ARTICLE IN PRESS

Data & Knowledge Engineering xxx (2009) xxx–xxx

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Semantics preserving SPARQL-to-SQL translation

Artem Chebotko [a,*], Shiyong Lu [b], Farshad Fotouhi [b]

[a] Department of Computer Science, University of Texas-Pan American, 1201 West University Drive, Edinburg, TX 78539, USA
[b] Department of Computer Science, Wayne State University, 431 State Hall, 5143 Cass Avenue, Detroit, MI 48202, USA

ARTICLE INFO

Article history:
Received 6 July 2008
Received in revised form 2 April 2009
Accepted 3 April 2009
Available online xxxx

Keywords:
SPARQL-to-SQL translation
SPARQL semantics
SPARQL
SQL
RDF
query
RDF store
RDBMS

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 the gap between SPARQL and SQL query languages, 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, alternative 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 outline a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries and extend our defined semantics and translation to support the bag semantics of a SPARQL query solution. The experimental study showed that our proposed generic translation can serve as a good alternative to existing schema dependent translations in terms of efficient query evaluation and/or ensured query result correctness.
© 2009 Elsevier B.V. All rights reserved.

1. Introduction

The Semantic Web [7,47] 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) [56,58], the standard language for annotating resources on the Web, and searched using the query language for RDF, called SPARQL [59], that has been proposed by the World Wide Web Consortium (W3C) and has recently achieved the 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 [63,62], Sesame [9], 3store [27,28], KAON [54], RStar [35], OpenLink Virtuoso [22], DLDB [38], RDFSuite [3,52], DBOWL [37], PARKA [50], RDFProv [12], and RDFBroker [48] use a relational database management system (RDBMS) as a backend to manage RDF data. The main advantage of the

* Corresponding author. Tel.: +1 956 381 2577; fax: +1 956 384 5099.
E-mail addresses: artem@cs.panam.edu (A. Chebotko), shiyong@wayne.edu (S. Lu), fotouhi@wayne.edu (F. Fotouhi).

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.04.001

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

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.

We identify three goals of SPARQL-to-SQL translation that are very important to achieve:

(1) *Correctness*. A *semantics preserving* translation is required to ensure that the semantics of a SPARQL query is equivalent to the semantics of this query translated into SQL, such that the SPARQL and SQL queries produce equivalent results.
(2) *Schema-independence*. A *generic* translation which does not depend on a particular relational database schema can be used for various database representations employed in existing RDF stores.
(3) *Efficiency*. An *efficient* translation should not only generate equivalent SQL queries quickly, but also ensure that generated queries are efficient in terms of their evaluation over a relational database.

Existing relational RDF stores implement different SPARQL-to-SQL translation algorithms based on subjective interpretations of the mapping-based semantics of SPARQL [59,39,40]. Although the mapping-based semantics of SPARQL defines a precise and concise SPARQL query evaluation mechanism, it does not support SPARQL-to-SQL translation directly. As a result, existing solutions succeed in approaching the goal of efficiency, but fail to show to be semantics preserving and/or generic. The major obstacle to the definition of a mathematically rigorous SPARQL-to-SQL translation is the gap between RDF and relational models, in general, and between SPARQL and SQL, in particular.

In this work, we define our relational algebra based semantics of SPARQL and propose the first provably semantics preserving and generic SPARQL-to-SQL translation. Furthermore, we extend the semantics and translation to support the bag semantics of a SPARQL query solution and outline our simplifications to the translation to generate simpler and more efficient SQL queries. Our main contributions are summarized in the following:

- We formalize a relational algebra based semantics of SPARQL as a function *eval*, which bridges the gap between SPARQL and SQL. We prove that *eval* is equivalent to the mapping-based semantics of SPARQL under the interpretation function[1] λ, which is used to establish the equivalence relationship[2] between two SPARQL solution representations: a relational representation and a mapping-based representation.
- We define 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 the 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 queries with triple patterns, basic graph patterns, optional graph patterns, alternative graph patterns, and value constraints. *trans* is the first provably semantics preserving translation in the literature.
- We achieve the generic property for our SPARQL-to-SQL translation *trans*, such that it supports both schema-oblivious and schema-aware database representations of existing RDBMS-based RDF stores. We do this by full separation of the translation from the relational database schema design (represented by RDF-to-Relational mappings α and β). We verify that *trans* can be implemented in at least 12 existing RDF stores, including Jena, Sesame, 3store, KAON, RStar, OpenLink Virtuoso, DLDB, RDFSuite, DBOWL, PARKA, RDFProv, and RDFBroker.
- We outline a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries, and extend *eval* and *trans* to support the bag semantics of a SPARQL query solution.
- Finally, we conduct an experimental study to explore how our generic SPARQL-to-SQL translation compares to existing schema dependent translations and how our proposed simplifications affect query performance.

The big picture of our research flow is illustrated in Fig. 1. At the data level, we define RDF-to-Relational mappings α and β, which capture how an RDF graph is stored into a relational database. At the query level, the figure illustrates the first two contributions discussed above, where the dashed arrow represents the mapping-based semantics of SPARQL defined in [39], the dotted arrows represent our contributions to the definition of relational algebra based semantics of SPARQL, and the solid arrows represent our contributions to the definition of the SPARQL-to-SQL translation. The leftmost $\Longleftrightarrow$ arrow represents the equivalence between the two semantics definitions, and the rightmost $\Longleftrightarrow$ arrow represents that the translation is semantics preserving with respect to the relational algebra based semantics of SPARQL. The third contribution, the generic goal, is achieved by full separation of the translation from the relational database schema design via the use of mappings α and β, that are first defined at the data level and later passed as parameters to the translation.

[1] Here and after, by "under the interpretation function", we mean that the function is applied to a query solution.
[2] Here and after, by "equivalence" or "equivalence relationship", we mean the mathematical equivalence between sets of elements (represent query solutions) or functions (represent query language semantics), which should be clear from the context.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

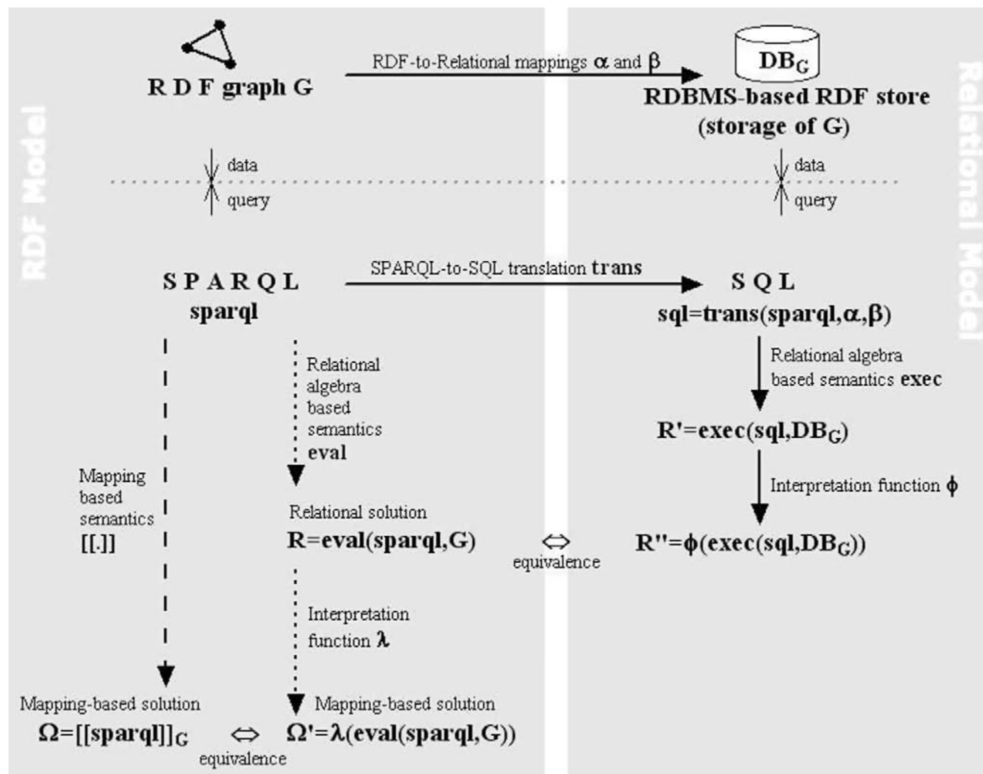

Fig. 1. The big picture of our research.

Organization. The rest of the paper is organized as follows. Section 2 reviews related work on storing and querying Semantic Web data using an RDBMS, in general, and on SPARQL-to-SQL translation, in particular. Section 3 presents preliminaries for our work. Section 4 defines our relational algebra based semantics of SPARQL. Section 5 presents our semantics preserving SPARQL-to-SQL translation. Section 6 outlines our simplifications to the translation to generate simpler and more efficient SQL queries. Section 7 deals with the extension of the semantics and translation to support the bag semantics of a SPARQL query solution. Section 8 presents our experimental study. Finally, Section 9 concludes the paper and discusses possible future work.

