.

Database schemas employed by RDF stores fall into three categories [41]:

Schema-oblivious (also called *generic* or *vertical*): A single relation, e.g., $Triple(s,p,o)$, is used to store RDF triples, such that attribute s stores the subject of a triple, p stores its predicate, and o stores its object. Schema-oblivious RDF stores include Jena [43, 31], Sesame [12], 3store [25, 26], KAON [42], RStar [30], and OpenLink Virtuoso [21]. This approach has no concerns of RDF schema or ontology evolution, since it employs a generic database representation.

Schema-aware (also called *specific* or *binary*): This approach usually employs an RDF schema or ontology to generate so called *property relations* and *class relations*. A property relation, e.g., $Property(s,o)$, is created for each property in an ontology and stores subjects s and objects o related by this property. A class relation, e.g., $Class(i)$, is created for each class in an ontology and stores instances i of this class. In [14], along with property and class relations, *class-subject* and *class-object* relations are introduced. A class-subject relation, e.g., $ClassSubject(i,p,o)$ stores triples whose subjects are instances of a particular class in an ontology. Similarly, a class-object relation, e.g., $ClassObject(s,p,i)$, stores

triples whose objects are instances of a particular class. Such relations are useful for queries that retrieve all information about an instance (subject or object) of a particular class. Representatives of schema-aware RDF stores are DLDB [33], RDFSuite [7, 41], BDOWL [32], PARKA [40], and ProvRDF [14]. Another research work that falls into this category, but rather stands out, is presented in [20], where a database schema is generated based on patterns found in RDF data using data mining techniques. Schema evolution for this approach is quite straightforward (except for [20]): the addition or deletion of a class/property in an ontology requires the addition or deletion of a relation (or relational tuples) in the database. The schema-aware approach is in general yields better query performance than the schema-oblivious approach as has been shown in several experimental studies [6, 41, 7, 14].

Hybrid: This approach uses the mix of features of the previous two approaches. An example of the hybrid database schema is presented in [41], where a schema-oblivious database representation, e.g., $Triple(s, p, o)$, is partitioned into multiple relations based on the data type of object o , and a binary relation, e.g., $Class(i, c)$, is introduced to store instances i of classes c . [41] reports comparable query performance of the hybrid and schema-aware approaches.

Data mapping algorithms employed by existing RDF stores are usually fairly straightforward, such that RDF triples are inserted into a single relation as in the schema-oblivious approach, or into one or multiple relations as in the other approaches. Several data mapping strategies and algorithms are presented in [14].

Inference support mechanisms employed by RDF stores can be classified as *forward-chaining* or *backward-chaining*. In forward-chaining, all inferences are precomputed and stored along with explicit triples of an RDF graph. This enables fast query response and increased result completeness [22]; however, it complicates RDF data updates and consumes more storage space. The forward-chaining inference can be supported on the data mapping stage. In backward-chaining, inferences are computed dynamically for each query, which simplifies updates and omits a storage overhead, but results in worse query performance and scalability. This mechanism is bound by the main memory space required to compute inferences. The backward-chaining inference can be supported on the query mapping stage.

One of the most difficult mappings in RDBMS-based RDF stores is the query mapping. The literature on the SPARQL-to-SQL query translation, SPARQL query processing and optimization includes [26, 18, 15, 13, 14, 36, 39, 28, 27, 10, 16, 29]. Harris and Shadbolt [26] show how basic graph pattern expressions, as well as simple optional graph patterns, can be translated into relational algebra expressions over a schema-oblivious RDF store. Cyganiak [18] presents a relational algebra for SPARQL assuming a schema-oblivious RDF store and outlines rules establishing equivalence between this algebra and SQL. Our work [15] presents algorithms for basic and optional graph pattern translation into SQL for a schema-oblivious RDF store. To improve the evaluation performance of the SPARQL optional graph patterns in a relational database, our work [13] proposes a novel relational operator, called *nested optional join*, that shows better performance than conventional left outer join implementations. The W3C semantics of SPARQL [5] has changed since then, which was triggered by the compositional semantics presented by Perez et al. [35, 34]. The new semantics defines the same evaluation results for most SPARQL queries with so called “well-designed” patterns [35], but it is different from the previously used semantics for other queries. Therefore, research results on the SPARQL-to-SQL translation described above need to be revisited. Our more recent report [14] defines a SPARQL-to-SQL translation algorithm for basic graph pattern queries. The algorithm is schema-independent, i.e., it works for both schema-aware and schema-oblivious RDF stores, and is optimized, i.e., it selects smallest relations to query based on the type information of an instance and the statistics of the size of the relations in the database, as well as eliminates redundancies in basic graph patterns. Polleres [36] contributes with the translation of SPARQL queries into Datalog, along with other contributions on the extensions of SPARQL and its semantics. Serfiotis et al. [39] study the containment and minimization problems of RDF query fragments using a logic framework that allows to reduce these problems into their relational equivalents. Hartig and Heese [28] propose a SPARQL query graph model and pursue query rewriting based on this model. Harth and Decker [27] propose optimized index structures for RDF that can support efficient evaluation of select-project-join queries and can be implemented in a relational database. Bernstein et al. [10] propose SPARQL query optimization techniques based on triple pattern selectivity estimation and evaluate them using an in-memory SPARQL query engine. Chong et al. [16] introduce an SQL table function into the Oracle database to query RDF data, such that the function can be combined with SQL statements for further processing. Hung et al. [29] study the problem of RDF aggregate queries

by extending an RDF query language with the GROUP BY clause and several aggregate functions. Several research works [38, 37, 19, 11] focus on accessing conventional relational databases using SPARQL, which requires the SPARQL-to-SQL query translation. Finally, Guo et al. [24, 23] define requirements for Semantic Web knowledge base systems benchmarks and propose a framework for developing such benchmarks.

To our best knowledge, this paper presents one of the most comprehensive research results on the SPARQL-to-SQL translation, including the translation of SPARQL triple patterns, basic graph patterns, optional graph patterns, union graph patterns, and value constraints. It is important that we formalize the relational algebra based semantics of SPARQL which is equivalent to the mapping-based SPARQL semantics defined by Perez et al. [35] and W3C [5]. This allows us to define the first provably semantics preserving SPARQL-to-SQL translation algorithm. Moreover, our translation is generic, i.e., supports both schema-oblivious and schema-aware database representations, which was achieved by full separation of the translation from the relational database schema design.

3 Relational Algebra Based Semantics of SPARQL

In this section, we introduce our relational algebra based semantics of SPARQL. First, we formalize the core fragment of SPARQL over RDF without RDFS vocabulary and literal rules. Second, we give an overview of the mapping-based semantics [35] of SPARQL. Third, we identify challenges for formalizing a relational algebra based semantics of SPARQL. Fourth, we introduce a relational representation of a SPARQL query solution and establish its equivalence relationship to the mapping-based representation via an interpretation function. Fifth, we define the relational algebra based semantics of SPARQL as a set of premise-conclusion rules. Finally, we prove that our proposed semantics is equivalent to the mapping-based semantics under the stated interpretation.

3.1 Syntax of SPARQL and RDF

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

Definition 3.1 (RDF triple and RDF graph) An RDF triple t is a tuple $(s, p, o) \in (IB) \times I \times (IBL)$, where s , p , and o are a subject, predicate, and object, respectively. An RDF graph G is a set of RDF triples. \diamond

A sample RDF graph that we use for subsequent examples is shown in Figure 2. The RDF graph is represented as a set of 11 triples, as well as a labeled graph, in which edges are directed from subjects to objects and represent predicates, circles denote IRIs, and rectangles denote literals.

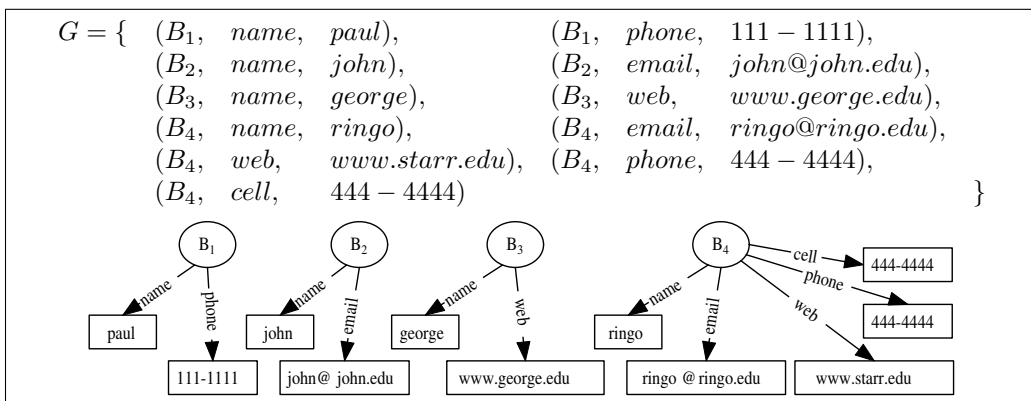


Figure 2: An RDF graph

Definition 3.2 (Triple pattern) A triple pattern tp is a triple $(sp, pp, op) \in (IVL) \times (IV) \times (IVL)$, where sp^1 , pp , and op are a subject pattern, predicate pattern, and object pattern, respectively. \diamond

For simplicity, we do not allow blank nodes in a triple pattern; such nodes can be considered as special kinds of variables, so called *non-distinguished variables*. We focus on the core fragment of SPARQL defined in the following.

Definition 3.3 (Graph pattern) A graph pattern gp is defined by the following abstract grammar:

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

where *AND*, *OPT*, and *UNION* are binary operators that correspond to SPARQL conjunction, OPTIONAL, and UNION constructs, respectively. *FILTER expr* represents the FILTER construct with a boolean expression $expr$, which is constructed using elements of the set IVL , constants, logical connectives (\neg , \vee , \wedge), inequality symbols ($<$, \leq , \geq , $>$), the equality symbol ($=$), unary predicates like *bound*, *isIRI*, and other features defined in [5]. We define function $var(gp)$ to return the set of variables that appear in gp . \diamond

Definition 3.4 (SPARQL query) A SPARQL query $sparql$ is defined as

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

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

3.2 An overview of the mapping-based semantics of SPARQL

In the following, we present a mapping-based representation of a SPARQL query solution and provide a brief overview of the mapping-based semantics of SPARQL defined in [35].

Definition 3.5 (Mapping-based representation of a SPARQL query solution) Let a mapping $\mu : V \rightarrow IBL$ be a partial function that assigns RDF terms of an RDF graph to variables of a SPARQL query. The domain of μ , $dom(\mu)$, is the subset of V over which μ is defined. The empty mapping μ_\emptyset is the mapping with empty domain. Then, the mapping-based representation of a SPARQL query solution is a set Ω of mappings μ . We define Σ as an infinite set of all possible mapping-sets, each of which represents a SPARQL query solution. \diamond

Example 3.6 (Mapping-based representation of a SPARQL query solution) Consider a graph pattern $(?a, email, ?e) \text{ OPT } (?a, web, ?w)$ that queries the RDF graph (see Figure 2) for an email $?e$ of a person $?a$ and, if available, for a web page $?w$ of $?a$, where $?a$, $?e$, and $?w$ are variables and *email* and *web* are URIs. The graph pattern solution is represented as follows.

$$\Omega = \begin{array}{l} \mu_1 : \\ \mu_2 : \end{array} \begin{array}{|l} \hline ?a \rightarrow B_2, \quad ?e \rightarrow john@john.edu \\ \hline ?a \rightarrow B_4, \quad ?e \rightarrow ringo@ringo.edu, \quad ?w \rightarrow www.starr.edu \\ \hline \end{array}$$

μ_1 is the result of successful match of the triple pattern $(?a, email, ?e)$ against triple $(B_2, email, john@john.edu)$. μ_2 is the result of successful match of the triple patterns $(?a, email, ?e)$ and $(?a, web, ?w)$ against triples $(B_4, email, ringo@ringo.edu)$ and $(B_4, web, www.starr.edu)$, respectively. \diamond

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 $\mu_1(x) = \mu_2(x)$; mappings with disjoint domains are always compatible; and μ_\emptyset is compatible with any other mapping. Let Ω_1 and Ω_2 be sets of mappings. In [35], the following operators (join, union, difference, and left outer join) are defined between Ω_1 and Ω_2 :

$$\begin{aligned} \Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\}, \\ \Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}, \end{aligned}$$

¹Note that a triple pattern can have a literal as a subject pattern, while an RDF triple cannot have a literal as a subject. This inconsistency between current RDF [4] and SPARQL [5] specifications does not affect our work and most likely will be resolved by W3C.

$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \text{for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\},$$

$$\Omega_1 \bowtie \Omega_2 = \{(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}.$$

The mapping-based semantics of SPARQL is defined as a function $[[\cdot]]_G$ which takes a graph pattern expression or a SPARQL query and an RDF graph G and returns a set of mappings. The definition of $[[\cdot]]$ is presented in Figure 3, where Rules 1–6 define the evaluation of triple pattern tp , gp_1 *AND* gp_2 , gp_1 *OPT* gp_2 , gp_1 *UNION* gp_2 , gp *FILTER* $expr$, and *SELECT* (v_1, v_2, \dots, v_n) *WHERE* (gp) , respectively, over an RDF graph G . Detailed description of $[[\cdot]]$ with illustrative examples is available in [35].

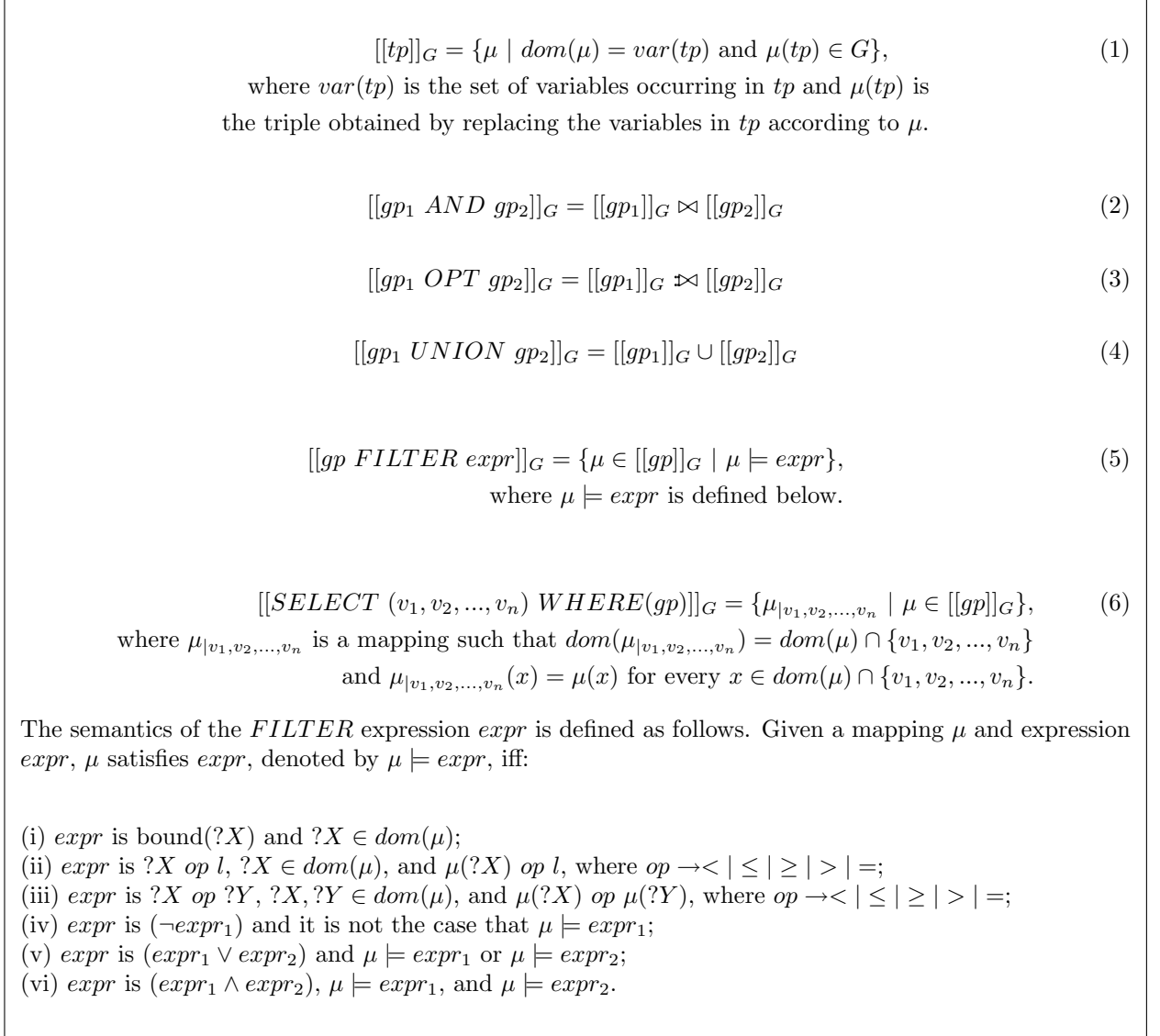


Figure 3: Mapping-based semantics of SPARQL

3.3 Challenges for formalizing a relational algebra based semantics of SPARQL

Although the mapping-based semantics of SPARQL defines precise and concise SPARQL query evaluation mechanism, it does not support SPARQL-to-SQL translation directly. To ensure semantics preserving SPARQL-to-SQL translation, an equivalent relational algebra based semantics is needed. However, formalizing such a semantics is challenging because:

1. While the mapping-based semantics works under the *abstract* form of partial functions, a relational algebra based semantics has to work under the *concrete* form of total functions, where each relational tuple is interpreted as a total function. A relational representation of a SPARQL query solution is required.
2. The notion of empty mapping, a mapping with empty domain, cannot be directly modeled using the relational algebra, because a tuple is defined on a non-empty set of relational attributes. Empty mappings occur when graph patterns have no variables, and therefore, a relational solution cannot store only variable bindings for such graph patterns.
3. One variable may occur in a triple pattern multiple times at various positions (subject, predicate, and object) simultaneously. A generic algorithm is needed to generate a select condition to ensure that multiple occurrences of the same variable are bounded to the same value. In addition, a generic algorithm is needed to generate a projection list to eliminate arbitrary duplicate relational attributes, which are disallowed in the relational model.
4. To encode the semantics of gp_1 AND gp_2 , we need to consider that one variable might occur multiple times within gp_1 and gp_2 and also across each other. These variables might be in a different binding status: unbound or bound to different or the same values. While the mapping-based semantics defines an abstract notion of “compatible mappings”, encoding such a notion in our concrete relational model is a very challenging task due to the difference between underlying solution representations and the difference between the mapping-based operators and relational algebra operators. In addition, inner or outer join resulting relations may have redundant attributes that must be eliminated. The same challenge also exists for formulating the semantics of gp_1 OPT gp_2 .
5. In contrast to the mapping-based union operator, the relational union requires its operands to be union-compatible, which frequently may not be the case. Therefore, the mapping-based union cannot be simply substituted by the relational union.
6. Finally, while evaluating *FILTER* expressions, the mapping-based semantics relies on mapping domains to detect unbound variables; however, a relational algebra based semantics cannot assume that unbound variables are not represented in a relational schema. A different mechanism is needed to deal with this situation.

3.4 Relational algebra based semantics of SPARQL

In this section, we first present our relational representation of a SPARQL query solution. Second, we define an interpretation function λ to relate the relational and mapping-based representations. Finally, we define our relational algebra based semantics of SPARQL and prove its equivalence to the mapping-based semantics.

Definition 3.7 (Relational representation of a SPARQL query solution) Let a tuple $r : IVL \rightarrow IBL \cup \{\text{NULL}\}$ be a total function, that assigns RDF terms of an RDF graph to URIs, literals, and variables of a SPARQL query, i.e., a URI or a literal is mapped to itself or to NULL, and a variable is mapped to an element of set $IBL \cup \{\text{NULL}\}$, where NULL denotes an undefined or unbound value. Then, the relational representation of a SPARQL query solution is a set R of tuples r or simply a relation R . The schema of R , denoted as $\xi(R)$, is the subset of IVL over which each tuple $r \in R$ is defined; abusing the notation, we denote a tuple schema as $\xi(r)$ and $\xi(r) \equiv \xi(R)$ for all $r \in R$. We define \mathcal{R} as an infinite set of all possible relations, each of which represents a SPARQL query solution. \diamond

Example 3.8 (Relational representation of a SPARQL query solution) Following the previous example, consider the same graph pattern $(?a, email, ?e)$ OPT $(?a, web, ?w)$. Its solution over the RDF graph (see Figure 2) is represented as follows.

$$R = \begin{array}{l} \xi(R) : \\ r_1 : \\ r_2 : \end{array} \begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@john.edu & NULL & NULL \\ \hline B_4 & email & ringo@ringo.edu & web & www.starr.edu \\ \hline \end{array}$$

r_1 is the result of successful match of the triple pattern $(?a, email, ?e)$ against triple $(B_2, email, john@john.edu)$.
 r_2 is the result of successful match of the triple patterns $(?a, email, ?e)$ and $(?a, web, ?w)$ against triples $(B_4, email, ringo@ringo.edu)$ and $(B_4, web, www.starr.edu)$, respectively. \diamond

To relate the relational representation and the mapping-based representation, we define an interpretation function λ as follows.

Definition 3.9 (Interpretation function λ) We define interpretation function $\lambda : \mathcal{R} \rightarrow \Sigma$ as the function that takes a relation $R \in \mathcal{R}$ and returns a mapping-set $\Omega \in \Sigma$, such that each tuple $r \in R$ is assigned a mapping $\mu \in \Omega$ in the following way: if $x \in \xi(r)$, $x \in V$ and $r(x) \neq \text{NULL}$, then $x \in \text{dom}(\mu)$ and $\mu(x) = r(x)$. \diamond

The example below shows that interpretation function λ can serve as a tool to establish the equivalence relationship between SPARQL query solutions when different representations are used.

Example 3.10 (Interpretation function λ) Given the solution Ω from Example 3.6 and the solution R from Example 3.8, one can verify that $\lambda(R) \equiv \Omega$.

$$R = \begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@... & \text{NULL} & \text{NULL} \\ \hline B_4 & email & ringo@... & web & www.st... \\ \hline \end{array} \xrightarrow{\lambda} \Omega = \begin{array}{|c|c|c|} \hline ?a \rightarrow B_2, & ?e \rightarrow john@... & \\ \hline ?a \rightarrow B_4, & ?e \rightarrow ringo@..., & ?w \rightarrow www.st... \\ \hline \end{array}$$

\diamond

Before we define the relational algebra based semantics of SPARQL, we need to introduce the following notations: R, R_1, R_2 , and R_3 denote relations, $\xi(R)$ denotes the schema of a relation R , \bowtie denotes an inner join, $\bowtie\leftarrow$ denotes a left outer join, \uplus denotes an outerunion, $/$ denotes a set difference, and ρ, σ , and π denote renaming, selection, and projection operators of the relational algebra, respectively. In addition, we introduce a new relational operator \dagger and two auxiliary functions, $genCond$ and $genPR$, in the following.

Definition 3.11 (Relational operator \dagger) Given a relation R with schema $\xi(R)$, two distinct relational attributes $a, b \in \xi(R)$, and a relational attribute $c \notin \xi(R) \setminus \{a, b\}$, the relational operator $\dagger_{(a,b) \rightarrow c}(R)$ merges attributes a and b of relation R into one single attribute c in the following way: for each tuple $r \in R$, if $r(a) \neq \text{NULL}$ then $r(c) \leftarrow r(a)$, else $r(c) \leftarrow r(b)$. \diamond

We show that \dagger can be derived from existing relational operators.

Theorem 3.12 Relational operator \dagger can be derived from existing relational operators as follows:

$$\dagger_{(a,b) \rightarrow c}(R) = \rho_{a \rightarrow c} \pi_{\xi(R) \setminus \{b\}} (\sigma_{a \neq \text{NULL}}(R)) \cup \rho_{b \rightarrow c} \pi_{\xi(R) \setminus \{a\}} (\sigma_{a = \text{NULL}}(R)).$$

Proof: $\dagger_{(a,b) \rightarrow c}(R)$ is defined to return the union of (i) all the tuples in R that have non-NULL values for a ; attribute b is excluded from the returned tuples; attribute a is renamed into c , and (ii) all the tuples in R that have NULL values for a ; attribute a is excluded from the returned tuples; attribute b is renamed into c . This conforms to the \dagger definition (see Definition 3.11). \square

Example 3.13 (Relational operator \dagger) Consider the following evaluation of $\dagger_{(a,b) \rightarrow c}(R)$ based on Theorem 3.12.

$$\dagger_{(a,b) \rightarrow c} \left(\begin{array}{|c|c|c|} \hline a & b & x \\ \hline a_1 & b_1 & x_1 \\ \hline a_2 & \text{NULL} & x_2 \\ \hline \text{NULL} & b_3 & x_3 \\ \hline \text{NULL} & \text{NULL} & x_4 \\ \hline \end{array} \right) = \begin{array}{|c|c|} \hline c & x \\ \hline a_1 & x_1 \\ \hline a_2 & x_2 \\ \hline \end{array} \cup \begin{array}{|c|c|} \hline c & x \\ \hline b_3 & x_3 \\ \hline \text{NULL} & x_4 \\ \hline \end{array} = \begin{array}{|c|c|} \hline c & x \\ \hline a_1 & x_1 \\ \hline a_2 & x_2 \\ \hline b_3 & x_3 \\ \hline \text{NULL} & x_4 \\ \hline \end{array}$$

\diamond

Further, we extend the definition of the \dagger operator to multiple attribute pair merging.

Definition 3.14 (Extended relational operator \dagger) Given a relation R with schema $\xi(R)$, n distinct relational attribute pairs $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n) \in \xi(R)$ and n relational attributes $c_1, c_2, \dots, c_n, c_i \notin \xi(R)/\{a_i, b_i\}$ ($i = 1 \dots n$), the relational operator $\dagger_{(a_1, b_1) \rightarrow c_1, (a_2, b_2) \rightarrow c_2, \dots, (a_n, b_n) \rightarrow c_n}(R)$ is defined recursively as:

$$\dagger_{(a_1, b_1) \rightarrow c_1, (a_2, b_2) \rightarrow c_2, \dots, (a_n, b_n) \rightarrow c_n}(R) = \dagger_{(a_1, b_1) \rightarrow c_1}(\dagger_{(a_2, b_2) \rightarrow c_2, \dots, (a_n, b_n) \rightarrow c_n}(R)).$$

◇

The two auxiliary functions are defined in Figure 4. Given a triple pattern tp , function $genCond$ generates a boolean expression which is evaluated to *true* if and only if tp matches an RDF triple t . The boolean expression ensures that either $tp.sp$ is a variable and thus can match any RDF term or $tp.sp = t.s$; similar conditions are introduced for $tp.pp$ and $tp.op$. Also, if $tp.sp = tp.pp$, then for tp to match t , it must be true that $t.s = t.p$; similarly for the cases when $tp.sp = tp.op$ and $tp.op = tp.pp$.

Example 3.15 (Function $genCond$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, email, ?a)$, $genCond$ generates the following conditions for tp_1 and tp_2 to match t :

$$\begin{aligned} genCond(tp_1) &= (tp_1.sp \in V \vee tp_1.sp = t.s) \wedge (tp_1.pp \in V \vee tp_1.pp = t.p) \wedge (tp_1.op \in V \vee tp_1.op = t.o) \\ &= (?a \in V \vee ?a = t.s) \wedge (email \in V \vee email = t.p) \wedge (?e \in V \vee ?e = t.o) \\ &= (email = t.p) \end{aligned}$$

For triple $t = (B_2, email, john@john.edu)$, $genCond(tp_1)$ is evaluated to $(email = email) = true$ and therefore, tp_1 matches t .

$$\begin{aligned} genCond(tp_2) &= (tp_2.sp \in V \vee tp_2.sp = t.s) \wedge (tp_2.pp \in V \vee tp_2.pp = t.p) \wedge (tp_2.op \in V \vee tp_2.op = t.o) \\ &\quad \wedge (t.s = t.o) \\ &= (?a \in V \vee ?a = t.s) \wedge (email \in V \vee email = t.p) \wedge (?a \in V \vee ?a = t.o) \wedge (t.s = t.o) \\ &= (email = t.p) \wedge (t.s = t.o) \end{aligned}$$

For triple $t = (B_2, email, john@john.edu)$, $genCond(tp_2)$ is evaluated to $(email = email) \wedge (B_2 = john@john.edu) = false$ and therefore, tp_2 does not match t . ◇

Let relation R with schema $\xi(R) = (s, p, o)$ stores the subset of triples of G that match triple pattern tp . We define function $genPR$ that, given a triple pattern tp , generates a relational algebra expression which projects only those attributes of relation R that correspond to distinct $tp.sp$, $tp.pp$, and $tp.op$ and renames the projected attributes as $s \rightarrow tp.sp$, $p \rightarrow tp.pp$, and $o \rightarrow tp.op$. $R.s$ is always projected and renamed into $tp.sp$, $R.p$ is projected and renamed into $tp.pp$ if $tp.pp \neq tp.sp$, and $R.o$ is projected and renamed into $tp.op$ if $tp.op \neq tp.sp$ and $tp.op \neq tp.pp$. This projection procedure ensures that, after attribute renaming, the schema of the resulting relation does not have duplicate attribute names.

Example 3.16 (Function $genPR$) For the purpose of this example only, we extend the RDF graph G in Figure 2 with additional triple $(B_5, email, B_5)$. Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, email, ?a)$, $genPR$ generates the following relational algebra expressions:

$$\begin{aligned} genPR(tp_1) &= \rho_{s \rightarrow tp_1.sp, p \rightarrow tp_1.pp, o \rightarrow tp_1.op} \pi_{s,p,o}(R) = \rho_{s \rightarrow ?a, p \rightarrow email, o \rightarrow ?e} \pi_{s,p,o}(R) \\ genPR(tp_2) &= \rho_{s \rightarrow tp_2.sp, p \rightarrow tp_2.pp} \pi_{s,p}(R) = \rho_{s \rightarrow ?a, p \rightarrow email} \pi_{s,p}(R) \end{aligned}$$

Let relation R stores the subset of triples of G that match tp_1 (tp_2). The evaluation of the generated expressions for R is as follows:

$$\begin{aligned} R &= \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_2 & email & john@... \\ \hline B_4 & email & ringo@... \\ \hline \end{array} \xrightarrow{\text{evaluate } genPR(tp_1)} \begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@... \\ \hline B_4 & email & ringo@... \\ \hline \end{array} \\ R &= \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_5 & email & B_5 \\ \hline \end{array} \xrightarrow{\text{evaluate } genPR(tp_2)} \begin{array}{|c|c|} \hline ?a & email \\ \hline B_5 & email \\ \hline \end{array} \end{aligned}$$

◇

We define the relational algebra based semantics of SPARQL as a function $eval$ which takes a graph pattern expression or a SPARQL query and an RDF graph and returns a resulting relation. In Figure 5, $eval$ is defined as a set of premise-conclusion rules explained in the following.

```

01 Function genCond
02 Input: triple pattern tp
03 Output: boolean expression cond which is true iff tp matches an RDF triple t
04 Begin
05   cond = (tp.sp ∈ V ∨ tp.sp = t.s) ∧ (tp.pp ∈ V ∨ tp.pp = t.p) ∧ (tp.op ∈ V ∨ tp.op = t.o)
06   If tp.sp = tp.pp then cond += ∧(t.s = t.p) End If
07   If tp.sp = tp.op then cond += ∧(t.s = t.o) End If
08   If tp.op = tp.pp then cond += ∧(t.o = t.p) End If
09 Return cond
10 End Function

11 Function genPR
12 Input: triple pattern tp
13 Output: relational algebra expression which projects only those attributes of relation R
14 with schema  $\xi(R) = (s, p, o)$  that correspond to distinct tp.sp, tp.pp, and tp.op
15 and renames the projected attributes as s → tp.sp, p → tp.pp, o → tp.op
16 Begin
17   project-list = s
18   rename-list = s → tp.sp
19   If tp.pp ≠ tp.sp then project-list += p, rename-list = p → tp.pp End If
20   If tp.op ≠ tp.sp and tp.op ≠ tp.pp then project-list += o, rename-list = o → tp.op End If
21 Return  $\rho_{\text{rename-list}} \pi_{\text{project-list}}(R)$ 
22 End Function

```

Figure 4: Functions *genCond* and *genPR*

Rule 7 defines the evaluation of a triple pattern *tp* over *G* in two steps. First, the relation *R* with the fixed schema $\xi(R) = (s, p, o)$ is created and all the triples $t \in G$ that match *tp* based on the condition generated by *genCond(tp)* are stored into *R*. Then, attributes of *R* are projected and renamed based on the relational algebra expression generated by *genPR(tp)* and the new relation *R*₂ is created. Finally, *R*₂ is assigned as a solution to the triple pattern.

Example 3.17 (Rule 7: $eval(tp, G)$) The evaluation of the triple pattern $tp_1 = (?a, email, ?e)$ over the RDF graph *G* in Figure 2 is as follows.

$$\begin{aligned}
R = \{(t.s, t.p, t.o) \mid t \in G \wedge (email = t.p)\} &= \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array}, \\
R_2 = \rho_{s \rightarrow ?a, p \rightarrow email, o \rightarrow ?e} \pi_{s, p, o}(R) &= \begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array}, \\
eval(tp_1, G) &= R_2.
\end{aligned}$$

Similarly, the evaluation of the triple pattern $tp_2 = (?a, web, ?w)$ over the RDF graph *G* in Figure 2 is as follows.

$$\begin{aligned}
R = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_3 & web & www.george.edu \\ \hline B_4 & web & www.starr.edu \\ \hline \end{array}, \quad R_2 = \begin{array}{|c|c|c|} \hline ?a & web & ?w \\ \hline B_3 & web & www.george.edu \\ \hline B_4 & web & www.starr.edu \\ \hline \end{array}, \quad eval(tp_2, G) = R_2.
\end{aligned}$$

◇

Rule 8 defines the evaluation of the *AND* of two graph patterns *gp*₁ and *gp*₂ as the inner join of relations $R_1 = eval(gp_1, G)$ and $R_2 = eval(gp_2, G)$. The join condition ensures that for every pair of common relational attributes $(R_1.a_i, R_2.a_i)$ where $a_i \in \xi(R_1) \cap \xi(R_2)$, their values are equal $R_1.a_i = R_2.a_i$ or one or both values are NULLs. The † operator is used to merge redundant attributes of the join-resulting relation into one, such that out of each pair of attributes $(R_1.a_i, R_2.a_i)$, only one is projected and renamed into *a*_{*i*}. $R_1.a_i$ is projected for those tuples whose corresponding value is not NULL, otherwise $R_2.a_i$ is projected. Other attributes of R_1 and R_2 are projected per se.