2. Related work

In recent years, a number of RDBMS-based RDF stores (see [6] for a survey) have been developed to support large-scale Semantic Web applications. To resolve the conflict between the graph RDF [56,58] 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 (aka query translation). 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 [57] or OWL [60] ontologies. In the following, we give more details on advances in RDF store design.

Based on database schemas employed by existing relational RDF stores, we can classify them into four categories:

Schema-oblivious (also called *generic* or *vertical*): A single relation, e.g., *Triple(s, p, o)*, is used to store RDF triples, such that attribute s stores the subject of a triple, p stores its predicate, and o stores its object. Schema-oblivious RDF stores include Jena [63,62], Sesame [9], 3store [27,28], KAON [54], RStar [35], and OpenLink Virtuoso [22]. This approach has no concerns of RDF schema or ontology evolution, since it employs a generic database representation.

Schema-aware (also called *specific* or *binary*): This approach usually employs an RDF schema or ontology to generate so called *property relations* and *class relations*. A property relation, e.g., *Property(s, o)*, is created for each property in an ontology and stores subjects s and objects o related by this property. A class relation, e.g., *Class(i)*, is created for each class in an ontology and stores instances i of this class. An extension to the idea of property relations is a *clustered property relation* [64], e.g., *Clustered*$(s, o_1, o_2, \ldots, o_n)$, which stores subjects s and objects $o_1, o_2, \ldots, o_n$ related by n distinct properties (e.g., $\langle s\ p_1\ o_1\rangle$, $\langle s\ p_2\ o_2\rangle$, etc.). In [12], along with property and class relations, we introduce *class–subject* and *class–object* relations. A

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

class–subject relation, e.g., *ClassSubject(i, p, o)*, stores triples whose subjects are instances of a particular class in an ontology. Similarly, a class–object relation, e.g., *ClassObject(s, p, i)*, stores triples whose objects are instances of a particular class. Such relations are useful for queries that retrieve all information about an instance (subject or object) of a particular class. Representatives of schema-aware RDF stores are Jena [64,63,62], DLDB [38], RDFSuite [3,52], DBOWL [37], PARKA [50], and RDFProv [11,12]. Schema evolution for this approach is quite straightforward: the addition or deletion of a class/property in an ontology requires the addition or deletion of a relation (or relational tuples) in the database. More information on ontology evolution can be found in [51] and [23]. The schema-aware approach is in general yields better query performance than the schema-oblivious approach as has been shown in several experimental studies [2,52,3,12]. In addition, the use of a column-oriented DBMS, in conjunction with vertical partitioning of relations, has shown further improvements in query performance [1].

Data-driven: This approach uses an RDF data, as opposed to an RDF schema or ontology, to generate database schema. For example, in [21], a database schema is generated based on patterns found in RDF data using data mining techniques. In general, relations generated by the schema-aware approach can also be supported by the data-driven approach (e.g., property relations in Sesame [10] are created when their instances are first seen in an RDF document during data mapping). RDF store RDFBroker [48] implements *signature relations*, which are conceptually similar to clustered property relations, but are generated based on RDF data rather than RDF Schema information. RDFBroker [48] reports improved in-memory query performance over Sesame and Jena for some test queries. Schema evolution for the data-driven approach, if supported, might be expensive.

Hybrid: This approach uses the mix of features of the previous approaches. An example of the hybrid database schema (resulted from schema-oblivious and schema-aware approaches) is presented in [52], 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*. Theoharis et al. [52] 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 [12].

Inference support techniques 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 [24]; 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 technique is bound by the main memory space required to compute inferences. The backward-chaining inference can be supported on the query mapping stage. Additional readings on inference for Semantic Web include [66,36,32,4].

One of the most difficult mappings in RDBMS-based RDF stores is the query mapping. Related literature on the SPARQL-to-SQL query translation, SPARQL query processing and optimization includes the following research works. Harris and Shadbolt [28] show how basic graph pattern expressions, as well as simple optional graph patterns, can be translated into relational algebra expressions. Cyganiak [20] presents a relational algebra for SPARQL and outlines rules establishing equivalence between this algebra and SQL. In [14], we present algorithms for basic and optional graph pattern translation into SQL. The W3C semantics of SPARQL [59] has changed since then, which was triggered by the compositional semantics presented by Perez et al. [39,40]. The new semantics defines the same evaluation results for the most common in practice SPARQL queries with so called well-designed patterns [39], 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 to accommodate graph patterns which are not well-designed.

One of the first SPARQL-to-SQL translations that is based on the new semantics is outlined by Zemke [65]. Our translation in this work, besides being derived from the relational algebra based semantics, has several distinct features when compared to the translation in [65]: (1) We prove that our translation is semantics preserving; (2) We explicitly define RDF-to-Relational mappings to make our translation generic or database schema independent; (3) We do not require SQL constructs like `With` or `Case-When-Then-Else-End` which are not supported by all relational databases; (4) We do not require to maintain the history of each subpattern solution to indicate that a constant subpattern (e.g., with no variables or blank nodes) has been matched; and (5) We provide several simplifications to the translation to generate simpler and more efficient SQL queries.

More recently, in [11,12], we define a SPARQL-to-SQL translation algorithm for basic graph pattern queries, which is optimized to select the 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 to eliminate redundancies in basic graph patterns. To improve the evaluation performance of the SPARQL optional graph patterns in a relational database, in [13,10], we propose a novel relational operator, called *nested optional join*, that shows better performance than conventional left outer join implementations.

Polleres [41] and Schenk [45] contribute with the translation of SPARQL queries into Datalog. Anyanwu et al. [5] propose an extended SPARQL query language called SPARQ2L, which supports subgraph extraction queries. Serfiotis et al. [46] 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 [30] propose a SPARQL query graph model and pursue query

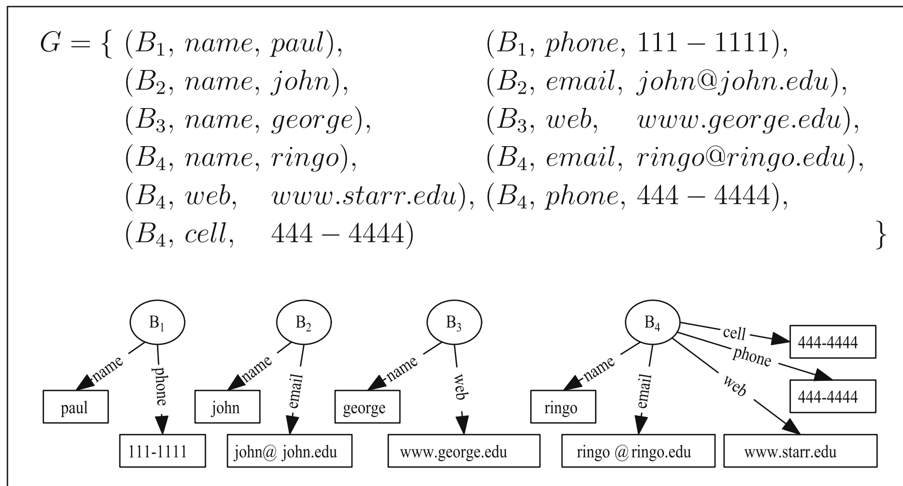

Fig. 2. Sample RDF graph.

rewriting based on this model. Stocker et al. [49] study the problem of SPARQL basic graph pattern optimization using selectivity estimation. Harth and Decker [29] propose optimized index structures for RDF that can support efficient evaluation of select-project-join queries and can be implemented in a relational database. Udrea et al. [53] propose an in-memory index structure to store RDF graph regions defined by center nodes and their associated radii; the index helps to reduce the number of joins during SPARQL query evaluation. Weiss et al. [61] introduce a sextuple-indexing scheme that can support efficient querying of RDF data based on six types of indexes, one for each possible ordering of a subject, predicate, and object. 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. [31] study the problem of RDF aggregate queries by extending an RDF query language with the GROUP BY clause and several aggregate functions. Schenk and Staab [44], Volz et al. [55], and Magkanaraki et al. [34] define RDF and SPARQL views for RDF data personalization and integration. Several research works [42,43,33,8] focus on accessing conventional relational databases using SPARQL, which requires the SPARQL-to-SQL query translation. Finally, Guo et al. [26,25] define requirements for Semantic Web knowledge base systems benchmarks and propose a framework for developing such benchmarks.

3. Preliminaries

In this section, we formalize the core fragment of SPARQL over RDF without RDFS vocabulary and literal rules, and give an overview of the mapping-based semantics [39] of SPARQL.

3.1. Syntax of SPARQL and RDF

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

Definition 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.

A sample RDF graph that we use for subsequent examples is shown in Fig. 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.

We focus on the core fragment of SPARQL defined in the following.