Example 3.18 (Rule 8: $eval(gp_1 \text{ AND } gp_2, G)$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the evaluation of the graph pattern $(tp_1 \text{ AND } tp_2)$ over the RDF graph *G* in Figure 2 is as follows. Let $eval(tp_1, G) = R_1$ and $eval(tp_2, G) = R_2$, then $eval(tp_1 \text{ AND } tp_2, G) =$

$$\frac{R(s, p, o) = \{(t.s, t.p, t.o) \mid t \in G \wedge \text{genCond}(tp)\}, R_2 = \text{genPR}(tp)}{\text{eval}(tp, G) = R_2} \quad (7)$$

$$\frac{R_1 = \text{eval}(gp_1, G), R_2 = \text{eval}(gp_2, G),}{R_3 = \dagger_{[(R_1.a_i, R_2.a_i) \rightarrow a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1 \bowtie_{\wedge_{[a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1.a_i = R_2.a_i \vee R_1.a_i = \text{NULL} \vee R_2.a_i = \text{NULL})} R_2)}{\text{eval}(gp_1 \text{ AND } gp_2, G) = R_3} \quad (8)$$

$$\frac{R_1 = \text{eval}(gp_1, G), R_2 = \text{eval}(gp_2, G),}{R_3 = \dagger_{[(R_1.a_i, R_2.a_i) \rightarrow a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1 \bowtie_{\vee_{[a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1.a_i = R_2.a_i \vee R_1.a_i = \text{NULL} \vee R_2.a_i = \text{NULL})} R_2)}{\text{eval}(gp_1 \text{ OPT } gp_2, G) = R_3} \quad (9)$$

$$\frac{R_1 = \text{eval}(gp_1, G), R_2 = \text{eval}(gp_2, G), R_3 = R_1 \uplus R_2}{\text{eval}(gp_1 \text{ UNION } gp_2, G) = R_3} \quad (10)$$

$$\frac{R_1 = \text{eval}(gp, G), R_2 = \{r \mid r \in R_1 \wedge \text{expr}(r)\}}{\text{eval}(gp \text{ FILTER } \text{expr}, G) = R_2} \quad (11)$$

$$\frac{R = \text{eval}(gp, G)}{\text{eval}(\text{SELECT } (v_1, v_2, \dots, v_n) \text{ WHERE}(gp), G) = \pi_{v_1, v_2, \dots, v_n}(R)} \quad (12)$$

The semantics of the *FILTER* expression *expr* is defined as follows. Given a tuple *r* of a relation *R* and expression *expr*, *expr*(*r*) = *true*, iff:

- (i) *expr* is bound(?*X*), ?*X* ∈ ξ(*R*), and *r*(?*X*) ≠ NULL;
- (ii) *expr* is ?*X* *op* *l*, ?*X* ∈ ξ(*R*), *r*(?*X*) ≠ NULL, and *r*(?*X*) *op* *l*, where *op* → < | ≤ | ≥ | > | =;
- (iii) *expr* is ?*X* *op* ?*Y*, ?*X*, ?*Y* ∈ ξ(*R*), *r*(?*X*) ≠ NULL, *r*(?*Y*) ≠ NULL, and *r*(?*X*) *op* *r*(?*Y*);
- (iv) *expr* is (¬*expr*₁) and *expr*₁(*r*) = *false*;
- (v) *expr* is (*expr*₁ ∨ *expr*₂) and *expr*₁(*r*) = *true* or *expr*₂(*r*) = *true*;
- (vi) *expr* is (*expr*₁ ∧ *expr*₂), *expr*₁(*r*) = *true*, and *expr*₂(*r*) = *true*.

Figure 5: Relational algebra based semantics of SPARQL

$$\dagger_{(R_1.?a, R_2.?a) \rightarrow ?a} \left(\begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array} \bowtie_{(R_1.?a=R_2.?a \vee R_1.?a=\text{NULL} \vee R_2.?a=\text{NULL})} \begin{array}{|c|c|c|} \hline ?a & web & ?w \\ \hline B_3 & web & www.george.edu \\ \hline B_4 & web & www.starr.edu \\ \hline \end{array} \right) = \begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_4 & email & ringo@ringo.edu & web & www.starr.edu \\ \hline \end{array}$$

◇

Rule 9 defines the evaluation of the *OPT* of two graph patterns *gp*₁ and *gp*₂ as the left outer join of relations *R*₁ = *eval*(*gp*₁, *G*) and *R*₂ = *eval*(*gp*₂, *G*). The join condition and the use of the † operator are analogous to the previous rule. The only difference between the evaluations of *AND* and *OPT* operators is the use of inner join and left outer join, respectively.

Example 3.19 (Rule 9: *eval*(*gp*₁ *OPT* *gp*₂, *G*)) Given triple patterns *tp*₁ = (?*a*, *email*, ?*e*) and *tp*₂ = (?*a*, *web*, ?*w*), the evaluation of the graph pattern (*tp*₁ *OPT* *tp*₂) over the RDF graph *G* in Figure 2 is as follows. Let *eval*(*tp*₁, *G*) = *R*₁ and *eval*(*tp*₂, *G*) = *R*₂, then *eval*(*tp*₁ *OPT* *tp*₂, *G*) =

$$\begin{aligned} & \dagger_{(R_1.?a, R_2.?a) \rightarrow ?a} \left(\begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array} \bowtie_{\substack{(R_1.?a=R_2.?a \vee R_1.?a=NULL \vee \\ R_2.?a=NULL)}} \begin{array}{|c|c|c|} \hline ?a & web & ?w \\ \hline B_3 & web & www.george.edu \\ \hline B_4 & web & www.starr.edu \\ \hline \end{array} \right) = \\ & = \begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@john.edu & NULL & NULL \\ \hline B_4 & email & ringo@ringo.edu & web & www.starr.edu \\ \hline \end{array} \end{aligned}$$

◇

Rule 10 defines the evaluation of the *UNION* of two graph patterns gp_1 and gp_2 as the outerunion of relations $R_1 = eval(gp_1, G)$ and $R_2 = eval(gp_2, G)$. The outerunion NULL-pads the tuples of each relation to schema $\xi(R_1) \cup \xi(R_2)$ and computes the union of the resulting relations [17]. If R_1 and R_2 have identical schemas, i.e., $\xi(R_1) \equiv \xi(R_2)$, then the outerunion is equivalent to the relational union, i.e., $R_1 \uplus R_2 \equiv R_1 \cup R_2$.

Example 3.20 (Rule 10: $eval(gp_1 \text{ UNION } gp_2, G)$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the evaluation of the graph pattern $(tp_1 \text{ UNION } tp_2)$ over the RDF graph G in Figure 2 is as follows. Let $eval(tp_1, G) = R_1$ and $eval(tp_2, G) = R_2$, then $eval(tp_1 \text{ UNION } tp_2, G) =$

$$\begin{aligned} & \begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array} \uplus \begin{array}{|c|c|c|} \hline ?a & web & ?w \\ \hline B_3 & web & www.george.edu \\ \hline B_4 & web & www.starr.edu \\ \hline \end{array} = \\ & = \begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@john.edu & NULL & NULL \\ \hline B_4 & email & ringo@ringo.edu & NULL & NULL \\ \hline B_3 & NULL & NULL & web & www.george.edu \\ \hline B_4 & NULL & NULL & web & www.starr.edu \\ \hline \end{array} \end{aligned}$$

Similarly, the evaluation of the graph pattern $(tp_1 \text{ UNION } tp_1)$ over the RDF graph G in Figure 2 is shown below. Let $eval(tp_1, G) = R_1$, then $eval(tp_1 \text{ UNION } tp_1, G) =$

$$\begin{aligned} & \begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array} \uplus \begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline ?a & email & ?e \\ \hline B_2 & email & john@john.edu \\ \hline B_4 & email & ringo@ringo.edu \\ \hline \end{array} \end{aligned}$$

◇

Rule 11 defines the evaluation of the *FILTER* expression $expr$ for graph pattern gp as the subset of tuples r of relation $R_1 = eval(gp, G)$, for which the condition $expr(r)$ is true. The semantics of $expr(r)$ is elaborated in Figure 5.

Example 3.21 (Rule 11: $eval(gp \text{ FILTER } expr, G)$) Given the graph pattern $gp = (?a, email, ?e) \text{ OPT } (?a, web, ?w)$ and the boolean expression $expr = \neg bound(?w)$, the evaluation of the graph pattern $gp \text{ FILTER } expr$ over the RDF graph G in Figure 2 is as follows. Let $eval(gp, G) = R$ (see Example 3.19), then $eval(gp \text{ FILTER } expr, G) =$

$$\begin{aligned} & \left\{ r \mid r \in \left(\begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@john.edu & NULL & NULL \\ \hline B_4 & email & ringo@ringo.edu & web & www.starr.edu \\ \hline \end{array} \right) \wedge \neg(r(?w) \neq NULL) \right\} = \\ & = \begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@john.edu & NULL & NULL \\ \hline \end{array} \end{aligned}$$

◇

Finally, Rule 12 defines the evaluation of a SPARQL query as the projection of specified variables v_1, v_2, \dots, v_n from the relation corresponding to the evaluation of the query graph pattern gp .

Example 3.22 (Rule 12: $eval(SELECT (v_1, v_2, \dots, v_n) WHERE(gp), G)$) Given the graph pattern $gp = (?a, email, ?e) OPT (?a, web, ?w)$ and the variable list $(?a, ?e, ?w)$, the evaluation of the SPARQL query $SELECT (?a, ?e, ?w) WHERE(gp)$ over the RDF graph G in Figure 2 is as follows. Let $eval(gp, G) = R$ (see Example 3.19), then $eval(SELECT (?a, ?e, ?w) WHERE(gp), G) =$

$$\pi_{?a, ?e, ?w} \left(\begin{array}{|c|c|c|c|c|} \hline ?a & email & ?e & web & ?w \\ \hline B_2 & email & john@john.edu & NULL & NULL \\ \hline B_4 & email & ringo@ringo.edu & web & www.starr.edu \\ \hline \end{array} \right) =$$

$$\begin{array}{|c|c|c|} \hline ?a & ?e & ?w \\ \hline B_2 & john@john.edu & NULL \\ \hline B_4 & ringo@ringo.edu & www.starr.edu \\ \hline \end{array}$$

◇

Additionally, we illustrate how $eval$ works on several more complex queries. To facilitate easy comparison of the relational algebra based semantics with the mapping-based semantics, we use similar RDF graph and queries as in [35]. For each SPARQL query Q_i below and the RDF graph G in Figure 2, one can verify that $\lambda(eval(Q_i, G)) \equiv [[Q_i]]_G$, where $[[\cdot]]$ is the mapping-based semantics of SPARQL defined in [35].

Example 3.23 (Evaluation of more complex SPARQL queries)

The following are sample SPARQL queries and their evaluations over the RDF graph G in Figure 2:

The first query includes two OPT operators that correspond to the case of so called *sequential* **OPTIONALS**.
 Q_1 : $SELECT ?a, ?n, ?e, ?w WHERE (((?a, name, ?n) OPT (?a, email, ?e)) OPT (?a, web, ?w))$.
 $R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$
 $R_2 = eval((?a, email, ?e), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$
 $R_3 = eval((?a, web, ?w), G) = \{(B_3, web, www.george.edu), (B_4, web, www.starr.edu)\}$
 $R_4 = eval((?a, name, ?n) OPT(?a, email, ?e), G) = \dagger_{(R_1.?a, R_2.?a) \rightarrow ?a} (R_1 \bowtie_{(R_1.?a=R_2.?a \vee R_1.?a=NULL \vee R_2.?a=NULL)} R_2) =$
 $\{(B_1, name, paul, NULL, NULL), (B_2, name, john, email, john@john.edu), (B_3, name, george, NULL, NULL),$
 $(B_4, name, ringo, email, ringo@ringo.edu)\}$
 $eval(Q_1, G) = \pi_{?a, ?n, ?e, ?w} (\dagger_{(R_4.?a, R_3.?a) \rightarrow ?a} (R_4 \bowtie_{(R_4.?a=R_3.?a \vee R_4.?a=NULL \vee R_3.?a=NULL)} R_3)) =$

?a	?n	?e	?w
B ₁	paul	NULL	NULL
B ₂	john	john@john.edu	NULL
B ₃	george	NULL	www.george.edu
B ₄	ringo	ringo@ringo.edu	www.starr.edu

The second query is similar to the first one, except that variables $?e$ and $?w$ are substituted by the same variable $?ew$.

Q_2 : $SELECT ?a, ?n, ?ew WHERE (((?a, name, ?n) OPT (?a, email, ?ew)) OPT (?a, web, ?ew))$.
 $R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$
 $R_2 = eval((?a, email, ?ew), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$
 $R_3 = eval((?a, web, ?ew), G) = \{(B_3, web, www.george.edu), (B_4, web, www.starr.edu)\}$
 $R_4 = eval((?a, name, ?n) OPT(?a, email, ?ew), G) = \dagger_{(R_1.?a, R_2.?a) \rightarrow ?a} (R_1 \bowtie_{(R_1.?a=R_2.?a \vee R_1.?a=NULL \vee R_2.?a=NULL)} R_2) =$
 $\{(B_1, name, paul, NULL, NULL), (B_2, name, john, email, john@john.edu), (B_3, name, george, NULL, NULL),$
 $(B_4, name, ringo, email, ringo@ringo.edu)\}$
 $eval(Q_2, G) = \pi_{?a, ?n, ?ew} (\dagger_{(R_4.?a, R_3.?a) \rightarrow ?a, (R_4.?ew, R_3.?ew) \rightarrow ?ew} (R_4$
 $\bowtie_{(R_4.?a=R_3.?a \vee R_4.?a=NULL \vee R_3.?a=NULL) \wedge (R_4.?ew=R_3.?ew \vee R_4.?ew=NULL \vee R_3.?ew=NULL)} R_3)) =$

?a	?n	?ew
B ₁	paul	NULL
B ₂	john	john@john.edu
B ₃	george	www.george.edu
B ₄	ringo	ringo@ringo.edu

The third query includes two *OPT* operators that correspond to the case of so called *nested* *OPTIONALS*.

Q_3 : SELECT ?a, ?n, ?e, ?w WHERE ((?a, name, ?n) OPT ((?a, email, ?e) OPT (?a, web, ?w))).

$R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$

$R_2 = eval((?a, email, ?e), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$

$R_3 = eval((?a, web, ?w), G) = \{(B_3, web, www.george.edu), (B_4, web, www.starr.edu)\}$

$R_4 = eval((?a, email, ?e) OPT(?a, web, ?w), G) = \dagger_{(R_2.?a, R_3.?a) \rightarrow ?a} (R_2 \bowtie_{(R_2.?a=R_3.?a \vee R_2.?a=NULL \vee R_3.?a=NULL)} R_3) = \{(B_2, email, john@john.edu, NULL, NULL), (B_4, email, ringo@ringo.edu, web, www.starr.edu)\}$

$eval(Q_3, G) = \pi_{?a, ?n, ?e, ?w}(\dagger_{(R_1.?a, R_4.?a) \rightarrow ?a} (R_1 \bowtie_{(R_1.?a=R_4.?a \vee R_1.?a=NULL \vee R_4.?a=NULL)} R_4)) =$

?a	?n	?e	?w
B ₁	paul	NULL	NULL
B ₂	john	john@john.edu	NULL
B ₃	george	NULL	NULL
B ₄	ringo	ringo@ringo.edu	www.starr.edu

The fourth query includes two nested *OPT* operators, however the query contains so called “not-well-designed” graph pattern [35], i.e., ?x occurs in both (?x, name, paul) and (?x, email, ?z), but not in the intermediate subpattern (?y, name, george).

Q_4 : SELECT ?x, ?y, ?z WHERE ((?x, name, paul) OPT ((?y, name, george) OPT (?x, email, ?z))).

$R_1 = eval((?x, name, paul), G) = \{(B_1, name, paul)\}$

$R_2 = eval((?y, name, george), G) = \{(B_3, name, george)\}$

$R_3 = eval((?x, email, ?z), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$

$R_4 = eval((?y, name, george) OPT(?x, email, ?z), G) = (R_2 \bowtie_{(true)} R_3) =$

$\{(B_3, name, george, B_2, email, john@john.edu), (B_3, name, george, B_4, email, ringo@ringo.edu)\}$

$eval(Q_4, G) = \pi_{?x, ?y, ?z}(\dagger_{(R_1.?x, R_4.?x) \rightarrow ?x, (R_1.name, R_4.name) \rightarrow name} (R_1$

$\bowtie_{(R_1.?x=R_4.?x \vee R_1.?x=NULL \vee R_4.?x=NULL) \wedge (R_1.name=R_4.name \vee R_1.name=NULL \vee R_4.name=NULL)} R_4)) =$

?x	?y	?z
B ₁	NULL	NULL

The last query includes *AND* and *UNION* operators. This query is interesting because triple patterns that participate in the *UNION* contain the same variables ?a and ?p, while differ in predicate patterns *phone* and *cell*.

Q_5 : SELECT ?a, ?n, ?p WHERE ((?a, name, ?n) AND ((?a, phone, ?p) UNION (?a, cell, ?p))).

$R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$

$R_2 = eval((?a, phone, ?p), G) = \{(B_1, phone, 111 - 1111), (B_4, phone, 444 - 4444)\}$

$R_3 = eval((?a, cell, ?p), G) = \{(B_4, phone, 444 - 4444)\}$

$R_4 = eval((?a, phone, ?p) UNION (?a, cell, ?p), G) = R_2 \uplus R_3 =$

$\{(B_1, phone, 111 - 1111, NULL), (B_4, phone, 444 - 4444, NULL), (B_4, NULL, 444 - 4444, cell)\}$

$R_5 = \dagger_{(R_1.?a, R_4.?a) \rightarrow ?a} (R_1 \bowtie_{(R_1.?a=R_4.?a \vee R_1.?a=NULL \vee R_4.?a=NULL)} R_4) =$

?a	name	?n	phone	?p	cell
B ₁	name	paul	phone	111 - 1111	NULL
B ₄	name	ringo	phone	444 - 4444	NULL
B ₄	name	ringo	NULL	444 - 4444	cell

$eval(Q_5, G) = \pi_{?a, ?n, ?p}(R_5) =$

?a	?n	?p
B ₁	paul	111 - 1111
B ₄	ringo	444 - 4444

◇

The above examples illustrate that our proposed semantics *eval* gives the same solutions as the mapping-based semantics under interpretation function λ . In the following, we prove that the relational algebra based semantics *eval* is equivalent to the mapping-based semantics $[[\cdot]]$ defined in [35] under interpretation function λ .

Theorem 3.24 *Given a SPARQL query $sparql \in \mathcal{Q}$ and an RDF graph G , $eval$ is equivalent to $[[\cdot]]$ under interpretation λ , i.e., $\lambda(eval(sparql, G)) \equiv [[sparql]]_G$.*

Proof:

We start with presenting our intuition and then pursue the proof.

The definition of $[[\cdot]]$ is based on partial mappings $\mu : V \rightarrow IBL$. The domain of μ , $dom(\mu)$, is the subset of V over which μ is defined. The empty mapping μ_\emptyset is the mapping with empty domain. 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 $\mu_1(x) = \mu_2(x)$; mappings with disjoint domains are always compatible; and μ_\emptyset is compatible with any other mapping. A set of mappings is denoted as Ω and binary operators of join, left outer join, and union are defined on sets of mappings, e.g., $\Omega_1 \bowtie \Omega_2$, $\Omega_1 \bowtie \Omega_2$, and $\Omega_1 \cup \Omega_2$. In $[[\cdot]]$, Ω encodes a SPARQL query solution.

In the case of *eval*, we encode a SPARQL query solution as a relation R , however, unlike in Ω , we map not only variables, but also literals and URIs. Each tuple $r \in R$ is a total mapping $r : IVL \rightarrow IBL \cup \{\text{NULL}\}$, i.e., a URI or a literal is mapped to itself or to **NULL**, and a variable is mapped to an element of set $IBL \cup \{\text{NULL}\}$. Thus, a relation R denotes a set of tuples (mappings) and each tuple is defined on the schema (mapping domain) $\xi(R)$. The core difference between relation R and set Ω is that all tuples $r \in R$ have the same schema and contain **NULL**s for unbound values, while mappings $\mu \in \Omega$ may have different domains and are not defined for unbound values. Therefore, the notion of compatibility is adapted to ignore unbound or **NULL** values as follows. Two tuples $r_1 \in R_1$ and $r_2 \in R_2$ are compatible when for all $x \in \xi(R_1) \cap \xi(R_2)$, it is the case that $r_1(x) = r_2(x)$ or $r_1(x) = \text{NULL}$ or $r_2(x) = \text{NULL}$; tuples with disjoint schemas are always compatible. This new compatibility is ensured in the conditions of our join (see Rule 8) and left outer join (see Rule 9) between relations R_1 and R_2 . In addition, the redundant relational attributes generated by a join are merged into one attribute with non-**NULL** value, when possible, using our \dagger operator. Similarly, as we do for the join and left outer join, we encode mapping-based union into its relational equivalent – outerunion. Thus, all the operators over mapping sets in $[[\cdot]]$ are given their relational equivalents in *eval*. It can be shown that all the rules in the *eval* definition return solutions that are equivalent to the solutions of the corresponding rules of $[[\cdot]]$ under interpretation λ (see Definition 3.9).

We pursue the proof by structural induction. The induction basis is $\lambda(eval(tp, G)) \equiv [[tp]]_G$, i.e., triple pattern evaluation with *eval* and $[[\cdot]]$ results in equivalent solutions. To prove the induction basis, we show that for any $\mu \in \lambda(eval(tp, G))$, it is the case that $\mu \in [[tp]]_G$, and vice-versa, which can be proved by contradiction as follows. First, it is easy to see that conditions for a triple pattern tp to match a triple $t \in G$ in Rules 7 and 1, although expressed differently, are equivalent, i.e., tp matches t if URIs and literals in tp match (are equal to) URIs and literals in t at corresponding positions, and the same variables in tp match the same values in t . Second, let $\mu \in \lambda(eval(tp, G))$ and $\mu(tp) = t$, where t is the triple obtained by replacing the variables in tp according to μ , then it must be true that $t \in G$ (see Rule 7). Assume that $\mu \notin [[tp]]_G$, then it must be true that $\mu(tp) = t \notin G$ (see Rule 1), which contradicts to the previous statement. Therefore, $\mu \in [[tp]]_G$. Finally, to prove the other direction, let $\mu \in [[tp]]_G$, then it must be true that $\mu(tp) = t \in G$ (see Rule 1). Assume that $\mu \notin \lambda(eval(tp, G))$, then it must be true that $\mu(tp) = t \notin G$ (see Rule 7), which contradicts to the previous statement. Therefore, $\mu \in \lambda(eval(tp, G))$.

Next, we define an induction hypothesis and prove an induction step for every remaining pair of rules of *eval* and $[[\cdot]]$. For the evaluation of $gp_1 \text{ AND } gp_2$ (see Rules 8 and 2), the induction hypothesis is $\lambda(eval(gp_1, G)) \equiv [[gp_1]]_G$ and $\lambda(eval(gp_2, G)) \equiv [[gp_2]]_G$, and the induction step is $\lambda(eval(gp_1 \text{ AND } gp_2, G)) \equiv [[gp_1 \text{ AND } gp_2]]_G$. The proof of the induction step is as follows. Let $r_1 \in eval(gp_1, G)$, $r_2 \in eval(gp_2, G)$, $\mu_1 \in [[gp_1]]_G$, $\mu_2 \in [[gp_2]]_G$, $\lambda(r_1) = \mu_1$, and $\lambda(r_2) = \mu_2$, where λ is applied on a single tuple r_1 or r_2 . We show that the relational join of r_1 and r_2 produces the same tuple under interpretation λ as the mapping-based join of μ_1 and μ_2 . First, note that common URIs and literals in $\xi(r_1)$ and $\xi(r_2)$ do not affect the join condition, i.e., expressions (see Rule 8) of the form $(r_1(a_i) = r_2(a_i) \vee r_1(a_i) = \text{NULL} \vee r_2(a_i) = \text{NULL})$ that involve such attributes are always evaluated to *true*, because common URI/literal attributes can only be either both bound to the same value or arbitrarily unbound (**NULL**). Therefore, the join conditions in Rules 8 and 2 only depend on variable bindings, which must be respectively the same since $\lambda(r_1) = \mu_1$, and $\lambda(r_2) = \mu_2$. Second, there may be unbound (**NULL**) variables in $\xi(r_1)$ or $\xi(r_2)$ that are not in $dom(\mu_1)$ or $dom(\mu_2)$, respectively, however they do not affect the join condition. Therefore, the compatibility property is identical in both rules. Finally, the relational join resulting redundant attributes are merged into a single attribute with non-**NULL** value, when possible, using our \dagger operator. Therefore, it is true that the relational join of r_1 and r_2 produces the same tuple under interpretation λ as the mapping-based join of μ_1 and μ_2 , and, under the given induction hypothesis, $\lambda(eval(gp_1 \text{ AND } gp_2, G)) \equiv [[gp_1 \text{ AND } gp_2]]_G$.

For the evaluation of $gp_1 \text{ OPT } gp_2$ (see Rules 9 and 3), the induction hypothesis is $\lambda(\text{eval}(gp_1, G)) \equiv [[gp_1]]_G$ and $\lambda(\text{eval}(gp_2, G)) \equiv [[gp_2]]_G$, and the induction step is $\lambda(\text{eval}(gp_1 \text{ OPT } gp_2, G)) \equiv [[gp_1 \text{ OPT } gp_2]]_G$. The proof of the induction step is very similar to the proof of the induction step for $gp_1 \text{ AND } gp_2$ above.

For the evaluation of $gp_1 \text{ UNION } gp_2$ (see Rules 10 and 4), the induction hypothesis is $\lambda(\text{eval}(gp_1, G)) \equiv [[gp_1]]_G$ and $\lambda(\text{eval}(gp_2, G)) \equiv [[gp_2]]_G$, and the induction step is $\lambda(\text{eval}(gp_1 \text{ UNION } gp_2, G)) \equiv [[gp_1 \text{ UNION } gp_2]]_G$. The proof of the induction step is as follows. Let $r_1 \in \text{eval}(gp_1, G)$, $r_2 \in \text{eval}(gp_2, G)$, $\mu_1 \in [[gp_1]]_G$, $\mu_2 \in [[gp_2]]_G$, $\lambda(r_1) = \mu_1$, and $\lambda(r_2) = \mu_2$, where λ is applied on a single tuple r_1 or r_2 . We show that the relational outerunion of r_1 and r_2 produces the same tuple under interpretation λ as the mapping-based union of μ_1 and μ_2 . There are two possible cases when union two mappings. First, when μ_1 and μ_2 are identical mappings, the union result will contain only one of them. r_1 and r_2 may or may not be identical. If r_1 and r_2 are identical, the outerunion result will contain only one of them and it is given that $\lambda(r_1) = \mu_1$ and $\lambda(r_2) = \mu_2$. If r_1 and r_2 are not identical, the outerunion result will contain both of them (possibly with extended schemas), however after applying λ to the resulting relation and obtaining a set of mappings, this set will contain only one mapping out of $\lambda(r_1)$ and $\lambda(r_2)$, since $\lambda(r_1) = \mu_1$, $\lambda(r_2) = \mu_2$, and μ_1 and μ_2 are identical. Second, when μ_1 and μ_2 differ in either their domains or their ranges (or both), the union result will contain both mappings. Similarly, r_1 and r_2 will differ and the outerunion result will contain both of them (possibly with extended schemas), and after applying λ to the resulting relation and obtaining a set of mappings, this set will contain $\lambda(r_1)$ and $\lambda(r_2)$, since $\lambda(r_1) = \mu_1$, $\lambda(r_2) = \mu_2$, and μ_1 and μ_2 are not identical. Therefore, under the given induction hypothesis, $\lambda(\text{eval}(gp_1 \text{ UNION } gp_2, G)) \equiv [[gp_1 \text{ UNION } gp_2]]_G$.

For the evaluation of $gp \text{ FILTER } expr$ (see Rules 11 and 5), the induction hypothesis is $\lambda(\text{eval}(gp, G)) \equiv [[gp]]_G$ and the induction step is $\lambda(\text{eval}(gp \text{ FILTER } expr, G)) \equiv [[gp \text{ FILTER } expr]]_G$. The proof of the induction step is as follows. The semantics of expression $expr$ is encoded similarly in both eval and $[[\cdot]]$ with the difference that $[[\cdot]]$ relies on a mapping domain to detect unbound variables and eval checks whether the corresponding attribute is not NULL for the same purpose, because unbound variables are represented in a relational schema. Since URI/literal attributes do not participate in the filtering and $\lambda(\text{eval}(gp, G)) \equiv [[gp]]_G$, it is true that $\lambda(\text{eval}(gp \text{ FILTER } expr, G)) \equiv [[gp \text{ FILTER } expr]]_G$.

For the evaluation of $SELECT (v_1, v_2, \dots, v_n) \text{ WHERE } (gp)$ (see Rules 12 and 6), the induction hypothesis is $\lambda(\text{eval}(gp, G)) \equiv [[gp]]_G$ and the induction step is $\lambda(\text{eval}(SELECT (v_1, v_2, \dots, v_n) \text{ WHERE } (gp), G)) \equiv [[SELECT (v_1, v_2, \dots, v_n) \text{ WHERE } (gp)]]_G$. It is trivial that the induction step is true under the given induction hypothesis, since only the projection operator is involved in the evaluation.

Since all the corresponding rules of eval and $[[\cdot]]$ are proved to produce equivalent solutions, it can be concluded that $\forall sparql \in \mathcal{Q}$, $\lambda(\text{eval}(sparql, G)) \equiv [[sparql]]_G$. \square

The presented relational algebra based semantics of SPARQL provides an important bridge between Semantic Web and relational databases and serves as the foundation for SPARQL query processing using a relational database query engine.

4 Semantics Preserving SPARQL-to-SQL Translation

In this section, we define our SPARQL-to-SQL query translation for an RDBMS-based RDF store and prove that the translation is semantics preserving with respect to the relational algebra based semantics of SPARQL.

In order to support a generic translation of SPARQL queries into equivalent SQL queries, we need a generic representation for an RDBMS-based RDF store scheme, in which the following information will be modeled: (1) which relation is used to store RDF triples that can potentially match a triple pattern, and (2) which relational attributes of the relation are used to store the components (subjects, predicates, and objects) of triples. To capture these two information, we formalize an RDBMS-based RDF store scheme as the following two RDF-to-Relational mappings α and β . In this paper, we study the set of schemes \mathcal{S} for which both α and β are many-to-one mappings.

Definition 4.1 (Mapping α) Given a set of all possible triple patterns $TP = (IVL) \times (IV) \times (IVL)$ and a set of relations REL in an RDBMS-based RDF store, a mapping α is a many-to-one mapping $\alpha : TP \rightarrow REL$, if given a triple pattern $tp \in TP$, $\alpha(tp)$ is a relation in which all the triples that may match tp are stored. \diamond

Definition 4.2 (Mapping β) Given a set of all possible triple patterns $TP = (IVL) \times (IV) \times (IVL)$, a set $POS = \{sub, pre, obj\}$, and a set of relational attributes ATR in an RDBMS-based RDF store, a mapping β is a many-to-one mapping $\beta : TP \times POS \rightarrow ATR$, if given a triple pattern $tp \in TP$ and a position $pos \in POS$, $\beta(tp, pos)$ is a relational attribute whose value may match tp at position pos . \diamond

An example of mappings α and β for different RDBMS-based RDF store schemes is presented in the following.

Example 4.3 (Mappings α and β) First, consider an RDBMS-based RDF store that employs a single relation $Triple(s, p, o)$ to store RDF triples. For the RDF graph G in Figure 2, the relation is as follows:

$$Triple = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_1 & name & paul \\ \hline B_1 & phone & 111 - 1111 \\ \hline B_2 & name & john \\ \hline B_2 & email & john@john.edu \\ \hline \dots & \dots & \dots \\ \hline B_4 & cell & 444 - 4444 \\ \hline \end{array}$$

In this case, for any triple pattern tp , $\alpha(tp) = \mathbf{Triple}$, $\beta(tp, sub) = \mathbf{s}$, $\beta(tp, pre) = \mathbf{p}$, and $\beta(tp, obj) = \mathbf{o}$.

Second, consider an RDBMS-based RDF store that employs relation $Triple(s, p, o)$, as well as so called *property* relations $P_{p_i}(s, p, o)$, where p_i is a particular predicate (property). Each relation P_{p_i} is the result of partitioning relation $Triple$ based on a predicate value p_i , e.g.,

$$P_{name} = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_1 & name & paul \\ \hline B_2 & name & john \\ \hline B_3 & name & george \\ \hline B_4 & name & ringo \\ \hline \end{array}, P_{phone} = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_1 & phone & 111 - 1111 \\ \hline B_4 & phone & 444 - 4444 \\ \hline \end{array}, P_{email} = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_2 & email & john@... \\ \hline B_4 & email & ringo@... \\ \hline \end{array},$$

and similarly for P_{web} and P_{cell} .

In this case, α and β can be calculated as follows. For any triple pattern tp , if $tp.pp \notin V$, then $\alpha(tp) = P_{tp.pp}$, otherwise $\alpha(tp) = \mathbf{Triple}$; $\beta(tp, sub) = \mathbf{s}$, $\beta(tp, pre) = \mathbf{p}$, and $\beta(tp, obj) = \mathbf{o}$.

Finally, consider an RDBMS-based RDF store that employs relation $Triple(s, p, o)$, *property* relations $P_{p_i}(s, p, o)$, as well as so called *subject* relations S_{s_j} and *object* relations O_{o_k} , where s_j (o_k) is a particular subject (object). Each relation S_{s_j} (O_{o_k}) is the result of partitioning relation $Triple$ based on a subject (object) value s_j (o_k), e.g.,

$$S_{B_1} = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_1 & name & paul \\ \hline B_1 & phone & 111 - 1111 \\ \hline \end{array}, O_{paul} = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_1 & name & paul \\ \hline \end{array}, S_{B_2} = \begin{array}{|c|c|c|} \hline s & p & o \\ \hline B_2 & name & john \\ \hline B_2 & email & john@... \\ \hline \end{array},$$

and so forth.