Definition 3.2 (*Triple pattern*). A triple pattern tp is a triple $(sp, pp, op) \in (IVL) \times (IV) \times (IVL)$, where sp,[3] pp, and op are a subject pattern, predicate pattern, and object pattern, respectively.

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

[3] Note 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 [58] and SPARQL [59] specifications does not affect our work and most likely will be resolved by W3C.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

$$gp \rightarrow tp \mid gp\ AND\ gp \mid gp\ OPT\ gp \mid gp\ UNION\ gp \mid gp\ FILTER\ expr$$

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 ($<, \leqslant, \geqslant, >$), the equality symbol ($=$), unary predicates like bound, isIRI, and other features defined in [59]. We define function $var(gp)$ to return the set of variables that appear in *gp*.

Definition 3.4 (*SPARQL query*). A SPARQL query *sparql* is defined as

$$sparql \rightarrow SELECT\ varlist\ WHERE\ (gp)$$

where $varlist = (v_1, v_2, \ldots, v_n)$ 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.

For simplicity, we do not explicitly introduce blank nodes in the triple pattern definition. Such nodes can be considered as special kinds of variables (part of *V*), so called *non-distinguished variables*, with two restricting properties [59]: (1) same blank node labels cannot be used in two different basic graph patterns in the same query, and therefore, blank nodes are variables that are always scoped to the basic graph pattern (set of triple patterns), and (2) blank node labels cannot occur in the *varlist* of the SPARQL query, and therefore, blank node bindings are not part of the query solution. Despite these syntactic constraints, blank nodes share the same semantics with regular variables and can be treated the same way. Our findings, which we present in this article, are fully applicable to blank nodes without any modification.

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 [39].

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 $\mu_\emptyset$ 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.

Example 3.6 (*Mapping-based representation of a SPARQL query solution*). Consider a graph pattern $(?a, email, ?e)$ *OPT* $(?a, web, ?w)$ that queries the RDF graph (see Fig. 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 Uniform Resource Identifiers (URIs). The graph pattern solution is represented as follows:

$$\Omega = \begin{array}{r|l|} \cline{2-2} \mu_1 : & ?a \rightarrow B_2, \quad ?e \rightarrow john@john.edu \\ \cline{2-2} \mu_2 : & ?a \rightarrow B_4, \quad ?e \rightarrow ringo@ringo.edu, \quad ?w \rightarrow www.starr.edu \\ \cline{2-2} \end{array}$$

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.

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 $\mu_\emptyset$ is compatible with any other mapping. Let Ω_1 and Ω_2 be sets of mappings. In [39], the following operators (join, union, difference, and left outer join) are defined between Ω_1 and Ω_2:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ are compatible mappings}\},$$
$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},$$
$$\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid \text{for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\},$$
$$\Omega_1 ⟕ \Omega_2 = \{(\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)\}.$$

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 Fig. 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* $(v_1, v_2, \ldots, v_n)$ *WHERE*(gp), respectively, over an RDF graph *G*. Detailed description of $[\![\cdot]\!]$ with illustrative examples is available in [39].

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

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

$$[[tp]]_G = \{\mu \mid dom(\mu) = var(tp) \text{ and } \mu(tp) \in G\}, \tag{1}$$
where $var(tp)$ is the set of variables occurring in tp and $\mu(tp)$ is the triple obtained by replacing the variables in tp according to μ.

$$[[gp_1\ AND\ gp_2]]_G = [[gp_1]]_G \bowtie [[gp_2]]_G \tag{2}$$

$$[[gp_1\ OPT\ gp_2]]_G = [[gp_1]]_G \mathbin{⟕} [[gp_2]]_G \tag{3}$$

$$[[gp_1\ UNION\ gp_2]]_G = [[gp_1]]_G \cup [[gp_2]]_G \tag{4}$$

$$[[gp\ FILTER\ expr]]_G = \{\mu \in [[gp]]_G \mid \mu \models expr\}, \tag{5}$$
where $\mu \models expr$ is defined below.

$$[[SELECT\ (v_1, v_2, ..., v_n)\ WHERE(gp)]]_G = \{\mu_{|v_1,v_2,...,v_n} \mid \mu \in [[gp]]_G\}, \tag{6}$$
where $\mu_{|v_1,v_2,...,v_n}$ is a mapping such that $dom(\mu_{|v_1,v_2,...,v_n}) = dom(\mu) \cap \{v_1, v_2, ..., v_n\}$ and $\mu_{|v_1,v_2,...,v_n}(x) = \mu(x)$ for every $x \in dom(\mu) \cap \{v_1, v_2, ..., v_n\}$.

The semantics of the $FILTER$ expression $expr$ is defined as follows. Given a mapping μ and expression $expr$, μ satisfies $expr$, denoted by $\mu \models expr$, iff:

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

NOTATION:

Symbol	Explanation	Symbol	Explanation
$[[\cdot]]$	Evaluation function	AND	Conjunction of graph patterns
G	RDF graph	OPT	Optional graph pattern
tp	Triple pattern	$UNION$	Union of graph patterns
gp	Graph pattern	$FILTER$	SPARQL selection construct
$expr$	Boolean expression	$SELECT$	Projection in a SPARQL query
μ	Mapping $V \rightarrow IBL$	$WHERE$	Pattern in a SPARQL query
dom	Domain over which μ is defined	$\bowtie$	Mapping-based join
var	Variables in tp	$⟕$	Mapping-based left outer join
$\mu_{\vert ...}$	Projection over μ's domain	$\cup$	Mapping-based union
$\models$	Mapping satisfies expression	$\cap$	Set intersection
$?X, ?Y, v$	SPARQL variables	$\neg$, $\vee$, $\wedge$	Logical NOT, OR, AND
bound	SPARQL unary predicate	$<$, $\leq$, $\geq$, $>$, $=$	Inequality/equality operators

Fig. 3. Mapping-based semantics of SPARQL.

- 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.
- The notion of empty mapping, i.e. a mapping with an 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.
- 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.
- To encode the semantics of group graph patterns and optional graph patterns, we need to consider that one variable might occur multiple times within different subpatterns 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.
- 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.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

- Finally, while evaluating value constraints, 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.

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 4.1 (*Relational representation of a SPARQL query solution*). Let a tuple $r : IVL \to IBL \cup \{\texttt{NULL}\}$ 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 $IBL \cup \{\texttt{NULL}\}$, 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 $\mathscr{R}$ as an infinite set of all possible relations, each of which represents a SPARQL query solution.

Example 4.2 (*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 Fig. 2) is represented as follows:

$R =$

$\xi(R)$:	$?a$	$email$	$?e$	web	$?w$
r_1 :	B_2	$email$	$john@john.edu$	NULL	NULL
r_2 :	B_4	$email$	$ringo@ringo.edu$	web	$www.starr.edu$

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.

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

Definition 4.3 (*Interpretation function* λ). We define interpretation function $\lambda : \mathscr{R} \to \Sigma$ as the function that takes a relation $R \in \mathscr{R}$ 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 $r(x)$ is not NULL, then $x \in dom(\mu)$ and $\mu(x) = r(x)$.

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

Example 4.4 (*Interpretation function* λ). Given the solution Ω from Example 3.6 and the solution R from Example 4.2, one can verify that $\lambda(R) \equiv \Omega$

$R =$

$?a$	$email$	$?e$	web	$?w$
B_2	$email$	$john@\ldots$	NULL	NULL
B_4	$email$	$ringo@\ldots$	web	$www.st\ldots$

$\xrightarrow{\lambda} \Omega =$

$?a \to B_2, \quad ?e \to john@\ldots$
$?a \to B_4, \quad ?e \to ringo@\ldots, \quad ?w \to www.st\ldots$

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, $⟕$ 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 4.5 (*Relational operator* $\dagger$). Given a relation R with schema $\xi(R)$, two distinct relational attributes $a, b \in \xi(R)$, and a relational attribute $c \notin \xi(R)/\{a, b\}$, the relational operator $\dagger_{(a,b)\to c}(R)$ merges attributes a and b of relation R into one single attribute c in the following way: for each tuple $r \in R$, if $r(a)$ is not NULL then $r(c) \leftarrow r(a)$, else $r(c) \leftarrow r(b)$.

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

Theorem 4.6. *Relational operator* $\dagger$ *can be derived from existing relational operators as follows:*

$$\dagger_{(a,b)\to c}(R) = \rho_{a\to c}\pi_{\xi(R)/\{b\}}(\sigma_{a \texttt{ is not NULL}}(R)) \bigcup \rho_{b\to c}\pi_{\xi(R)/\{a\}}(\sigma_{a \texttt{ is NULL}}(R)).$$

The proof of Theorem 4.6 is available in [15].