In this case, α and β can be calculated as follows. For any triple pattern tp , if $tp.sp \notin V$, then $\alpha(tp) = P_{tp.sp}$, otherwise if $tp.op \notin V$, then $\alpha(tp) = P_{tp.op}$, otherwise if $tp.pp \notin V$, then $\alpha(tp) = P_{tp.pp}$, otherwise $\alpha(tp) = \mathbf{Triple}$; $\beta(tp, sub) = \mathbf{s}$, $\beta(tp, pre) = \mathbf{p}$, and $\beta(tp, obj) = \mathbf{o}$. \diamond

The two mappings provide a foundation for a schema-independent SPARQL-to-SQL translation, such that the relational schema design, which concerns about α and β , is fully separated from the translation procedure which is parameterized by α and β . We check the relational database schemas of 11 existing RDF stores, including Jena [43, 31], Sesame [12], 3store [25, 26], KAON [42], RStar [30], OpenLink Virtuoso [21], DLDB [33], RDFSuite [7, 41], DBOWL [32], PARKA [40], and ProvRDF [14], and confirm that α and β can be derived for all of them. To achieve this, there are three minor issues that we should address as described in the following.

First, many of the existing RDF stores employ normalized database schemas. For example, one relation $Triple(s, p, o)$ can be used to store all triples, however URIs and literals in this relation are substituted with integer IDs. The mappings from IDs to URIs and literals are stored in two other relations. This design

minimizes data redundancy in the database and facilitates faster indexes on numeric values. However, the maintenance of these mappings, as well as query processing in such a setting, is expensive. As a result, some of the systems switch to denormalized schemas, e.g., Jena1 uses a normalized schema, while Jena2 employs a denormalized schema [43]. Our mappings α and β work for denormalized database schemas naturally; to deal with normalized schemas, we propose to create a denormalized view of a database and derive α and β with respect to this view. For example, given relation $Triple(s,p,o)$ and two relations with ID-to-URI and ID-to-literal mappings, one can create a view $TripleView(s,p,o)$ that joins the available relations to substitute IDs with actual URIs and literals. Creating such a denormalized database view is quite simple and enables α and β to encode schemas of the following RDF stores: Jena, Sesame, 3store, KAON, RStar, OpenLink Virtuoso, DBOWL, and the schema-oblivious version of ProvRDF.

Second, to support some other DRF stores, β should be a partial mapping. For example, property relations of the form $P_{p_i}(s,p,o)$, where p_i is a particular predicate (property), are usually simplified as $P_{p_i}(s,o)$, because the relation name itself encodes the name of the predicate p_i and attribute p , which always stores the value of p_i , can be dropped. Therefore, β may be undefined for the predicate position pre , i.e., $\beta(tp,pre) = undef$. In this paper, our translation is defined for the total mappings to keep the presentation simple. However, it is quite straightforward to adapt the translation to the partial mappings by simply ignoring undefined values in SQL projection lists and join/selection conditions. For more details, please refer to our work in [14] that deals with the SPARQL-to-SQL translation with the partial mappings.

Finally, in some RDF stores, such as DLDB, RDFSuite, and PARKA, the RDF-to-Relational mappings should be many-to-many mappings. For example, to retrieve all triples from the above RDF stores, one needs to select all triples from all property relations and union them. Therefore, in this case, α is a many-to-many mapping. To avoid many-to-many mappings, one can create a view, e.g., $TripleView(s,p,o)$, that stores all triples in the system and thus, always derive α and β as many-to-one mappings, since this view alone can answer any query without the need to access multiple relations for some queries. A more efficient solution based on the many-to-many versions of α and β exists; we leave out such details for the simplicity of presentation.

In addition to mappings α and β , our translation uses five auxiliary functions. The first three functions are (1) a function *alias* that generates a unique alias for a relation, (2) a function *terms* that returns a set of all the terms in a graph pattern, such that each term is in *IVL*, and (3) a function *name* that, given a term in *IVL*, generates a unique name, such that the generated name conforms to the SQL syntax for relational attribute names (e.g., SPARQL variables can be “renamed” by simply removing initial ‘?’ or ‘\$’). The other two functions are (4) *genCond-SQL* and (5) *genPR-SQL* which are similar to the previously defined *genCond* and *genPR*, but generate expressions in SQL syntax.

Functions *genCond-SQL* and *genPR-SQL* are defined in Figure 6. Function *genCond-SQL*, given a triple pattern tp and a mapping β , generates an SQL boolean expression which is evaluated to *true* if and only if tp matches a tuple represented by relational attributes $\beta(tp,sub)$, $\beta(tp,pre)$, and $\beta(tp,obj)$. The boolean expression ensures that if $tp.sp$ is not a variable (a variable can match any RDF term), it must be true that $\beta(tp,sub) = 'tp.sp'$; similar conditions are introduced for $tp.pp$ and $tp.op$. Also, if $tp.sp = tp.pp$, then for tp match the tuple, it must be true that $\beta(tp,sub) = \beta(tp,pre)$; similarly for the cases when $tp.sp = tp.op$ and $tp.op = tp.pp$.

Example 4.4 (Function *genCond-SQL*) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, email, ?a)$ and mapping β defined as $\beta(tp,sub) = s$, $\beta(tp,pre) = p$, and $\beta(tp,obj) = o$ for any triple pattern tp , *genCond-SQL* generates the following conditions for tp_1 and tp_2 to match a tuple represented by $(\beta(tp,sub), \beta(tp,pre), \beta(tp,obj))$:

$$\begin{aligned} genCond-SQL(tp_1, \beta) &= \text{“True And } \beta(tp_1,pre) = 'tp_1.pp' \text{”} = \text{“True And } p = \text{‘email’”} \\ genCond-SQL(tp_2, \beta) &= \text{“True And } \beta(tp_2,pre) = 'tp_2.pp' \text{ And } \beta(tp_2,sub) = \beta(tp_2,obj) \text{”} \\ &= \text{“True And } p = \text{‘email’ And } s = o \text{”} \end{aligned}$$

◇

Function *genPR-SQL*, given a triple pattern tp , a mapping β , and a function *name*, generates an SQL expression which can be used to project only those relational attributes that correspond to distinct $tp.sp$, $tp.pp$, and $tp.op$ and rename the projected attributes as $\beta(tp,sub) \rightarrow name(tp.sp)$, $\beta(tp,pre) \rightarrow name(tp.pp)$, and $\beta(tp,obj) \rightarrow name(tp.op)$. $\beta(tp,sub)$ is always projected and renamed into $name(tp.sp)$, $\beta(tp,pre)$ is projected and renamed into $name(tp.pp)$ if $tp.pp \neq tp.sp$, and $\beta(tp,pre)$ is projected and renamed into

$name(tp.op)$ if $tp.op \neq tp.sp$ and $tp.op \neq tp.pp$. Later, we use this function to generate the project-and-rename attribute list of a relation $\alpha(tp)$, where $\alpha(tp)$ stores all the tuples that may match tp . This ensures that, after projection and renaming, the schema of the resulting relation does not have duplicate attribute names.

Example 4.5 (Function $genPR\text{-}SQL$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, email, ?a)$, mapping β defined as $\beta(tp, sub) = s$, $\beta(tp, pre) = p$, and $\beta(tp, obj) = o$ for any triple pattern tp , and function $name$ (e.g., $name(?a) = a$, $name(?e) = e$, and $name(email) = email$), $genPR\text{-}SQL$ generates the following SQL strings:

```

genPR-SQL(tp1,  $\beta$ , name) = " $\beta(tp_1, sub)$  As  $name(tp_1.sp)$ ,  $\beta(tp_1, pre)$  As  $name(tp_1.pp)$ ,
                              $\beta(tp_1, obj)$  As  $name(tp_1.op)$ "
                             = "s As a, p As email, o As e".
genPR-SQL(tp2,  $\beta$ , name) = " $\beta(tp_2, sub)$  As  $name(tp_2.sp)$ ,  $\beta(tp_2, pre)$  As  $name(tp_2.pp)$ "
                             = "s As a, p As email".

```

◇

```

01 Function genCond-SQL
02 Input: triple pattern  $tp$ , mapping  $\beta$ 
03 Output: SQL boolean expression  $cond$  which is true
04 iff a relational tuple represented by  $\beta(tp, sub)$ ,  $\beta(tp, pre)$ , and  $\beta(tp, obj)$  matches  $tp$ 
05 Begin
06    $cond = \text{"True"}$ 
07   If  $tp.sp \notin V$  then  $cond += \text{" And } \beta(tp, sub) = 'tp.sp' \text{"}$  End If
08   If  $tp.pp \notin V$  then  $cond += \text{" And } \beta(tp, pre) = 'tp.pp' \text{"}$  End If
09   If  $tp.op \notin V$  then  $cond += \text{" And } \beta(tp, obj) = 'tp.op' \text{"}$  End If
10   If  $tp.sp = tp.pp$  then  $cond += \text{" And } \beta(tp, sub) = \beta(tp, pre) \text{"}$  End If
11   If  $tp.sp = tp.op$  then  $cond += \text{" And } \beta(tp, sub) = \beta(tp, obj) \text{"}$  End If
12   If  $tp.op = tp.pp$  then  $cond += \text{" And } \beta(tp, obj) = \beta(tp, pre) \text{"}$  End If
13 Return  $cond$ 
14 End Function

15 Function genPR-SQL
16 Input: triple pattern  $tp$ , mapping  $\beta$ , function  $name$ 
17 Output: SQL expression which projects only those relational attributes that correspond to
18 distinct  $tp.sp$ ,  $tp.pp$ , and  $tp.op$  and renames the projected attributes as
19  $\beta(tp, sub) \rightarrow name(tp.sp)$ ,  $\beta(tp, pre) \rightarrow name(tp.pp)$ , and  $\beta(tp, obj) \rightarrow name(tp.op)$ 
20 Begin
21    $pr\text{-}list = \text{"}\beta(tp, sub) \text{ As } name(tp.sp)\text{"}$ 
22   If  $tp.pp \neq tp.sp$  then  $pr\text{-}list += \text{"}, \beta(tp, pre) \text{ As } name(tp.pp)\text{"}$  End If
23   If  $tp.op \neq tp.sp$  and  $tp.op \neq tp.pp$  then  $pr\text{-}list += \text{"}, \beta(tp, obj) \text{ As } name(tp.op)\text{"}$  End If
24 Return  $pr\text{-}list$ 
25 End Function

```

Figure 6: Functions $genCond\text{-}SQL$ and $genPR\text{-}SQL$

In the rest of the examples in this section, we assume that for any triple pattern tp , $\alpha(tp) = \text{Triple}$, $\beta(tp, sub) = s$, $\beta(tp, pre) = p$, and $\beta(tp, obj) = o$; function $name$, given a variable $?v \in V$ or a URI uri , returns strings $name(?v) = v$ and $name(uri) = uri$ that conform to the SQL syntax for relational attribute names. In addition, for brevity, all SQL boolean expressions of the form “True And $subexpression$ ” are simplified as “ $subexpression$ ”.

We define the SPARQL-to-SQL translation as a function $trans$, which takes a graph pattern expression or a SPARQL query, an RDBMS-based RDF store scheme represented by α and β , and returns an SQL query. The computation of $trans$ is defined in Figure 7 and is explained in the following.

Rule 13 defines the translation of a triple pattern tp into SQL over an RDBMS-based RDF store represented by α and β . The resulting SQL query retrieves tuples of the form $(\beta(tp, sub), \beta(tp, pre), \beta(tp, obj))$ from relation $\alpha(tp)$, where each matching tuple must satisfy the condition generated by $genCond\text{-}SQL(tp, \beta)$ in the SQL **Where** clause. The relational attributes are projected and renamed using the projection list generated by $genPR\text{-}SQL(tp, \beta, name)$ in the SQL **Select** clause.

Example 4.6 (Rule 13: $trans(tp, \alpha, \beta)$) The translation of two sample triple patterns into SQL is as follows.

```

trans((?a, email, ?e),  $\alpha, \beta$ ) = Select Distinct s As a, p As email, o As e From Triple Where p='email'
trans((?a, web, ?w),  $\alpha, \beta$ ) = Select Distinct s As a, p As web, o As w From Triple Where p='web'

```

$trans(tp, \alpha, \beta) =$

```
Select Distinct genPR-SQL(tp,  $\beta$ , name) From  $\alpha$ (tp) Where genCond-SQL(tp,  $\beta$ );
```

 (13)

$trans(gp_1 \text{ AND } gp_2, \alpha, \beta) =$

```
Select Distinct name(a), [a|a ∈ (terms(gp1) – terms(gp2))] name(b), [b|b ∈ (terms(gp2) – terms(gp1))]
Coalesce(r1.name(c), r2.name(c)) As name(c), [c|c ∈ (terms(gp1) ∩ terms(gp2))]
From ( trans(gp1,  $\alpha$ ,  $\beta$ ) ) r1 Inner Join ( trans(gp2,  $\alpha$ ,  $\beta$ ) ) r2
On (True And [c|c ∈ (terms(gp1) ∩ terms(gp2))]
(r1.name(c)=r2.name(c) Or r1.name(c) Is Null Or r2.name(c) Is Null));
where r1 = alias() and r2 = alias().
```

 (14)

$trans(gp_1 \text{ OPT } gp_2, \alpha, \beta) =$

```
Select Distinct name(a), [a|a ∈ (terms(gp1) – terms(gp2))] name(b), [b|b ∈ (terms(gp2) – terms(gp1))]
Coalesce(r1.name(c), r2.name(c)) As name(c), [c|c ∈ (terms(gp1) ∩ terms(gp2))]
From ( trans(gp1,  $\alpha$ ,  $\beta$ ) ) r1 Left Outer Join ( trans(gp2,  $\alpha$ ,  $\beta$ ) ) r2
On (True And [c|c ∈ (terms(gp1) ∩ terms(gp2))]
(r1.name(c)=r2.name(c) Or r1.name(c) Is Null Or r2.name(c) Is Null));
where r1 = alias() and r2 = alias().
```

 (15)

$trans(gp_1 \text{ UNION } gp_2, \alpha, \beta) =$

```
Select name(a) [a|a ∈ A], name(b) [b|b ∈ B], r1.name(c) [c|c ∈ C] As name(c)
From ( trans(gp1,  $\alpha$ ,  $\beta$ ) ) r1 Left Outer Join ( trans(gp2,  $\alpha$ ,  $\beta$ ) ) r2 On (False)
Union
Select name(a) [a|a ∈ A], name(b) [b|b ∈ B], r3.name(c) [c|c ∈ C] As name(c)
From ( trans(gp2,  $\alpha$ ,  $\beta$ ) ) r3 Left Outer Join ( trans(gp1,  $\alpha$ ,  $\beta$ ) ) r4 On (False);
where r1, r2, r3, and r4 = alias(); A, B, and C are ordered sets (terms(gp1) – terms(gp2)),
(terms(gp2) – terms(gp1)), and (terms(gp1) ∩ terms(gp2)), respectively.
```

 (16)

$trans(gp \text{ FILTER } expr, \alpha, \beta) =$

```
Select * From ( trans(gp,  $\alpha$ ,  $\beta$ ) ) alias() Where transexpr(expr);
```

 (17)

$trans(SELECT (v_1, v_2, \dots, v_n) WHERE(gp), \alpha, \beta) =$

```
Select Distinct name(v1), name(v2), \dots, name(vn) From ( trans(gp,  $\alpha$ ,  $\beta$ ) ) alias();
```

 (18)

The procedure for translating filter constraints *expr* into SQL syntax is

- transexpr*(*expr*) : Replace (1) each variable *v* with *name*(*v*)
- (2) each literal, URI, and numeric value *l* with ‘*l*’
- (3) logical connectives \neg , \vee , and \wedge with Not, Or, and And
- (4) *bound*(*X*) with *X* Is Not Null

Figure 7: SPARQL-to-SQL translation

◇

Rule 14 defines the translation of the *AND* of two graph patterns gp_1 and gp_2 as the inner join of the relations that correspond to graph pattern translations $trans(gp_1, \alpha, \beta)$ and $trans(gp_2, \alpha, \beta)$ and are assigned aliases r_1 and r_2 , respectively. The join condition ensures that common attributes $r_1.name(c)$ and $r_2.name(c)$ are equal or one or both of them are NULLs, for each $c \in (terms(gp_1) \cap terms(gp_2))$; if there are no common attributes, the condition is “True”, resulting in the cross-product of r_1 and r_2 . The relational attributes of the join resulting relation are projected as follows: (1) unique attributes of both relations are projected per se and (2) common attributes are projected as $Coalesce(r_1.name(c), r_2.name(c))$ As $name(c)$. The SQL construct *Coalesce*, similar to the \dagger operator, returns the value of $r_1.name(c)$, if it is non-NULL, and the value of $r_2.name(c)$, otherwise. Therefore, the redundant attributes are combined into one single attribute that is renamed into $name(c)$.

Example 4.7 (Rule 14: $trans(gp_1 \text{ AND } gp_2, \alpha, \beta)$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the translation of the graph pattern $gp = (tp_1 \text{ AND } tp_2)$ into SQL is as follows.

```

q1 = trans((?a, email, ?e),  $\alpha, \beta$ ) = Select Distinct s As a, p As email, o As e From Triple Where p='email'
q2 = trans((?a, web, ?w),  $\alpha, \beta$ ) = Select Distinct s As a, p As web, o As w From Triple Where p='web'
trans(gp,  $\alpha, \beta$ ) = Select Distinct email,e,web,w,Coalesce(r1.a,r2.a) As a From (q1) r1
Inner Join (q2) r2 On (r1.a = r2.a Or r1.a Is Null Or r2.a Is Null)

```

◇

Rule 15 defines the translation of the *OPT* of two graph patterns gp_1 and gp_2 as the left outer join of the relations that correspond to graph pattern translations $trans(gp_1, \alpha, \beta)$ and $trans(gp_2, \alpha, \beta)$ and are assigned aliases r_1 and r_2 , respectively. The join condition in the *On* clause and the projection in the *Select* clause are analogous to the previous rule. The only difference between the translations of *AND* and *OPT* operators is the use of inner join and left outer join, respectively.

Example 4.8 (Rule 15: $trans(gp_1 \text{ OPT } gp_2, \alpha, \beta)$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the translation of the graph pattern $gp = (tp_1 \text{ OPT } tp_2)$ into SQL is as follows.

```

q1 = trans((?a, email, ?e),  $\alpha, \beta$ ) = Select Distinct s As a, p As email, o As e From Triple Where p='email'
q2 = trans((?a, web, ?w),  $\alpha, \beta$ ) = Select Distinct s As a, p As web, o As w From Triple Where p='web'
trans(gp,  $\alpha, \beta$ ) = Select Distinct email,e,web,w,Coalesce(r1.a,r2.a) As a From (q1) r1
Left Outer Join (q2) r2 On (r1.a = r2.a Or r1.a Is Null Or r2.a Is Null)

```

◇

Rule 16 defines the translation of the *UNION* of two graph patterns gp_1 and gp_2 as the SQL *Union* of two relations represented by the two SQL statements. The first statement left outer joins relations $r_1 = trans(gp_1, \alpha, \beta)$ and $r_2 = trans(gp_2, \alpha, \beta)$ on the false condition, resulting in a relation with the tuples of r_1 NULL-padded to schema $\xi(r_1) \cup \xi(r_2)$. Similarly, the second statement left outer joins relations $r_3 = trans(gp_2, \alpha, \beta)$ and $r_4 = trans(gp_1, \alpha, \beta)$ on the false condition, resulting in a relation with the tuples of r_3 NULL-padded to schema $\xi(r_3) \cup \xi(r_4)$. Both statements project the relational attributes in the same order, such that the first projected attribute in the first statement is the same as the first projected attribute in the second statement and so forth. In particular, unique attributes of $trans(gp_1, \alpha, \beta)$, which correspond to elements of ordered set $(terms(gp_1) - terms(gp_2))$, are projected at first; unique attributes of $trans(gp_2, \alpha, \beta)$, which correspond to elements of ordered set $(terms(gp_2) - terms(gp_1))$, are projected at second; and common attributes of $trans(gp_1, \alpha, \beta)$ and $trans(gp_2, \alpha, \beta)$, which correspond to elements of ordered set $(terms(gp_1) \cap terms(gp_2))$, are projected at last.

Example 4.9 (Rule 16: $trans(gp_1 \text{ UNION } gp_2, \alpha, \beta)$) Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the translation of the graph pattern $gp = (tp_1 \text{ UNION } tp_2)$ into SQL is as follows.

```

q1 = trans((?a, email, ?e),  $\alpha, \beta$ ) = Select Distinct s As a, p As email, o As e From Triple Where p='email'
q2 = trans((?a, web, ?w),  $\alpha, \beta$ ) = Select Distinct s As a, p As web, o As w From Triple Where p='web'
trans(gp,  $\alpha, \beta$ ) = Select Distinct email,e,web,w,r1.a As a From (q1) r1
Left Outer Join (q2) r2 On (False)
Union
Select Distinct email,e,web,w,r3.a As a From (q2) r3
Left Outer Join (q1) r4 On (False)

```

◇

Rule 17 defines the translation of the *FILTER* expression *expr* for graph pattern *gp* as the selection over relation $trans(gp)$ based on condition $transexpr(expr)$. The $transexpr$ translation procedure is described in Figure 7.

Example 4.10 (Rule 17: $trans(gp \text{ FILTER } expr, \alpha, \beta)$) Given the graph pattern $gp = (?a, email, ?e) OPT (?a, web, ?w)$ and the boolean expression $expr = \neg bound(?w)$, the translation of the graph pattern $gp \text{ FILTER } expr$ into SQL is as follows. Let $trans(gp, \alpha, \beta) = q$ (see Example 4.8), then

$trans((gp \text{ FILTER } expr), \alpha, \beta) = \text{Select } * \text{ From } (q) \text{ r Where Not } (w \text{ Is Not Null})$

◇

Finally, Rule 18 defines the translation of a SPARQL query with graph pattern *gp* and projection list v_1, v_2, \dots, v_n as the projection of relational attributes $name(v_1), name(v_2), \dots, name(v_n)$ from the relation that corresponds to $trans(gp, \alpha, \beta)$.

Example 4.11 (Rule 18: $trans(SELECT (v_1, v_2, \dots, v_n) WHERE(gp), \alpha, \beta)$) Given the graph pattern $gp = (?a, email, ?e) OPT (?a, web, ?w)$, the translation of the SPARQL query $SELECT ?a, ?e, ?w WHERE(gp)$ into SQL is as follows. Let $trans(gp, \alpha, \beta) = q$ (see Example 4.8), then

$trans((SELECT ?a, ?e, ?w WHERE(gp)), \alpha, \beta) = \text{Select } a, e, w \text{ From } (q) \text{ r}$

◇

Additionally, we present the translation of several SPARQL queries whose evaluation is described in Example 3.23.

Example 4.12 (SPARQL-to-SQL translation)

As before, we assume an RDBMS-based RDF store with a single relation $\text{Triple}(s, p, o)$ that stores all the RDF triples of the RDF graph described in Figure 2. Therefore, for any triple pattern *tp*, $\alpha(tp) = \text{Triple}$, $\beta(tp, sub) = s$, $\beta(tp, pre) = p$, and $\beta(tp, obj) = o$.

The following are sample SPARQL queries and their SQL counterparts:

Q_1 : $SELECT ?a, ?n, ?e, ?w \text{ WHERE } (((?a, name, ?n) OPT (?a, email, ?e)) OPT (?a, web, ?w))$.
 $q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } name, o \text{ As } n \text{ From Triple Where } p='name'$
 $q_2 = trans((?a, email, ?e), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } email, o \text{ As } e \text{ From Triple Where } p='email'$
 $q_3 = trans((?a, web, ?w), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } web, o \text{ As } w \text{ From Triple Where } p='web'$
 $q_4 = trans(((?a, name, ?n) OPT (?a, email, ?e)), \alpha, \beta) =$
 $\text{Select Distinct } name, n, email, e, Coalesce(r1.a, r2.a) \text{ As } a \text{ From } (q_1) \text{ r1}$
 $\text{Left Outer Join } (q_2) \text{ r2 On } (r1.a = r2.a \text{ Or } r1.a \text{ Is Null Or } r2.a \text{ Is Null})$
 $trans(Q_1, \alpha, \beta) = \text{Select Distinct } a, n, e, w \text{ From } ($
 $\text{Select Distinct } name, n, email, e, web, w, Coalesce(r3.a, r4.a) \text{ As } a \text{ From } (q_4) \text{ r3}$
 $\text{Left Outer Join } (q_3) \text{ r4 On } (r3.a = r4.a \text{ Or } r3.a \text{ Is Null Or } r4.a \text{ Is Null}) \text{ r5}$

Q_2 : $SELECT ?a, ?n, ?ew \text{ WHERE } (((?a, name, ?n) OPT (?a, email, ?ew)) OPT (?a, web, ?ew))$.
 $q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } name, o \text{ As } n \text{ From Triple Where } p='name'$
 $q_2 = trans((?a, email, ?ew), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } email, o \text{ As } ew \text{ From Triple Where } p='email'$
 $q_3 = trans((?a, web, ?ew), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } web, o \text{ As } ew \text{ From Triple Where } p='web'$
 $q_4 = trans(((?a, name, ?n) OPT (?a, email, ?ew)), \alpha, \beta) =$
 $\text{Select Distinct } name, n, email, ew, Coalesce(r1.a, r2.a) \text{ As } a \text{ From } (q_1) \text{ r1}$
 $\text{Left Outer Join } (q_2) \text{ r2 On } (r1.a = r2.a \text{ Or } r1.a \text{ Is Null Or } r2.a \text{ Is Null})$
 $trans(Q_2, \alpha, \beta) = \text{Select Distinct } a, n, ew \text{ From } ($
 $\text{Select Distinct } name, n, email, web, Coalesce(r3.a, r4.a) \text{ As } a, Coalesce(r3.ew, r4.ew) \text{ As } ew$
 $\text{From } (q_4) \text{ r3 Left Outer Join } (q_3) \text{ r4 On } ((r3.a = r4.a \text{ Or } r3.a \text{ Is Null Or } r4.a \text{ Is Null})$
 $\text{And } (r3.ew = r4.ew \text{ Or } r3.ew \text{ Is Null Or } r4.ew \text{ Is Null})) \text{ r5}$

Q_3 : $SELECT ?a, ?n, ?e, ?w \text{ WHERE } ((?a, name, ?n) OPT ((?a, email, ?e) OPT (?a, web, ?w)))$.
 $q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } name, o \text{ As } n \text{ From Triple Where } p='name'$
 $q_2 = trans((?a, email, ?e), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } email, o \text{ As } e \text{ From Triple Where } p='email'$
 $q_3 = trans((?a, web, ?w), \alpha, \beta) = \text{Select Distinct } s \text{ As } a, p \text{ As } web, o \text{ As } w \text{ From Triple Where } p='web'$

$q_4 = \text{trans}(((?a, \text{email}, ?e) \text{ OPT}(?a, \text{web}, ?w)), \alpha, \beta) =$
 Select Distinct email,e,web,w,Coalesce(r1.a,r2.a) As a From (q_2) r1
 Left Outer Join (q_3) r2 On (r1.a = r2.a Or r1.a Is Null Or r2.a Is Null)
 $\text{trans}(Q_3, \alpha, \beta) =$ Select Distinct a,n,e,w From (
 Select Distinct name,n,email,e,web,w,Coalesce(r3.a,r4.a) As a From (q_1) r3
 Left Outer Join (q_4) r4 On (r3.a = r4.a Or r3.a Is Null Or r4.a Is Null)) r5

Q_4 : SELECT ?x, ?y, ?z WHERE ((?x, name, paul) OPT ((?y, name, george) OPT (?x, email, ?z))).

$q_1 = \text{trans}((?x, \text{name}, \text{paul}), \alpha, \beta) =$ Select Distinct s As x, p As name, o As paul From Triple
 Where p='name' And o='paul'
 $q_2 = \text{trans}((?y, \text{name}, \text{george}), \alpha, \beta) =$ Select Distinct s As y, p As name, o As george From Triple
 Where p='name' And o='george'
 $q_3 = \text{trans}((?x, \text{email}, ?z), \alpha, \beta) =$ Select Distinct s As x, p As email, o As z From Triple Where p='email'
 $q_4 = \text{trans}(((?y, \text{name}, \text{george}) \text{ OPT}(?x, \text{email}, ?z)), \alpha, \beta) =$
 Select Distinct y,name,george,x,email,z From (q_2) r1
 Left Outer Join (q_3) r2 On (True)
 $\text{trans}(Q_4, \alpha, \beta) =$ Select Distinct x,y,z From (
 Select Distinct paul,y,george,email,z,Coalesce(r3.x,r4.x) As x, Coalesce(r3.name,r4.name) As name
 From (q_1) r3 Left Outer Join (q_4) r4 On ((r3.x = r4.x Or r3.x Is Null Or r4.x Is Null) And (r3.name
 = r4.name Or r3.name Is Null Or r4.name Is Null))) r5

Q_5 : SELECT ?a, ?n, ?p WHERE ((?a, name, ?n) AND ((?a, phone, ?p) UNION (?a, cell, ?p))).

$q_1 = \text{trans}((?a, \text{name}, ?n), \alpha, \beta) =$ Select Distinct s As a, p As name, o As n From Triple Where p='name'
 $q_2 = \text{trans}((?a, \text{phone}, ?p), \alpha, \beta) =$ Select Distinct s As a, p As phone, o As p From Triple Where p='phone'
 $q_3 = \text{trans}((?a, \text{cell}, ?p), \alpha, \beta) =$ Select Distinct s As a, p As cell, o As p From Triple Where p='cell'
 $q_4 = \text{trans}(((?a, \text{phone}, ?p) \text{ UNION } (?a, \text{cell}, ?p)), \alpha, \beta) =$
 Select phone,cell, r1.a As a, r1.p As p From (q_2) r1 Left Outer Join (q_3) r2 On (False)
 Union
 Select phone,cell, r3.a As a, r3.p As p From (q_3) r3 Left Outer Join (q_2) r4 On (False)
 $\text{trans}(Q_5, \alpha, \beta) =$ Select Distinct a,n,p From (
 Select Distinct name,n,phone,p,cell,Coalesce(r5.a,r6.a) As a From (q_1) r5
 Inner Join (q_4) r6 On (r5.a = r6.a Or r5.a Is Null Or r6.a Is Null)) r7

◇

In the rest of this section, we prove that the SPARQL-to-SQL translation *trans* is semantics preserving with respect to the relational algebra based semantics of SPARQL, as well as the mapping-based semantics of SPARQL. To achieve this, we first define what it means for an RDBMS-based RDF store *DB* to store an RDF graph *G*. Second, we define the semantics of *trans*-generated SQL statements as a function *exec*. Finally, we define an interpretation function ϕ to relate solutions of *eval* and *exec*, since *eval* and *exec* may produce relations with different relational attribute names due to the SQL naming constraints.

Definition 4.13 (Relational storage of an RDF graph) Given an RDBMS-based RDF store *DB*, whose scheme is represented by mappings α and β , and an RDF graph *G*, *DB* is a relational storage of *G*, denoted as DB_G , if for any triple pattern *tp*, *tp* matches the same subsets of triples in *G* and in *DB*, i.e.,

$$\forall tp, \{(t.s, t.p, t.o) \mid t \in G \wedge \text{genCond}(tp)\} \equiv \{(t.s, t.p, t.o) \mid t \in \pi_{\beta(tp, \text{sub}), \beta(tp, \text{pre}), \beta(tp, \text{obj})}(\alpha(tp)) \wedge \text{genCond}(tp)\}^2.$$

◇

Let *exec* denote a function that defines the relational algebra based semantics of *trans*-generated SQL statements. *exec* is formally defined in Appendix A. To relate a solution produced by *exec*, e.g., $R_1 = \text{exec}(\text{trans}(\text{sparql}, \alpha, \beta), DB_G)$, to a solution produced by *eval*, e.g., $R_2 = \text{eval}(\text{sparql}, G)$, we define an interpretation function ϕ as follows.

²Note that although α and β identify a relation and its attributes, the relational instance is a part of *DB* which is implicit in this equation.

Definition 4.14 (Interpretation function ϕ) Given a relation R_1 with schema $\xi(R_1)$, interpretation function ϕ returns the relation R_2 , that is derived from R_1 by renaming its relational attributes, such that $\forall x \in \xi(R_1), name^{-1}(x) \in \xi(R_2)$ and $\forall y \in \xi(R_2), name(y) \in \xi(R_1)$, where $name$ is the renaming function defined for translation $trans$ and $name^{-1}$ is the inverse function of $name$. \diamond

In other words, ϕ renames each attribute x of an input relation into $name^{-1}(x)$, while leaving attribute values untouched, and returns this relation as a result.

We prove that the SPARQL-to-SQL translation $trans$ is semantics preserving with respect to the relational algebra based semantics of SPARQL and the mapping-based semantics of SPARQL in the following two theorems.

Theorem 4.15 *Given a SPARQL query $sparql \in \mathcal{Q}$, an RDF graph G , and a relational storage DB_G of G , whose scheme is represented by mappings α and β , the SPARQL-to-SQL translation $trans$ is semantics preserving with respect to the relational algebra based semantics of SPARQL under interpretation ϕ , i.e., $\forall sparql \in \mathcal{Q}, \phi(exec(trans(sparql, \alpha, \beta), DB_G)) \equiv eval(sparql, G)$.*

Proof: The pairwise comparison of the rules that define $eval$ (see Figure 5) and the corresponding rules that define $exec$ (see Appendix A) shows that only Rule 7, which defines the evaluation of a triple pattern, and Rule 19, which defines the execution of the SQL translation of a triple pattern, use different relational algebra expressions in their premises.

One difference is how relation $R(s, p, o)$ is computed. In Rule 7,

$$R(s, p, o) = \{(t.s, t.p, t.o) \mid t \in G \wedge genCond(tp)\}$$

and in Rule 19,

$$R(s, p, o) = \{(t.s, t.p, t.o) \mid t \in \pi_{\beta(tp, sub), \beta(tp, pre), \beta(tp, obj)}(\alpha(tp)) \wedge genCond(tp)\}.$$

However, these expressions are equivalent as follows from Definition 4.13.

The other difference is how relation R_2 is computed. In Rule 7, $R_2 = genPR(tp)$ and in Rule 19, $R_2 = genPR'(tp, name)$. $genPR'$ behaves similarly to $genPR$, except that $genPR'$ renames relational attributes using the $name$ function to ensure that the attribute names conform to SQL syntax. Since $name$ is a function, it returns the same values for the same input terms and different values for different input terms. Therefore, while relational attribute names are affected, the evaluation of relational operators in all the rules of $exec$ is not affected, resulting in equivalent sets of tuples for the corresponding rules of $exec$ and $eval$. Finally, the renaming is inverted under interpretation ϕ (see Definition 4.14).