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

Example 4.7 (*Relational operator* †). Consider the following evaluation of $\dagger_{(a,b)\to c}(R)$ based on Theorem 4.6

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

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

Definition 4.8 (*Extended relational operator* †). Given a relation R with schema $\xi(R)$, n pairs $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$, where $a_1, b_1, a_2, b_2, \ldots, a_n, b_n \in \xi(R)$ are all distinct relational attributes, and n distinct relational attributes $c_1, c_2, \ldots, c_n \notin \xi(R)/\{a_1, b_1, a_2, b_2, \ldots, a_n, b_n\}$, the relational operator $\dagger_{(a_1,b_1)\to c_1,(a_2,b_2)\to c_2,\ldots,(a_n,b_n)\to c_n}(R)$ is defined recursively as:

$$\dagger_{(a_1,b_1)\to c_1,(a_2,b_2)\to c_2,\ldots,(a_n,b_n)\to c_n}(R) = \dagger_{(a_1,b_1)\to c_1}\left(\dagger_{(a_2,b_2)\to c_2,\ldots,(a_n,b_n)\to c_n}(R)\right).$$

The two auxiliary functions are defined in Fig. 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 4.9 (*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*:

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

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*.

```
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 ξ(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 ρ_rename-list π_project-list(R)
22 End Function
```

Fig. 4. Functions *genCond* and *genPR*.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

Let relation R with schema $\xi(R) = (s, p, o)$ store 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 $s \to tp.sp$, $p \to tp.pp$, and $o \to tp.op$. $R.s$ 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 4.10 (*Function genPR*). For the purpose of this example only, we extend the RDF graph G in Fig. 2 with the 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:

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

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

$R =$

s	p	o
B_2	email	john@...
B_4	email	ringo@...

evaluate $genPR(tp_1)$ $\longrightarrow$

?a	email	?e
B_2	email	john@...
B_4	email	ringo@...

$R =$

s	p	o
B_5	email	B_5

evaluate $genPR(tp_2)$ $\longrightarrow$

?a	email
B_5	email

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 Fig. 5, *eval* is defined as a set of premise-conclusion rules explained in the following.

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

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

s	p	o
B_2	email	john@john.edu
B_4	email	ringo@ringo.edu

,

$R_2 = \rho_{s \to ?a, p \to email, o \to ?e} \pi_{s,p,o}(R) =$

?a	email	?e
B_2	email	john@john.edu
B_4	email	ringo@ringo.edu

, $eval(tp_1, G) = R_2$.

Similarly, the evaluation of the triple pattern $tp_2 = (?a, web, ?w)$ over the RDF graph G in Fig. 2 results in the following:

$R =$

s	p	o
B_3	web	www.george.edu
B_4	web	www.starr.edu

, $R_2 =$

?a	web	?w
B_3	web	www.george.edu
B_4	web	www.starr.edu

, $eval(tp_2, G) = R_2$.

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 $\dagger$ 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 4.12 (*Rule 8: eval*$(gp_1$ *AND* $gp_2, G)$). Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the evaluation of the graph pattern $(tp_1$ *AND* $tp_2)$ over the RDF graph G in Fig. 2 is as follows. Let $eval(tp_1, G) = R_1$ (see Example 4.11) and $eval(tp_2, G) = R_2$ (see Example 4.11), then

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

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

$$\frac{R_1 = eval(gp_1, G),\ R_2 = eval(gp_2, G),\quad R_3 = \dagger_{[(R_1.a_i, R_2.a_i) \rightarrow a_i | a_i \in \xi(R_1) \cap \xi(R_2)]}(R_1 ⟕_{\bigwedge_{[a_i | a_i \in \xi(R_1) \cap \xi(R_2)]}(R_1.a_i = R_2.a_i \vee R_1.a_i \texttt{ is NULL} \vee R_2.a_i \texttt{ is NULL})} R_2)}{eval(gp_1\ OPT\ gp_2, G) = R_3} \quad (9)$$

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

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

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

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$(?X)$, $?X \in \xi(R)$, and $r(?X)$ is not NULL;
(ii) $expr$ is $?X\ op\ l$, $?X \in \xi(R)$, $r(?X)$ is not NULL, and $r(?X)\ op\ l$, where $op \rightarrow < | \leq | \geq | > | =$;
(iii) $expr$ is $?X\ op\ ?Y$, $?X, ?Y \in \xi(R)$, $r(?X)$ is not NULL, $r(?Y)$ is not NULL, and $r(?X)\ op\ r(?Y)$;
(iv) $expr$ is $(\neg expr_1)$ and $expr_1(r) = false$;
(v) $expr$ is $(expr_1 \vee expr_2)$ and $expr_1(r) = true$ or $expr_2(r) = true$;
(vi) $expr$ is $(expr_1 \wedge expr_2)$, $expr_1(r) = true$, and $expr_2(r) = true$.

NOTATION:

Symbol	Explanation
$eval$	Evaluation function
t	RDF triple
G	RDF graph
tp	Triple pattern
gp	Graph pattern
$expr$	Boolean expression
$?X, ?Y, v$	SPARQL variables
bound	SPARQL unary predicate
AND	Conjunction of graph patterns
OPT	Optional graph pattern
$UNION$	Union of graph patterns
$FILTER$	SPARQL selection construct
$SELECT$	Projection in a SPARQL query
$WHERE$	Pattern in a SPARQL query

Symbol	Explanation
R	Relation
r	Relational tuple
$\xi(...)$	Relational schema
a	Relational attribute
NULL	Undefined/unbound value
$genCond$, $genPR$	Auxiliary functions (Figure 4)
$\dagger$	Rel. operator (Def. 4.5, 4.8)
π	Relational projection
$\bowtie$	Relational join
⟕	Relational left outer join
$\uplus$	Relational outerunion
$\cap$	Set intersection
$\neg$, $\vee$, $\wedge$	Logical NOT, OR, AND
$<$, $\leq$, $\geq$, $>$, $=$	Inequality/equality operators

Fig. 5. Relational algebra based semantics of SPARQL.

$$eval(tp_1\ AND\ tp_2, G) = \dagger_{(R_1.?a, R_2.?a) \rightarrow ?a}(R_1 \bowtie_{R_1.?a = R_2.?a \vee R_1.?a \texttt{ is NULL} \vee R_2.?a \texttt{ is NULL}} R_2)$$

=

?a	email	?e	web	?w
B_4	email	ringo@ringo.edu	web	www.starr.edu

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 $\dagger$ 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 4.13 (*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 Fig. 2 is as follows. Let $eval(tp_1, G) = R_1$ (see Example 4.11) and $eval(tp_2, G) = R_2$ (see Example 4.11), then

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

$$eval(tp_1\ OPT\ tp_2, G) = \dagger_{(R_1.?a, R_2.?a)\to ?a}(R_1 : \bowtie_{R_1.?a=R_2.?a \vee R_1.?a \texttt{ is NULL } \vee R_2.?a \texttt{ is NULL}} R_2)$$

=

?a	email	?e	web	?w
B2	email	john@john.edu	NULL	NULL
B4	email	ringo@ringo.edu	web	www.starr.edu

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 [18]. 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 4.14 (*Rule 10:* $eval(gp_1\ UNION\ gp_2, G)$). Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the evaluation of the graph pattern $(tp_1\ UNION\ tp_2)$ over the RDF graph G in Fig. 2 is as follows. Let $eval(tp_1, G) = R_1$ and $eval(tp_2, G) = R_2$, then

$$eval(tp_1\ UNION\ tp_2, G) = R_1 \uplus R_2 =$$

?a	email	?e
B2	email	john@...
B4	email	ringo@...

$\uplus$

?a	web	?w
B3	web	www.george.edu
B4	web	www.starr.edu

=

?a	email	?e	web	?w
B2	email	john@...	NULL	NULL
B4	email	ringo@...	NULL	NULL
B3	NULL	NULL	web	www.george.edu
B4	NULL	NULL	web	www.starr.edu

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 Fig. 5.

Example 4.15 (*Rule 11:* $eval(gp\ FILTER\ expr, G)$). Given the graph pattern $gp = (?a, email, ?e)\ OPT\ (?a, web, ?w)$ and the boolean expression $expr = \neg bound(?w)$, the evaluation of the graph pattern *gp FILTER expr* over the RDF graph G in Fig. 2 is as follows. Let $eval(gp, G) = R$ (see Example 4.13), then

$$eval(gp\ FILTER\ expr, G) = \{r | r \in R \wedge \neg(r(?w)\ \texttt{is not NULL})\} =$$

?a	email	?e	web	?w
B2	email	john@john.edu	NULL	NULL

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

Example 4.16 (*Rule 12:* $eval(SELECT\ (v_1, v_2, \ldots, v_n)\ WHERE(gp), G)$). 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 Fig. 2 is as follows. Let $eval(gp, G) = R$ (see Example 4.13), then

$$eval(SELECT\ (?a, ?e, ?w)\ WHERE(gp), G) = \pi_{?a,?e,?w}(R) =$$

?a	?e	?w
B2	john@john.edu	NULL
B4	ringo@ringo.edu	www.starr.edu

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 [39]. For each SPARQL query Q_i below and the RDF graph G in Fig. 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 [39].

Example 4.17 (*Evaluation of more complex SPARQL queries*). The following are sample SPARQL queries and their evaluations over the RDF graph G in Fig. 2: The first query includes two *OPT* operators that correspond to the case of so called *sequential* OPTIONALs.

$Q_1 : SELECT\ ?a, ?n, ?e, ?w\ WHERE\ (((?a, name, ?n)\ OPT\ (?a, email, ?e))\ OPT\ (?a, web, ?w))$.
$R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$
$R_2 = eval((?a, email, ?e), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$
$R_3 = eval((?a, web, ?w), G) = \{(B_3, web, www.george.edu), (B_4, web, www.starr.edu)\}$

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

ARTICLE IN PRESS

$R_4 = eval((?a, name, ?n)\ OPT\ (?a, email, ?e), G) = \dagger_{(R_1.?a, R_2.?a) \to ?a}\ (R_1 : ⟕_{(R_1.?a = R_2.?a \vee R_1.?a \texttt{ is NULL } \vee\ R_2.?a \texttt{ is NULL })} R_2) = \{(B_1, name, paul, \texttt{NULL}, \texttt{NULL}), (B_2, name, john, email, john@john.edu), (B_3, name, george, \texttt{NULL}, \texttt{NULL}), (B_4, name, ringo, email, ringo@ringo.edu)\}$

$eval(Q_1, G) = \pi_{?a,?n,?e,?w}(\dagger_{(R_4.?a, R_3.?a) \to ?a}\ (R_4 : ⟕_{(R_4.?a = R_3.?a \vee R_4.?a \texttt{ is NULL } \vee R_3.?a \texttt{ is NULL })} R_3)) =$

?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.

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

$eval(Q_2, G) = \pi_{?a,?n,?ew}(\dagger_{(R_4.?a, R_3.?a) \to ?a, (R_4.?ew, R_3.?ew) \to ?ew}\ (R_4 : ⟕_{(R_4.?a = R_3.?a \vee R_4.?a \texttt{ is NULL} \vee R_3.?a \texttt{ is NULL}) \wedge (R_4.?ew = R_3.?ew \vee R_4.?ew \texttt{ is NULL} \vee R_3.?ew \texttt{ is NULL})} R_3)) =$

?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.

$Q_3 : SELECT\ ?a, ?n, ?e, ?w\ WHERE\ ((?a, name, ?n)\ OPT\ ((?a, email, ?e)\ OPT\ (?a, web, ?w))).$
$R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$
$R_2 = eval((?a, email, ?e), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$
$R_3 = eval((?a, web, ?w), G) = \{(B_3, web, www.george.edu), (B_4, web, www.starr.edu)\}$
$R_4 = eval((?a, email, ?e)\ OPT\ (?a, web, ?w), G) = \dagger_{(R_2.?a, R_3.?a) \to ?a}\ (R_2 : ⟕_{(R_2.?a = R_3.?a \vee R_2.?a \texttt{ is NULL } \vee\ R_3.?a \texttt{ is NULL })} R_3) = \{(B_2, email, john@john.edu, \texttt{NULL}, \texttt{NULL}), (B_4, email, ringo@ringo.edu, web, www.starr.edu)\}$

$eval(Q_3, G) = \pi_{?a,?n,?e,?w}(\dagger_{(R_1.?a, R_4.?a) \to ?a}\ (R_1 : ⟕_{(R_1.?a = R_4.?a \vee R_1.?a \texttt{ is NULL } \vee R_4.?a \texttt{ is NULL })} R_4)) =$

?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 [39], i.e. ?x occurs in both (?x, name, paul) and (?x, email, ?z), but not in the intermediate subpattern (?y, name, george).

$Q_4 : SELECT\ ?x, ?y, ?z\ WHERE\ ((?x, name, paul)\ OPT\ ((?y, name, george)\ OPT\ (?x, email, ?z))).$
$R_1 = eval((?x, name, paul), G) = \{(B_1, name, paul)\}$
$R_2 = eval((?y, name, george), G) = \{(B_3, name, george)\}$
$R_3 = eval((?x, email, ?z), G) = \{(B_2, email, john@john.edu), (B_4, email, ringo@ringo.edu)\}$
$R_4 = eval((?y, name, george)\ OPT\ (?x, email, ?z), G) = (R_2 : ⟕_{(true)} R_3) = \{(B_3, name, george, B_2, email, john@john.edu), (B_3, name, george, B_4, email, ringo@ringo.edu)\}$

$eval(Q_4, G) = \pi_{?x,?y,?z}(\dagger_{(R_1.?x, R_4.?x) \to ?x, (R_1.name, R_4.name) \to name}\ (R_1 : ⟕_{(R_1.?x = R_4.?x \vee R_1.?x \texttt{ is NULL} \vee R_4.?x \texttt{ is NULL}) \wedge (R_1.name = R_4.name \vee R_1.name \texttt{ is NULL} \vee R_4.name \texttt{ is NULL})} R_4)) =$

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

?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* $(?a, cell, ?p)))$.
$R_1 = eval((?a, name, ?n), G) = \{(B_1, name, paul), (B_2, name, john), (B_3, name, george), (B_4, name, ringo)\}$
$R_2 = eval((?a, phone, ?p), G) = \{(B_1, phone, 111-1111), (B_4, phone, 444-4444)\}$
$R_3 = eval((?a, cell, ?p), G) = \{(B_4, phone, 444-4444)\}$
$R_4 = eval((?a, phone, ?p)\ UNION\ (?a, cell, ?p), G) = R_2 \uplus R_3 = \{(B_1, phone, 111-1111, \texttt{NULL}), (B_4, phone, 444-4444, \texttt{NULL}), (B_4, \texttt{NULL}, 444-4444, cell)\}$
$R_5 = \dagger_{(R_1.?a, R_4.?a) \to ?a}\ (R_1 \bowtie_{(R_1.?a = R_4.?a \vee R_1.?a \texttt{ is NULL} \vee R_4.?a \texttt{ is NULL})} R_4) =$

?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* provides the same solutions as the mapping-based semantics under the 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 [39] under the interpretation function λ.

Theorem 4.18. *Given a SPARQL query sparql* $\in \mathcal{Q}$ *and an RDF graph G, eval is equivalent to* $[\![\cdot]\!]$ *under the interpretation* λ*, i.e.* $\lambda(eval(sparql, G)) \equiv [\![sparql]\!]_G$.

The proof of Theorem 4.18 is available in [15].

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.

5. 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 this information, we formalize an RDBMS-based RDF store scheme as the following two RDF-to-Relational mappings α and β. In this work, we study the set of schemes $\mathcal{S}$ for which both α and β are many-to-one mappings.

Definition 5.1 (*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 $\alpha : TP \to REL$, if given a triple pattern $tp \in TP$, $\alpha(tp)$ is a relation in which all the triples that may match tp are stored.

Definition 5.2 (*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 $\beta : TP \times POS \to ATR$, 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.

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

Example 5.3 (*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 Fig. 2, the relation is as follows:

$Triple =$

s	p	o
B_1	name	paul
B_1	phone	111 – 1111
B_2	name	john
B_2	email	john@john.edu
...	...	...
B_4	cell	444 – 4444

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

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.,

$P_{name} =$

s	p	o
B_1	name	paul
B_2	name	john
B_3	name	george
B_4	name	ringo

$P_{phone} =$

s	p	o
B_1	phone	111 – 1111
B_4	phone	444 – 4444

$P_{email} =$

s	p	o
B_2	email	john@...
B_4	email	ringo@...

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 $\alpha(tp) = \texttt{Triple}$; $\beta(tp, sub) = \texttt{s}$, $\beta(tp, pre) = \texttt{p}$, and $\beta(tp, obj) = \texttt{o}$.

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.,

$S_{B_1} =$

s	p	o
B_1	name	paul
B_1	phone	111 – 1111

$O_{paul} =$

s	p	o
B_1	name	paul

$S_{B_2} =$

s	p	o
B_2	name	john
B_2	email	john@...

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 $\alpha(tp) = \texttt{Triple}$; $\beta(tp, sub) = \texttt{s}$, $\beta(tp, pre) = \texttt{p}$, and $\beta(tp, obj) = \texttt{o}$.

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 several existing RDF stores, including Jena [63,62], Sesame [9], 3store [27,28], KAON [54], RStar [35], OpenLink Virtuoso [22], DLDB [38], RDFSuite [3,52], DBOWL [37], PARKA [50], RDFProv [12], and RDFBroker [48], 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 facilitates faster indexes on numeric values, however the maintenance of these mappings, as well as query processing in such a setting, are 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 [63]. 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 RDFProv.