Since all the rules in the $exec$ definition return solutions that are equivalent to the solutions of the corresponding rules of $eval$ under interpretation ϕ , it can be concluded that $\forall sparql \in \mathcal{Q}, \phi(exec(trans(sparql, \alpha, \beta), DB_G)) \equiv eval(sparql, G)$. \square

Corollary 4.16 *Given a SPARQL query $sparql \in \mathcal{Q}$, an RDF graph G , and a relational storage DB_G of G , whose scheme is represented by mappings α and β , the SPARQL-to-SQL translation $trans$ is semantics preserving with respect to the mapping-based semantics of SPARQL under interpretations λ and ϕ , i.e., $\forall sparql \in \mathcal{Q}, \lambda(\phi(exec(trans(sparql, \alpha, \beta), DB_G))) \equiv [[sparql]]_G$.*

Proof: Directly follows from Theorems 3.24 and 4.15. \square

5 Extension of $eval$ and $trans$ to support the bag semantics of a SPARQL query solution

Previously, we defined the SPARQL query solution as a set – a set of mappings for the mapping-based representation Ω (see Definition 3.5), or a set of tuples for the relational representation R (see Definition 3.7). This complies with the SPARQL semantics defined in [35]. However, the W3C SPARQL specification [5],

although adopts the ideas of [35], generalizes the SPARQL query solution as a sequence of possibly unordered mappings or a bag of mappings. In the following, we briefly explain how our defined relational algebra based semantics of SPARQL and the SPARQL-to-SQL translation can be extended to support the bag semantics of a SPARQL query solution.

The extension is fairly straightforward. All the rules defining the relational algebra based semantics of SPARQL still hold, except we interpret each relation as a bag of tuples and ensure that all relational operators preserve duplicates. Similarly, all the rules defining the SPARQL-to-SQL translation still hold, except we eliminate the SQL `Distinct` construct from the queries in Rules 13, 14, 15, and 18 and substitute the SQL `Union All` construct in Rule 16 with `Union All` to ensure that duplicate tuples are preserved.

We show how a sample SPARQL query is evaluated and translated with *eval* and *trans* under the bag semantics.

Example 5.1 (eval and trans under the bag semantics)

In this example, we use SPARQL query Q_5 whose evaluation and translation under the set semantics were presented in Example 3.23 and Example 4.12, respectively.

The evaluation of Q_5 under the bag semantics over the RDF graph G in Figure 2 is as follows.

$$\begin{aligned}
Q_5: & \text{SELECT } ?a, ?n, ?p \text{ WHERE } ((?a, \text{name}, ?n) \text{ AND } ((?a, \text{phone}, ?p) \text{ UNION } (?a, \text{cell}, ?p))). \\
R_1 = \text{eval}((?a, \text{name}, ?n), G) &= \{(B_1, \text{name}, \text{paul}), (B_2, \text{name}, \text{john}), (B_3, \text{name}, \text{george}), (B_4, \text{name}, \text{ringo})\} \\
R_2 = \text{eval}((?a, \text{phone}, ?p), G) &= \{(B_1, \text{phone}, 111 - 1111), (B_4, \text{phone}, 444 - 4444)\} \\
R_3 = \text{eval}((?a, \text{cell}, ?p), G) &= \{(B_4, \text{phone}, 444 - 4444)\} \\
R_4 = \text{eval}((?a, \text{phone}, ?p) \text{ UNION } (?a, \text{cell}, ?p), G) &= R_2 \uplus R_3 = \\
& \{(B_1, \text{phone}, 111 - 1111, \text{NULL}), (B_4, \text{phone}, 444 - 4444, \text{NULL}), (B_4, \text{NULL}, 444 - 4444, \text{cell})\} \\
R_5 = \dagger_{(R_1, ?a, R_4, ?a) \rightarrow ?a} (R_1 \bowtie_{(R_1, ?a = R_4, ?a \vee R_1, ?a = \text{NULL} \vee R_4, ?a = \text{NULL})} R_4) &=
\end{aligned}$$

?a	name	?n	phone	?p	cell
B ₁	name	paul	phone	111 - 1111	NULL
B ₄	name	ringo	phone	444 - 4444	NULL
B ₄	name	ringo	NULL	444 - 4444	cell

$$\text{eval}(Q_5, G) = \pi_{?a, ?n, ?p}(R_5) =$$

?a	?n	?p
B ₁	paul	111 - 1111
B ₄	ringo	444 - 4444
B ₄	ringo	444 - 4444

Given an RDBMS-based RDF store scheme, i.e., for any triple pattern tp , $\alpha(tp) = \text{Triple}$, $\beta(tp, \text{sub}) = \mathbf{s}$, $\beta(tp, \text{pre}) = \mathbf{p}$, and $\beta(tp, \text{obj}) = \mathbf{o}$, the translation of Q_5 under the bag semantics is as follows.

$$\begin{aligned}
Q_5: & \text{SELECT } ?a, ?n, ?p \text{ WHERE } ((?a, \text{name}, ?n) \text{ AND } ((?a, \text{phone}, ?p) \text{ UNION } (?a, \text{cell}, ?p))). \\
q_1 = \text{trans}((?a, \text{name}, ?n), \alpha, \beta) &= \text{Select } \mathbf{s} \text{ As } a, \mathbf{p} \text{ As } \text{name}, \mathbf{o} \text{ As } n \text{ From Triple Where } p = \text{'name'} \\
q_2 = \text{trans}((?a, \text{phone}, ?p), \alpha, \beta) &= \text{Select } \mathbf{s} \text{ As } a, \mathbf{p} \text{ As } \text{phone}, \mathbf{o} \text{ As } p \text{ From Triple Where } p = \text{'phone'} \\
q_3 = \text{trans}((?a, \text{cell}, ?p), \alpha, \beta) &= \text{Select } \mathbf{s} \text{ As } a, \mathbf{p} \text{ As } \text{cell}, \mathbf{o} \text{ As } p \text{ From Triple Where } p = \text{'cell'} \\
q_4 = \text{trans}(((?a, \text{phone}, ?p) \text{ UNION } (?a, \text{cell}, ?p)), \alpha, \beta) &= \\
\text{Select } \text{phone, cell}, r1.a \text{ As } a, r1.p \text{ As } p \text{ From } (q_2) \text{ r1 Left Outer Join } (q_3) \text{ r2 On } (\text{False}) \\
\text{Union All} \\
\text{Select } \text{phone, cell}, r3.a \text{ As } a, r3.p \text{ As } p \text{ From } (q_3) \text{ r3 Left Outer Join } (q_2) \text{ r4 On } (\text{False}) \\
\text{trans}(Q_5, \alpha, \beta) &= \text{Select } a, n, p \text{ From } (\\
\text{Select } \text{name, n, phone, p, cell, Coalesce}(r5.a, r6.a) \text{ As } a \text{ From } (q_1) \text{ r5} \\
\text{Inner Join } (q_4) \text{ r6 On } (r5.a = r6.a \text{ Or } r5.a \text{ Is Null Or } r6.a \text{ Is Null}) \text{) r7}
\end{aligned}$$

◇

6 Simplification of the SPARQL-to-SQL Translation

Our proposed SPARQL-to-SQL translation closely resembles the definition rules of the relational algebra based semantics of SPARQL, which makes it straightforward to show that the translation is correct or

semantics preserving. However, while the semantics definition does not concern efficiency, the translation does. In this section, we present our research results on the simplification of the original SPARQL-to-SQL translation *trans* to generate simpler and more efficient SQL queries.

The following are six important simplifications that we pursue in this paper.

Simplification 1. Our first observation is that, in *trans*-generated SQL statements, the projection of relational attributes that correspond to URIs and literals in graph patterns is frequently redundant, since such attributes do not affect the SQL evaluation. In particular, such attributes, if any, are first projected in Rule 13. In Rules 14 and 15, these attributes are also projected, however they do not affect the join conditions, i.e., the expressions with such attributes always evaluate to *true*. In Rules 16 and 17, the attributes are projected, but do not participate in the join and selection conditions, respectively. Finally, in Rule 18, such attributes are eliminated from the final query solution. Projecting unnecessary URI and literal attributes in intermediate relations brings extra space and computation overhead. Therefore, our first simplification is to project only those relational attributes that store variable bindings; if a relation has no such attributes, it is sufficient and necessary to project any one of the available attributes, since the SQL `Select` projection list must contain at least one attribute. The application of this simplification to the projection lists of Rules 13-16 (Rules 17 and 18 stay the same) is straightforward. However, since the modified *trans* does not project all the attributes that correspond to graph pattern terms, the *terms* function must be redefined to return a correct set of elements. A new function *terms* must return a set of terms in a graph pattern *gp*, such that for all $x \in \text{terms}(gp)$, $\text{name}(x) \in \xi(\text{trans}(gp, \alpha, \beta))$ and for all $y \in \xi(\text{trans}(gp, \alpha, \beta))$, there exist $x \in \text{terms}(gp)$ and $\text{name}(x) = y$. This new function depends on the translation itself, i.e., the elements of *terms*(*gp*) correspond to the elements of $\xi(\text{trans}(gp, \alpha, \beta))$, which ensures that the modification of projection lists in *trans* implicitly “modifies” the result of *terms* to contain only those elements $x \in \text{terms}(gp)$ whose corresponding relational attributes has been projected $\text{name}(x) \in \xi(\text{trans}(gp, \alpha, \beta))$.

Simplification 2. Our second observation is related to the projection expression `Coalesce($r_1.\text{name}(c)$, $r_2.\text{name}(c)$) As $\text{name}(c)$` in Rules 14 and 15, where r_1 corresponds to $(\text{trans}(gp_1, \alpha, \beta))$, r_2 corresponds to $(\text{trans}(gp_2, \alpha, \beta))$, and $c \in (\text{terms}(gp_1) \cap \text{terms}(gp_2))$. Note that, if $(\text{trans}(gp_1, \alpha, \beta))$ contains no left outer joins, $r_1.\text{name}(c)$ cannot have a NULL value and therefore, is always projected by the `Coalesce` function. Therefore, the second simplification is to replace the original expression with $r_1.\text{name}(c)$ As $\text{name}(c)$ when $(\text{trans}(gp_1, \alpha, \beta))$ contains no left outer joins.

Simplification 3. The third simplification is related to the join condition `($r_1.\text{name}(c)=r_2.\text{name}(c)$ Or $r_1.\text{name}(c)$ Is Null Or $r_2.\text{name}(c)$ Is Null)` in Rules 14 and 15, where r_1 corresponds to $(\text{trans}(gp_1, \alpha, \beta))$, r_2 corresponds to $(\text{trans}(gp_2, \alpha, \beta))$, and $c \in (\text{terms}(gp_1) \cap \text{terms}(gp_2))$. This expression can sometimes be replaced with a simpler one as follows:

(i) `True`, if c is a URI or a literal. A URI or literal attribute $\text{name}(c)$ can be either “unbound” (NULL) or “bound” to itself (to c). Therefore, if $r_1.\text{name}(c)$ or $r_2.\text{name}(c)$ is NULL, then the original expression is *true*; if both $r_1.\text{name}(c)$ and $r_2.\text{name}(c)$ are not NULLs, then $r_1.\text{name}(c) = c$, $r_2.\text{name}(c) = c$, and the original expression is *true*. Since the expression always evaluates to *true*, it can be replaced with `True`.

(ii) `($r_1.\text{name}(c)=r_2.\text{name}(c)$ Or $r_2.\text{name}(c)$ Is Null)`, if $\text{trans}(gp_1, \alpha, \beta)$ contains no left outer joins.

(iii) `($r_1.\text{name}(c)=r_2.\text{name}(c)$ Or $r_1.\text{name}(c)$ Is Null)`, if $\text{trans}(gp_2, \alpha, \beta)$ contains no left outer joins.

(iv) `($r_1.\text{name}(c)=r_2.\text{name}(c)$)`, if both $\text{trans}(gp_1, \alpha, \beta)$ and $\text{trans}(gp_2, \alpha, \beta)$ contain no left outer joins.

Expressions in (ii), (iii), and (iv) are valid simplifications that are based on the following observation. When the corresponding graph pattern translation (e.g., $\text{trans}(gp_1, \alpha, \beta)$) contains no left outer joins, its resulting relation cannot have NULL values (e.g., relation r_1), and therefore, the `Is Null` check (e.g., $r_1.\text{name}(c)$ `Is Null`) always evaluates to *false* and does not affect the evaluation of the original expression.

Simplification 4. The fourth simplification is to rewrite predicates of the form “`True And subexpression`” generated in the SQL `Where` clause (Rule 13) and in the SQL `On` clause (Rules 14 and 15) as “*subexpression*”. Although this tautology elimination does not improve query evaluation performance substantially, it does enhance the readability of the *trans*-generated SQL statements.

Simplification 5. The fifth simplification is for the translation of SPARQL `UNION` in Rule 16. Note that the only purpose of the left outer joins in Rule 16 is to extend the relational schemas of $\xi(\text{trans}(gp_1, \alpha, \beta))$ and $\xi(\text{trans}(gp_2, \alpha, \beta))$ to schema $\xi(\text{trans}(gp_1, \alpha, \beta)) \cap \xi(\text{trans}(gp_2, \alpha, \beta))$. When the two relations $(\text{trans}(gp_1, \alpha, \beta))$ and $(\text{trans}(gp_2, \alpha, \beta))$ have identical schemas, the schema extension is not needed, since $\xi(\text{trans}(gp_1, \alpha, \beta)) \equiv \xi(\text{trans}(gp_2, \alpha, \beta)) \equiv \xi(\text{trans}(gp_1, \alpha, \beta)) \cap \xi(\text{trans}(gp_2, \alpha, \beta))$. Therefore, in Rule 16, left outer joins can be omitted when the relations have identical schemas, but the attribute

projection for both relations should be in the same order to ensure correct result of the SQL Union evaluation.

Simplification 6. Our last simplification is to push projection in Rule 18 into immediately contained Select subqueries of $(trans(gp, \alpha, \beta))$, such that only required variables are projected in the subqueries directly.

The implementation of these simplifications is rather straightforward. Other simplifications are also possible under some stricter conditions; we leave them for our future work.

We apply our translation with the above simplifications on sample SPARQL queries in the following example.

Example 6.1 (SPARQL-to-SQL translation with simplifications)

As before, we assume an RDBMS-based RDF store with a single relation $\text{Triple}(s, p, o)$ that stores all the RDF triples of the RDF graph described in Figure 2. Therefore, for any triple pattern tp , $\alpha(tp) = \text{Triple}$, $\beta(tp, sub) = s$, $\beta(tp, pre) = p$, and $\beta(tp, obj) = o$.

The following are sample SPARQL (same as in Example 4.12) queries and their SQL counterparts:

Q_1 : SELECT ?a, ?n, ?e, ?w WHERE (((?a, name, ?n) OPT (?a, email, ?e)) OPT (?a, web, ?w)).

$q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct s As a, o As n From Triple Where p='name'}$
 $q_2 = trans((?a, email, ?e), \alpha, \beta) = \text{Select Distinct s As a, o As e From Triple Where p='email'}$
 $q_3 = trans((?a, web, ?w), \alpha, \beta) = \text{Select Distinct s As a, o As w From Triple Where p='web'}$
 $q_4 = trans(((?a, name, ?n) OPT(?a, email, ?e)), \alpha, \beta) =$
 Select Distinct n, e, r1.a As a From (q_1) r1 Left Outer Join (q_2) r2 On (r1.a = r2.a)
 $trans(Q_1, \alpha, \beta) =$
 Select Distinct Coalesce(r3.a, r4.a) As a, n, e, w From (q_4) r3
 Left Outer Join (q_3) r4 On (r3.a = r4.a Or r3.a Is Null)

Q_2 : SELECT ?a, ?n, ?ew WHERE (((?a, name, ?n) OPT (?a, email, ?ew)) OPT (?a, web, ?ew)).

$q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct s As a, o As n From Triple Where p='name'}$
 $q_2 = trans((?a, email, ?ew), \alpha, \beta) = \text{Select Distinct s As a, o As ew From Triple Where p='email'}$
 $q_3 = trans((?a, web, ?ew), \alpha, \beta) = \text{Select Distinct s As a, o As ew From Triple Where p='web'}$
 $q_4 = trans(((?a, name, ?n) OPT(?a, email, ?ew)), \alpha, \beta) =$
 Select Distinct n, ew, r1.a As a From (q_1) r1 Left Outer Join (q_2) r2 On (r1.a = r2.a)
 $trans(Q_2, \alpha, \beta) =$
 Select Distinct Coalesce(r3.a, r4.a) As a, n, Coalesce(r3.ew, r4.ew) As ew From (q_4) r3
 Left Outer Join (q_3) r4 On ((r3.a = r4.a Or r3.a Is Null) And (r3.ew = r4.ew Or r3.ew Is Null))

Q_3 : SELECT ?a, ?n, ?e, ?w WHERE ((?a, name, ?n) OPT ((?a, email, ?e) OPT (?a, web, ?w))).

$q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct s As a, o As n From Triple Where p='name'}$
 $q_2 = trans((?a, email, ?e), \alpha, \beta) = \text{Select Distinct s As a, o As e From Triple Where p='email'}$
 $q_3 = trans((?a, web, ?w), \alpha, \beta) = \text{Select Distinct s As a, o As w From Triple Where p='web'}$
 $q_4 = trans(((?a, email, ?e) OPT(?a, web, ?w)), \alpha, \beta) =$
 Select Distinct e, w, r1.a As a From (q_2) r1 Left Outer Join (q_3) r2 On (r1.a = r2.a)
 $trans(Q_3, \alpha, \beta) =$
 Select Distinct r3.a As a, n, e, w From (q_1) r3
 Left Outer Join (q_4) r4 On (r3.a = r4.a Or r4.a Is Null)

Q_4 : SELECT ?x, ?y, ?z WHERE ((?x, name, paul) OPT ((?y, name, george) OPT (?x, email, ?z))).

$q_1 = trans((?x, name, paul), \alpha, \beta) = \text{Select Distinct s As x From Triple Where p='name' And o='paul'}$
 $q_2 = trans((?y, name, george), \alpha, \beta) = \text{Select Distinct s As y From Triple Where p='name' And o='george'}$
 $q_3 = trans((?x, email, ?z), \alpha, \beta) = \text{Select Distinct s As x, o As z From Triple Where p='email'}$
 $q_4 = trans(((?y, name, george) OPT(?x, email, ?z)), \alpha, \beta) =$
 Select Distinct y, x, z From (q_2) r1 Left Outer Join (q_3) r2 On (True)
 $trans(Q_4, \alpha, \beta) =$
 Select Distinct r3.x As x, y, z From (q_1) r3
 Left Outer Join (q_4) r4 On (r3.x = r4.x Or r4.x Is Null)

Q_5 : SELECT ?a, ?n, ?p WHERE ((?a, name, ?n) AND ((?a, phone, ?p) UNION (?a, cell, ?p))).

$q_1 = trans((?a, name, ?n), \alpha, \beta) = \text{Select Distinct s As a, o As n From Triple Where p='name'}$
 $q_2 = trans((?a, phone, ?p), \alpha, \beta) = \text{Select Distinct s As a, o As p From Triple Where p='phone'}$

```

q3 = trans((?a, cell, ?p),  $\alpha, \beta$ ) = Select Distinct s As a, o As p From Triple Where p='cell'
q4 = trans(((?a, phone, ?p) UNION (?a, cell, ?p)),  $\alpha, \beta$ ) =
Select a, p From (q2) r1 Union Select a, p From (q3) r2
trans(Q5,  $\alpha, \beta$ ) =
Select Distinct r3.a As a, p, n From (q1) r3 Inner Join (q4) r4 On (r3.a = r4.a)

```

◇

The comparison of the SQL queries generated in this example and the corresponding SQL queries generated in Example 4.12 shows that, with our proposed simplifications, *trans* generates less verbose and more efficient queries, while providing the same final result.

7 Conclusions and Future Work

RDF stores have become increasingly important to serve as metadata repositories on the Semantic Web. Recent interest in building such systems using relational database technology for storing and querying RDF data motivated us to study the problem of translating SPARQL queries into equivalent SQL queries. In our research, we first formalized the relational algebra based semantics of SPARQL that is very important to bridge the two worlds of the Semantic Web and relational databases. We proved that our defined semantics is equivalent to the mapping-based semantics of SPARQL. Second, based on the relational algebra based semantics of SPARQL, we defined the first provably semantics preserving SPARQL-to-SQL translation with support of SPARQL triple patterns, basic graph patterns, optional graph patterns, union graph patterns, and value constraints. Our translation is generic and can be implemented in existing schema-oblivious and schema-aware RDBMS-based RDF stores, including Jena, Sesame, 3store, KAON, RStar, OpenLink Virtuoso, DLDB, RDFSuite, DBOWL, PARKA, and ProvRDF. Such a flexibility was achieved by full separation of the translation from the relational database schema design. Third, we extended our semantics and translation to support the bag semantics of a SPARQL query solution. Finally, we presented a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries. Our study resulted in one of the most comprehensive SPARQL-to-SQL translations available in the literature that can serve as a reference solution for developers of RDBMS-based RDF stores.

One natural extension of our research will be an *RDFS-aware SPARQL-to-SQL translation*. Since we did not consider RDF Schema or OWL ontology in the current translation, an interesting direction is to incorporate class taxonomy, property hierarchy, and other types of ontology-based inference support into the translation. This will enable the support of the backward-chaining inference inside the relational query engine and will greatly reduce storage requirements by RDF stores with the forward-chaining inference mechanism.

Another promising direction for future work is the *translation-generated SQL query optimization*. In [13], we showed that a new relational operator enables faster SPARQL query evaluation than existing operators. Furthermore, in the current work, we observed that the translation frequently uses SQL features whose evaluation is not yet optimized by a relational database engine, e.g., multiple *coalesce* functions in one projection, null-accepting predicates, and outerunion implementations. Providing native support for these features might result in faster query evaluation.

Acknowledgements

We thank Dr. Hasan Jamil and Dr. Guizhen Yang for their thoughtful comments on this research.

References

- [1] OWL Web Ontology Language Reference. W3C Recommendation, 10 February 2004. M. Dean and G. Schreiber (Eds.). <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>.
- [2] RDF Primer. W3C Recommendation, 10 February 2004. F. Manola and E. Miller (Eds.). <http://www.w3.org/TR/rdf-primer/>.

- [3] RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, 10 February 2004. D. Brickley and R.V. Guha (Eds.). <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
- [4] Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. G. Klyne, J. J. Carroll, and B. McBride (Eds.). <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
- [5] SPARQL Query Language for RDF. W3C Candidate Recommendation, 14 June 2007. E. Prud'hommeaux and A. Seaborne (Eds.). <http://www.w3.org/TR/2007/CR-rdf-sparql-query-20070614/>.
- [6] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *Proc. of the VLDB Conference*, pages 149–158, 2001.
- [7] S. Alexaki, V. Christophides, G. Karvounarakis, and D. Plexousakis. On storing voluminous RDF descriptions: The case of Web portal catalogs. In *Proc. of the International Workshop on the Web and Databases (WebDB)*, 2001.
- [8] D. Beckett and J. Grant. SWAD-Europe Deliverable 10.2: Mapping Semantic Web Data with RDBMSes. http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/.
- [9] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.
- [10] A. Bernstein, C. Kiefer, and M. Stocker. OptARQ: A SPARQL optimization approach based on triple pattern selectivity estimation. Technical Report ifi-2007.03, March 2007. <http://www.ifi.uzh.ch/ddis/staff/goehring/btw/files/ifi-2007.03.pdf>.
- [11] C. Bizer and A. Seaborne. D2RQ - treating non-RDF databases as virtual RDF graphs. In *Proc. of the International Semantic Web Conference (ISWC)*, 2004. Poster presentation.
- [12] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 54–68, 2002.
- [13] A. Chebotko, M. Atay, S. Lu, and F. Fotouhi. Relational nested optional join for efficient Semantic Web query processing. In *Proc. of the joint conference of the Asia-Pacific Web Conference and the International Conference on Web-Age Information Management (APWeb/WAIM)*, pages 428–439, 2007.
- [14] A. Chebotko, X. Fei, S. Lu, and F. Fotouhi. Scientific workflow provenance metadata management using an RDBMS-based RDF store. Tech. Rep. TR-DB-092007-CFLF. September 2007. <http://www.cs.wayne.edu/~artem/main/research/TR-DB-092007-CFLF.pdf>.
- [15] A. Chebotko, S. Lu, H. M. Jamil, and F. Fotouhi. Semantics preserving SPARQL-to-SQL query translation for optional graph patterns. Tech. Rep. TR-DB-052006-CLJF. May 2006. <http://www.cs.wayne.edu/~artem/main/research/TR-DB-052006-CLJF.pdf>.
- [16] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proc. of the VLDB Conference*, pages 1216–1227, 2005.
- [17] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.
- [18] R. Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170. 2005. <http://www.hp1.hp.com/techreports/2005/HPL-2005-170.html>.
- [19] C. Perez de Laborda and S. Conrad. Bringing relational data into the Semantic Web using SPARQL and Relational.Owl. In *Proc. of the ICDE Workshops*, page 55, 2006.
- [20] L. Ding, K. Wilkinson, C. Sayers, and H. Kuno. Application specific schema design for storing large RDF datasets. In *Proc. of the International Workshop on Practical and Scaleable Semantic Systems (PSSS)*, 2003.

- [21] O. Erling. Implementing a SPARQL compliant RDF triple store using a SQL-ORDBMS. OpenLink Software Virtuoso. 2001. <http://virtuoso.openlinksw.com/wiki/main/Main/VOSRDFWP>.
- [22] Y. Guo, J. Heflin, and Z. Pan. Benchmarking DAML+OIL repositories. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 613–627, 2003.
- [23] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
- [24] Y. Guo, A. Qasem, Z. Pan, and J. Heflin. A requirements driven framework for benchmarking Semantic Web knowledge base systems. *IEEE Trans. Knowl. Data Eng.*, 19(2):297–309, 2007.
- [25] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *Proc. of the International Workshop on Practical and Scalable Semantic Systems (PSSS)*, pages 1–15, 2003.
- [26] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. In *Proc. of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2005.
- [27] A. Harth and S. Decker. Optimized index structures for querying RDF from the Web. In *Proc. of the Latin American Web Congress (LA-WEB)*, pages 71–80, 2005.
- [28] O. Hartig and R. Heese. The SPARQL query graph model for query optimization. In *Proc. of the European Semantic Web Conference (ESWC)*, pages 564–578, 2007.
- [29] E. Hung, Y. Deng, and V. S. Subrahmanian. RDF aggregate queries and views. In *Proc. of the ICDE Conference*, pages 717–728, 2005.
- [30] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu. RStar: an RDF storage and query system for enterprise resource management. In *Proc. of the International Conference on Information and Knowledge Management (CIKM)*, pages 484–491, 2004.
- [31] B. McBride. Jena: Implementing the RDF model and syntax specification. Hewlett Packard Laboratories. 2001. <http://www.hp1.hp.com/personal/bwm/papers/20001221-paper/>.
- [32] S. Narayanan, T. M. Kurc, and J. H. Saltz. DBOWL: Towards extensional queries on a billion statements using relational databases. Technical Report. 2006. <http://bmi.osu.edu/resources/techreports/osubmi.tr.2006.n3.pdf>.
- [33] Z. Pan and J. Heflin. DLDB: Extending relational databases to support Semantic Web queries. In *Proc. of the International Workshop on Practical and Scalable Semantic Web Systems (PSSS)*, pages 109–113, 2003.
- [34] J. Perez, M. Arenas, and C. Gutierrez. *Semantics of SPARQL*. http://ing.usalca.cl/~jperez/papers/sparql_semantics.pdf.
- [35] J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proc. of the International Semantic Web Conference (ISWC)*, 2006.
- [36] A. Polleres. From SPARQL to rules (and back). In *Proc. of the International World Wide Web Conference (WWW)*, pages 787–796, 2007.
- [37] E. Prud’hommeaux. Notes on adding SPARQL to MySQL. <http://www.w3.org/2005/05/22-SPARQL-MySQL/>.
- [38] E. Prud’hommeaux. Optimal RDF access to relational databases. <http://www.w3.org/2004/04/30-RDF-RDB-access/>.
- [39] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of RDF/S query patterns. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 607–623, 2005.

- [40] K. Stoffel, M. G. Taylor, and J. A. Hendler. Efficient management for very large ontologies. In *Proc. of the American Association for Artificial Intelligence Conference (AAAI)*, 1997.
- [41] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of RDF/S stores. In *Proc. of the International Semantic Web Conference (ISWC)*, 2005.
- [42] R. Volz, D. Oberle, B. Motik, and S. Staab. KAON SERVER - a Semantic Web management system. In *Proc. of the International World Wide Web Conference (WWW), Alternate Tracks - Practice and Experience*, 2003.
- [43] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *Proc. of the International Workshop on Semantic Web and Databases (SWDB)*, 2003.

Appendix A Semantics of *trans*-generated SQL

In this section, we define the semantics of the SQL statements generated by our translation *trans* as a function *exec*.

While we reuse previously defined operator \dagger , function *name*, and function *genCond*, we defined new function *genPR'* in Figure 8. The only difference between *genPR'* and *genPR* is the way they rename projected attributes, i.e., *genPR'* renames $s \rightarrow \text{name}(tp.sp), p \rightarrow \text{name}(tp.pp), o \rightarrow \text{name}(tp.op)$ and *genPR* renames $s \rightarrow tp.sp, p \rightarrow tp.pp, o \rightarrow tp.op$. Function *name* that is passed as a parameter to *genPR'* is the same function *name* that is used by *trans*.

```

1 Function genPR'
2 Input: triple pattern tp, function name
3 Output: relational algebra expression which projects only those attributes of relation R
4 with schema  $\xi(R) = (s, p, o)$  that correspond to distinct tp.sp, tp.pp, and tp.op
5 and renames the projected attributes as  $s \rightarrow \text{name}(tp.sp), p \rightarrow \text{name}(tp.pp), o \rightarrow \text{name}(tp.op)$ 
6 Begin
7   project-list = s
8   rename-list =  $s \rightarrow \text{name}(tp.sp)$ 
9   If tp.pp  $\neq$  tp.sp then project-list += p, rename-list =  $p \rightarrow \text{name}(tp.pp)$  End If
10  If tp.op  $\neq$  tp.sp and tp.op  $\neq$  tp.pp then project-list += o, rename-list =  $o \rightarrow \text{name}(tp.op)$  End If
11 Return  $\rho_{\text{rename-list}} \pi_{\text{project-list}}(R)$ 
12 End Function

```

Figure 8: Function *genPR'*

Given an RDBMS-based RDF store *DB*, represented by mappings α and β , *exec* is defined as follows.

$$\frac{R(s, p, o) = \{(t.s, t.p, t.o) \mid t \in \pi_{\beta(tp.sub), \beta(tp.pre), \beta(tp.obj)}(\alpha(tp)) \wedge \text{genCond}(tp)\}, R_2 = \text{genPR}'(tp, \text{name})}{\text{exec}(\text{trans}(tp, \alpha, \beta), DB) = R_2} \quad (19)$$

$$\frac{R_1 = \text{exec}(\text{trans}(gp_1, \alpha, \beta), DB), R_2 = \text{exec}(\text{trans}(gp_2, \alpha, \beta), DB), R_3 = \dagger_{[(R_1.a_i, R_2.a_i) \rightarrow a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1 \bowtie_{\wedge_{[a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1.a_i = R_2.a_i \vee R_1.a_i = \text{NULL} \vee R_2.a_i = \text{NULL})} R_2)}{\text{exec}(\text{trans}(gp_1 \text{ AND } gp_2, \alpha, \beta), DB) = R_3} \quad (20)$$

$$\frac{R_1 = \text{exec}(\text{trans}(gp_1, \alpha, \beta), DB), R_2 = \text{exec}(\text{trans}(gp_2, \alpha, \beta), DB), R_3 = \dagger_{[(R_1.a_i, R_2.a_i) \rightarrow a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1 \bowtie_{\vee_{[a_i \mid a_i \in \xi(R_1) \cap \xi(R_2)]} (R_1.a_i = R_2.a_i \vee R_1.a_i = \text{NULL} \vee R_2.a_i = \text{NULL})} R_2)}{\text{exec}(\text{trans}(gp_1 \text{ OPT } gp_2, \alpha, \beta), DB) = R_3} \quad (21)$$

$$\frac{R_1 = \text{exec}(\text{trans}(gp_1, \alpha, \beta), DB), R_2 = \text{exec}(\text{trans}(gp_2, \alpha, \beta), DB), R_3 = R_1 \uplus R_2}{\text{exec}(\text{trans}(gp_1 \text{ UNION } gp_2, \alpha, \beta), DB) = R_3} \quad (22)$$

$$\frac{R_1 = \text{exec}(\text{trans}(gp, \alpha, \beta), DB), R_2 = \{r \mid r \in R_1 \wedge \text{expr}' = \text{transexpr}(\text{expr}) \wedge \text{expr}'(r)\}}{\text{exec}(\text{trans}(gp \text{ FILTER } \text{expr}, \alpha, \beta), DB) = R_2} \quad (23)$$

$$\frac{R = \text{exec}(\text{trans}(gp, \alpha, \beta), DB),}{\text{exec}(\text{trans}(\text{SELECT } (v_1, v_2, \dots, v_n) \text{ WHERE } (gp), \alpha, \beta), DB) = \pi_{\text{name}(v_1), \text{name}(v_2), \dots, \text{name}(v_n)}(R)} \quad (24)$$

where \dagger is defined as in Figure 5.

The semantics of $\text{expr}'(r)$ is defined as follows. Given a tuple *r* of a relation *R* and expression expr' , $\text{expr}'(r) = \text{true}$, iff:

- (i) $expr'$ is *X Is Not Null*, $X \in \xi(R)$, and $r(X) \neq NULL$;
- (ii) $expr'$ is *X op 'l'*, $X \in \xi(R)$, $r(X) \neq NULL$, and $r(X) op l$, where $op \rightarrow < | \leq | \geq | > | =$;
- (iii) $expr'$ is *X op Y*, $X, Y \in \xi(R)$, $r(X) \neq NULL$, $r(Y) \neq NULL$, and $r(X) op r(Y)$;
- (iv) $expr'$ is *Not(expr'₁)* and $expr'_1(r) = false$;
- (v) $expr'$ is *(expr'₁ Or expr'₂)* and $expr'_1(r) = true$ or $expr'_2(r) = true$;
- (vi) $expr'$ is *(expr'₁ And expr'₂)*, $expr'_1(r) = true$ and $expr'_2(r) = true$.