Second, to support some other RDF 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 work, 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.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

ARTICLE IN PRESS

Finally, in some RDF stores, such as DLDB, RDFSuite, PARKA, and RDFBroker, 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 Fig. 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 5.4 (*Function genCond-SQL*). Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, email, ?a)$ and mapping β defined as $\beta(tp, sub) = \mathtt{s}$, $\beta(tp, pre) = \mathtt{p}$, and $\beta(tp, obj) = \mathtt{o}$ for any triple pattern tp, *genCond-SQL* generates the following conditions for tp_1 and tp_2 to match a tuple represented by $(\beta(tp, sub), \beta(tp, pre), \beta(tp, obj))$:

$$genCond\text{-}SQL(tp_1, \beta) = \text{“}\mathtt{True\ And}\ \beta(tp_1, pre) = \text{‘}tp_1.pp\text{’”} = \text{“}\mathtt{True\ And\ p} = \text{‘}\mathtt{email}\text{’”}.$$

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

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 $\beta(tp, sub) \to name(tp.sp)$, $\beta(tp, pre) \to name(tp.pp)$, and $\beta(tp, obj) \to name(tp.op)$. $\beta(tp, sub)$ 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.

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

15 Function genPR-SQL
16 Input: triple pattern tp, mapping β, 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 β(tp, sub) → name(tp.sp), β(tp, pre) → name(tp.pp), and β(tp, obj) → name(tp.op)
20 Begin
21     pr-list = "β(tp, sub) As name(tp.sp)"
22     If tp.pp ≠ tp.sp then pr-list += ", β(tp, pre) As name(tp.pp)" End If
23     If tp.op ≠ tp.sp and tp.op ≠ tp.pp then pr-list += ", β(tp, obj) As name(tp.op)" End If
24 Return pr-list
25 End Function
```

Fig. 6. Functions *genCond-SQL* and *genPR-SQL*.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

Example 5.5 (*Function genPR-SQL*). Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, email, ?a)$, mapping β defined as $\beta(tp, sub) = \texttt{s}$, $\beta(tp, pre) = \texttt{p}$, and $\beta(tp, obj) = \texttt{o}$ for any triple pattern tp, and function *name* (e.g., $name(?a) = \texttt{a}$, $name(?e) = \texttt{e}$, and $name(email) = \texttt{email}$), *genPR-SQL* generates the following SQL strings:

$$
\begin{aligned}
genPR\text{-}SQL(tp_1, \beta, name) &= \text{``}\beta(tp_1, sub)\ \texttt{As}\ name(tp_1.sp), \beta(tp_1, pre)\ \texttt{As}\ name(tp_1.pp),\\
&\quad\ \beta(tp_1, obj)\ \texttt{As}\ name(tp_1.op)\text{''}\\
&= \text{``}\texttt{s As a, p As email, o As e}\text{''}.\\
genPR\text{-}SQL(tp_2, \beta, name) &= \text{``}\beta(tp_2, sub)\ \texttt{As}\ name(tp_2.sp), \beta(tp_2, pre)\ \texttt{As}\ name(tp_2.pp)\text{''}\\
&= \text{``}\texttt{s As a, p As email}\text{''}.
\end{aligned}
$$

In the rest of the examples in this section, we assume that for any triple pattern tp, $\alpha(tp) = \texttt{Triple}$, $\beta(tp, sub) = \texttt{s}$, $\beta(tp, pre) = \texttt{p}$, and $\beta(tp, obj) = \texttt{o}$; function *name*, given a variable $?v \in V$ or a URI *uri*, returns strings $name(?v) = \texttt{v}$ and $name(uri) = \texttt{uri}$ 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 Fig. 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 5.6 (*Rule 13: trans*(tp, α, β)). The translation of two sample triple patterns into SQL is as follows:

$$
\begin{aligned}
trans((?a, email, ?e), \alpha, \beta) &= \texttt{Select Distinct s As a, p As email, o As e}\\
&\quad\ \texttt{From Triple Where p = 'email'}\\
trans((?a, web, ?w), \alpha, \beta) &= \texttt{Select Distinct s As a, p As web, o As w}\\
&\quad\ \texttt{From Triple Where p = 'web'}
\end{aligned}
$$

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 $rans(gp_2, \alpha, \beta)$ 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 † 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 5.7 (*Rule 14: trans*$(gp_1\ AND\ gp_2, \alpha, \beta)$). Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the translation of the graph pattern $gp = (tp_1\ AND\ tp_2)$ into SQL is as follows:

$$
\begin{aligned}
q_1 = trans((?a, email, ?e), \alpha, \beta) &= \texttt{Select Distinct s As a, p As email, o As e}\\
&\quad\ \texttt{From Triple Where p = 'email'}\\
q_2 = trans((?a, web, ?w), \alpha, \beta) &= \texttt{Select Distinct s As a, p As web, o As w}\\
&\quad\ \texttt{From Triple Where p = 'web'}\\
trans(gp, \alpha, \beta) = \texttt{Select Distinct email, e, web, w, Coalesce(r1.a, r2.a) As a From } (q_1)\ \texttt{r1}\\
\texttt{Inner Join } (q_2)\ \texttt{r2 On (r1.a = r2.a Or r1.a Is Null Or r2.a Is Null)}
\end{aligned}
$$

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

$$
\begin{aligned}
q_1 = trans((?a, email, ?e), \alpha, \beta) &= \texttt{Select Distinct s As a, p As email, o As e}\\
&\quad\ \texttt{From Triple Where p = 'email'}
\end{aligned}
$$

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

$$\texttt{Select Distinct } genPR\text{-}SQL(tp, \beta, name) \texttt{ From } \alpha(tp) \texttt{ Where } genCond\text{-}SQL(tp, \beta); \quad (13)$$

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

$$\begin{aligned}
&\texttt{Select Distinct } name(a), _{[a|a\in(terms(gp_1)-terms(gp_2))]} \ name(b), _{[b|b\in(terms(gp_2)-terms(gp_1))]}\\
&\texttt{Coalesce}(r_1.name(c), r_2.name(c)) \texttt{ As } name(c), _{[c|c\in(terms(gp_1) \ \cap \ terms(gp_2))]}\\
&\texttt{From (} trans(gp_1, \alpha, \beta) \texttt{) } r_1 \texttt{ Inner Join (} trans(gp_2, \alpha, \beta) \texttt{) } r_2 \quad (14)\\
&\texttt{On (True And}_{[c|c\in(terms(gp_1) \ \cap \ terms(gp_2))]}\\
&(r_1.name(c) = r_2.name(c) \texttt{ Or } r_1.name(c) \texttt{ Is Null Or } r_2.name(c) \texttt{ Is Null));}\\
&\text{where } r_1 = alias() \text{ and } r_2 = alias().
\end{aligned}$$

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

$$\begin{aligned}
&\texttt{Select Distinct } name(a), _{[a|a\in(terms(gp_1)-terms(gp_2))]} \ name(b), _{[b|b\in(terms(gp_2)-terms(gp_1))]}\\
&\texttt{Coalesce}(r_1.name(c), r_2.name(c)) \texttt{ As } name(c), _{[c|c\in(terms(gp_1) \ \cap \ terms(gp_2))]}\\
&\texttt{From (} trans(gp_1, \alpha, \beta) \texttt{) } r_1 \texttt{ Left Outer Join (} trans(gp_2, \alpha, \beta) \texttt{) } r_2 \quad (15)\\
&\texttt{On (True And}_{[c|c\in(terms(gp_1) \ \cap \ terms(gp_2))]}\\
&(r_1.name(c) = r_2.name(c) \texttt{ Or } r_1.name(c) \texttt{ Is Null Or } r_2.name(c) \texttt{ Is Null));}\\
&\text{where } r_1 = alias() \text{ and } r_2 = alias().
\end{aligned}$$

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

$$\begin{aligned}
&\texttt{Select } name(a)_{[a|a\in A]}, \ name(b)_{[b|b\in B]}, \ r_1.name(c)_{[c|c\in C]} \texttt{ As } name(c)\\
&\texttt{From } (trans(gp_1, \alpha, \beta)) \ r_1 \texttt{ Left Outer Join } (trans(gp_2, \alpha, \beta)) \ r_2 \texttt{ On (False)} \quad (16)\\
&\texttt{Union}\\
&\texttt{Select } name(a)_{[a|a\in A]}, \ name(b)_{[b|b\in B]}, \ r_3.name(c)_{[c|c\in C]} \texttt{ As } name(c)\\
&\texttt{From } (trans(gp_2, \alpha, \beta)) \ r_3 \texttt{ Left Outer Join } (trans(gp_1, \alpha, \beta)) \ r_4 \texttt{ On (False);}\\
&\text{where } r_1, r_2, r_3, \text{ and } r_4 = alias(); A, B, \text{ and } C \text{ are ordered sets } (terms(gp_1) - terms(gp_2)),\\
&(terms(gp_2) - terms(gp_1)), \text{ and } (terms(gp_1) \ \cap \ terms(gp_2)), \text{ respectively.}
\end{aligned}$$

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

$$\texttt{Select * From (} trans(gp, \alpha, \beta) \texttt{) } alias() \texttt{ Where } transexpr(expr); \quad (17)$$

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

$$\texttt{Select Distinct } name(v_1), name(v_2), ..., name(v_n) \texttt{ From (} trans(gp, \alpha, \beta) \texttt{) } alias(); \quad (18)$$

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`

NOTATION:

Symbol	Explanation
tp	Triple pattern
gp	Graph pattern
$expr$	Boolean expression
v	SPARQL variable
bound	SPARQL unary predicate
AND	Conjunction of graph patterns
OPT	Optional graph pattern
$UNION$	Union of graph patterns
$FILTER$	SPARQL selection construct
$SELECT$	Projection in a SPARQL query
$WHERE$	Pattern in a SPARQL query
`This font`	marks all SQL statements and constructs (`Select`, `From`, etc.)

Symbol	Explanation
$trans$	Translation function
$transexpr$	$expr$ translation procedure
α, β	RDF-to-Relational mappings
$alias$	Relation alias function
r_1, r_2	Relation aliases (tuple variables)
$terms$	Terms in a graph pattern
A, B, C	Ordered sets of terms
a, b, c	Elements in term sets
$name$	Term renaming function
$genCond\text{-}SQL$	Auxiliary function (Figure 6)
$genPR\text{-}SQL$	Auxiliary function (Figure 6)
$\cap, -$	Set intersection and difference
$\neg, \vee, \wedge$	Logical NOT, OR, AND

Fig. 7. SPARQL-to-SQL translation.

$q_2 = trans((?a, web, ?w), \alpha, \beta) =$ `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 5.9 (*Rule 16: trans*(gp_1 *UNION* gp_2, α, β)). Given triple patterns $tp_1 = (?a, email, ?e)$ and $tp_2 = (?a, web, ?w)$, the translation of the graph pattern $gp = (tp_1\ UNION\ tp_2)$ into SQL is as follows:

$q_1 = trans((?a, email, ?e), \alpha, \beta) =$ `Select Distinct s As a, p As email, o As e From Triple Where p = 'email'`

$q_2 = trans((?a, web, ?w), \alpha, \beta) =$ `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 Fig. 7.

Example 5.10 (*Rule 17: trans*(*gp FILTER expr*, α, β)). 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 FILTER expr* into SQL is as follows. Let $trans(gp, \alpha, \beta) = q$ (see Example 5.8), then

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

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

Example 5.11 (*Rule 18: trans*(*SELECT* $(v_1, v_2, \ldots, v_n)$ *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 5.8), then

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

Additionally, we present the translation of several SPARQL queries whose evaluation is described in Example 4.17.

Example 5.12 (*SPARQL-to-SQL translation*). As before, we assume an RDBMS-based RDF store with a single relation `Triple(s,p,o)` that stores all the RDF triples of the RDF graph described in Fig. 2. Therefore, for any triple pattern *tp*, $\alpha(tp) =$ `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 WHERE (((?a, name, ?n) OPT (?a, email, ?e)) OPT (?a, web, ?w)).

q_1 =trans((?a, name, ?n), α, β) =

`Select Distinct s As a, p As name, o As n From Triple Where p= 'name'`

q_2 =trans((?a, email, ?e), α, β) =

`Select Distinct s As a, p As email, o As e From Triple Where p= 'email'`

q_3 =trans((?a, web, ?w), α, β) =

```
Select Distinct s As a, p As web, o As w From Triple Where p = 'web'
```
q_4 =trans(((?a, name, ?n) OPT (?a, email, ?e)), α, β) =
```
Select Distinct name,n,email,e,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)
```
trans(Q_1, α, β) =`Select Distinct a,n,e,w From (`
```
Select Distinct name,n,email,e,web,w,Coalesce(r3.a,r4.a) As a
From (q4) r3 Left Outer Join (q3) r4 On (r3.a=r4.a Or r3.a Is Null Or r4.a Is Null)) r5
```

Q_2: SELECT ?a, ?n, ?ew WHERE (((?a, name, ?n) OPT (?a, email, ?ew)) OPT (?a, web, ?ew)).
q_1 =trans((?a, name, ?n), α, β) =
```
Select Distinct s As a, p As name, o As n From Triple Where p= 'name'
```
q_2 =trans((?a, email, ?ew), α, β) =
```
Select Distinct s As a,p As email,o As ew From Triple Where p = 'email'
```
q_3 =trans((?a, web, ?ew), α, β) =
```
Select Distinct s As a, p As web, o As ew From Triple Where p = 'web'
```
q_4 =trans(((?a, name, ?n) OPT (?a, email, ?ew)), α, β) =
```
Select Distinct name,n,email,ew,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)
```
trans(Q_2, α, β) =`Select Distinct a,n,ew From (`
```
Select Distinct name,n,email,web,Coalesce(r3.a,r4.a) As a, Coalesce(r3.ew,r4.ew) As ew
From (q4) r3 Left Outer Join (q3) r4 On ((r3.a=r4.a Or r3.a Is Null Or r4.a Is Null)
And (r3.ew=r4.ew Or r3.ew Is Null Or r4.ew Is Null))) r5
```

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), α, β) =
```
Select Distinct s As a, p As name, o As n From Triple Where p= 'name'
```
q_2 =trans((?a, email, ?e), α, β) =
```
Select Distinct s As a, p As email, o As e From Triple Where p= 'email'
```
q_3 =trans((?a, web, ?w), α, β) =
```
Select Distinct s As a, p As web, o As w From Triple Where p = 'web'
```
q_4 =trans(((?a, email, ?e) OPT (?a, web, ?w)), α, β) =
```
Select Distinct email,e,web,w,Coalesce(r1.a,r2.a) As a
From (q2) r1 Left Outer Join (q3) r2 On (r1.a=r2.a Or r1.a Is Null Or r2.a Is Null)
```
trans(Q_3, α, β) =`Select Distinct a,n,e,w From (`
```
Select Distinct name,n,email,e,web,w,Coalesce(r3.a,r4.a) As a
From (q1) r3 Left Outer Join (q4) 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 =trans((?x, name, paul), α, β) =
```
Select Distinct s As x, p As name, o As paul From Triple Where p= 'name' And o= 'paul'
```
q_2 =trans((?y, name, george),α, β) =
```
Select Distinct s As y, p As name, o As george From Triple Where p= 'name' And o= 'george'
```
q_3 =trans((?x, email, ?z),α, β) =
```
Select Distinct s As x, p As email, o As z From Triple Where p= 'email'
```
q_4 =trans(((?y, name, george) OPT (?x, email, ?z)),α, β) =
```
Select Distinct y,name,george,x,email,z From (q2) r1 Left Outer Join (q3) r2 On (True)
```
trans(Q_4, α, β) =`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 (q1) r3 Left Outer Join (q4) 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 =trans((?a, name, ?n),α, β) =
```
Select Distinct s As a, p As name, o As n From Triple Where p= 'name'
```
q_2 =trans((?a, phone, ?p),α, β) =
```
Select Distinct s As a, p As phone, o As p From Triple Where p= 'phone'
```
q_3 =trans((?a, cell, ?p),α, β) =
```
Select Distinct s As a, p As cell, o As p From Triple Where p= 'cell'
```
q_4 =trans(((?a, phone, ?p) UNION (?a, cell, ?p)),α, β) =
```
Select phone,cell, r1.a As a, r1.p As p From (q2) r1 Left Outer Join (q3) r2 On (False)
```

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

```
Union
Select phone,cell, r3.a As a, r3.p As p From (q3) r3 Left Outer Join (q2) r4 On (False)
trans(Q5,α,β) =Select Distinct a,n,p From (
Select Distinct name,n,phone,p,cell,Coalesce(r5.a,r6.a) As a From (q1) r5
Inner Join (q4) 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 5.13 (*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.[4]

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

Let *exec* denote a function that defines the relational algebra based semantics of *trans*-generated SQL statements. *exec* is formally defined in [15]. To relate a solution produced by *exec*, e.g., $R_1 = exec(trans(sparql, \alpha, \beta), DB_G)$, to a solution produced by *eval*, e.g., $R_2 = eval(sparql, G)$, we define an interpretation function ϕ as follows.

Definition 5.14 (*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*.

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 theorem and corollary.

Theorem 5.15. *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 the interpretation* ϕ, *i.e.* $\forall sparql \in \mathcal{Q},\ \phi(exec(trans(sparql, \alpha, \beta), DB_G)) \equiv eval(sparql, G)$.

The proof of Theorem 5.15 is available in [15].

Corollary 5.16. *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 the interpretations* λ *and* ϕ, *i.e.* $\forall sparql \in \mathcal{Q},\ \lambda(\phi(exec(trans(sparql, \alpha, \beta), DB_G))) \equiv [\![sparql]\!]_G$.

The proof of Corollary 5.16 directly follows from Theorems 4.18 and 5.15.

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.

6.1. 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

[4] Note that although $\alpha$and β identify a relation and its attributes, the relational instance is a part of *DB* which is implicit in this equation.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

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 $x \in terms(gp)$, $name(x) \in \xi(trans(gp, \alpha, \beta))$ and for all $y \in \xi(trans(gp, \alpha, \beta))$, there exist $x \in terms(gp)$ and $name(x) = y$. This new function depends on the translation itself, i.e. the elements of $terms(gp)$ correspond to the elements of $\xi(trans(gp, \alpha, \beta))$, which ensures that the modification of projection lists in *trans* implicitly "modifies" the result of *terms* to contain only those elements $x \in terms(gp)$ whose corresponding relational attributes have been projected $name(x) \in \xi(trans(gp, \alpha, \beta))$.

6.2. Simplification 2

Our second observation is related to the projection expression `Coalesce(`$r_1.name(c), r_2.name(c)$`) As` $name(c)$ in Rules 14 and 15, where r_1 corresponds to $(trans(gp_1, \alpha, \beta))$, r_2 corresponds to $(trans(gp_2, \alpha, \beta))$, and $c \in (terms(gp_1) \cap terms(gp_2))$. Note that, if $(trans(gp_1, \alpha, \beta))$ contains no left outer joins, $r_1.name(c)$ 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 $r_1.name(c)$ `As` $name(c)$ when $(trans(gp_1, \alpha, \beta))$ contains no left outer joins.

6.3. Simplification 3

The third simplification is related to the join condition `(`$r_1.name(c)$`=`$r_2.name(c)$`Or` $r_1.name(c)$`Is Null Or` $r_2.name(c)$`Is Null)` in Rules 14 and 15, where r_1 corresponds to $(trans(gp_1, \alpha, \beta))$, r_2 corresponds to $(trans(gp_2, \alpha, \beta))$, and $c \in (terms(gp_1) \cap terms(gp_2))$. 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 $name(c)$ can be either "unbound" (`NULL`) or "bound" to itself (to *c*). Therefore, if $r_1.name(c)$ or $r_2.name(c)$ is `NULL`, then the original expression is *true*; if both $r_1.name(c)$ and $r_2.name(c)$ are not `NULL`s, then $r_1.name(c) = c$, $r_2.name(c) = c$, and the original expression is *true*. Since the expression always evaluates to *true*, it can be replaced with `True`.
(ii) `(`$r_1.name(c) = r_2.name(c)$ `Or` $r_2.name(c)$ `Is Null)`, if $trans(gp_1, \alpha, \beta)$ contains no left outer joins.
(iii) `(`$r_1.name(c) = r_2.name(c)$ `Or` $r_1.name(c)$ `Is Null)`, if $trans(gp_2, \alpha, \beta)$ contains no left outer joins.
(iv) `(`$r_1.name(c) = r_2.name(c)$`)`, if both $trans(gp_1, \alpha, \beta)$ and $trans(gp_2, \alpha, \beta)$ 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., $trans(gp_1, \alpha, \beta)$) 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., $r_1.name(c)$ `Is Null`) always evaluates to *false* and does not affect the evaluation of the original expression.

6.4. 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.

6.5. 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 $\xi(trans(gp_1, \alpha, \beta))$ and $\xi(trans(gp_2, \alpha, \beta))$ to schema $\xi(trans(gp_1, \alpha, \beta)) \cup \xi(trans(gp_2, \alpha, \beta))$. When the two relations $(trans(gp_1, \alpha, \beta))$ and $(trans(gp_2, \alpha, \beta))$ have identical schemas, the schema extension is not needed, since $\xi(trans(gp_1, \alpha, \beta)) \equiv \xi(trans(gp_2, \alpha, \beta)) \equiv \xi(trans(gp_1, \alpha, \beta)) \cup \xi(trans(gp_2, \alpha, \beta))$. 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.

6.6. 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 to our sample SPARQL queries in the following example.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

Example 6.1 (*SPARQL-to-SQL translation with simplifications*). As before, we assume an RDBMS-based RDF store with a single relation `Triple(s,p,o)` that stores all the RDF triples of the RDF graph described in Fig. 2. Therefore, for any triple pattern tp, $\alpha(tp) = \texttt{Triple}$, $\beta(tp, sub) = \texttt{s}$, $\beta(tp, pre) = \texttt{p}$, and $\beta(tp, obj) = \texttt{o}$.

The following are sample SPARQL queries (same as in Example 5.12) 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),α, β) = `Select Distinct s As a, o As n From Triple Where p= 'name'`
q_2 =trans((?a, email, ?e),α, β) = `Select Distinct s As a, o As e From Triple Where p= 'email'`
q_3 =trans((?a, web, ?w),α, β) = `Select Distinct s As a, o As w From Triple Where p= 'web'`
q_4 =trans(((?a, name, ?n) OPT (?a, email, ?e)),α, β) =
`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, α, β) =
`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),α, β) = `Select Distinct s As a, o As n From Triple Where p= 'name'`
q_2 =trans((?a, email, ?ew),α, β) = `Select Distinct s As a, o As ew From Triple Where p= 'email'`
q_3 =trans((?a, web, ?ew),α, β) = `Select Distinct s As a, o As ew From Triple Where p= 'web'`
q_4 =trans(((?a, name, ?n) OPT (?a, email, ?ew)),α, β) =
`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, α, β) =
`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),α, β) = `Select Distinct s As a, o As n From Triple Where p= 'name'`
q_2 =trans((?a, email, ?e),α, β) = `Select Distinct s As a, o As e From Triple Where p= 'email'`
q_3 =trans((?a, web, ?w),α, β) = `Select Distinct s As a, o As w From Triple Where p= 'web'`
q_4 =trans(((?a, email, ?e) OPT (?a, web, ?w)),α, β) =
`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, α, β) =
`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),α, β) =`Select Distinct s As x From Triple Where p= 'name' And o= 'paul'`
q_2 =trans((?y, name, george),α, β) =
`Select Distinct s As y From Triple Where p= 'name' And o= 'george'`
q_3 =trans((?x, email, ?z),α, β) = `Select Distinct s As x, o As z From Triple Where p= 'email'`
q_4 =trans(((?y, name, george) OPT (?x, email, ?z)),α, β) =
`Select Distinct y, x, z From (`q_2`) r1 Left Outer Join (`q_3`) r2 On (True)`
trans(Q_4, α, β) =
`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),α, β) = `Select Distinct s As a, o As n From Triple Where p= 'name'`
q_2 =trans((?a, phone, ?p),α, β) = `Select Distinct s As a, o As p From Triple Where p= 'phone'`
q_3 =trans((?a, cell, ?p),α, β) = `Select Distinct s As a, o As p From Triple Where p= 'cell'`
q_4 =trans(((?a, phone, ?p) UNION (?a, cell, ?p)),α, β) =
`Select a, p From (`q_2`) r1 Union Select a, p From (`q_3`) r2`
trans(Q_5, α, β) =
`Select Distinct r3.a As a, p, n From (`q_1`) r3 Inner Join (`q_4`) r4 On (r3.a=r4.a)`

The comparison of the SQL queries generated in this example and the corresponding SQL queries generated in Example 5.12 shows that, with our proposed simplifications, *trans* generates less verbose and more efficient queries, while providing the same final result.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

7. Extension of the semantics and translation 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 4.1). This complies with the SPARQL semantics definition by Perez et al. [39,40] and the relational algebra definition by Codd [17,19]. However, the W3C SPARQL specification [59], although it adopts the ideas of [39,40], generalizes the SPARQL query solution as a sequence of possibly unordered mappings or a bag of mappings, similarly to SQL's generalization of a relation with the set semantics into a table with the bag semantics. 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` 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 7.1 (*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 Examples 4.17 and 5.12/6.1, respectively.

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

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

?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, $\alpha(tp) = \mathtt{Triple}$, $\beta(tp, sub) = \mathtt{s}$, $\beta(tp, pre) = \mathtt{p}$, and $\beta(tp, obj) = \mathtt{o}$, the translation of Q_5 under the bag semantics with the simplifications (see Example 6.1) is as follows.

Q_5: SELECT ?a, ?n, ?p WHERE ((?a, name, ?n) AND ((?a, phone, ?p) UNION (?a, cell, ?p))).
$q_1 = trans((?a, name, ?n), \alpha, \beta) =$ `Select s As a, o As n From Triple Where p = 'name'`
$q_2 = trans((?a, phone, ?p), \alpha, \beta) =$ `Select s As a, o As p From Triple Where p = 'phone'`
$q_3 = trans((?a, cell, ?p), \alpha, \beta) =$ `Select s As a, o As p From Triple Where p = 'cell'`
$q_4 = trans(((?a, phone, ?p)\ UNION\ (?a, cell, ?p)), \alpha, \beta) =$
`Select a, p From (`q_2`) r1 Union All Select a, p From (`q_3`) r2`
$trans(Q_5, \alpha, \beta) =$
`Select r3.a As a, p, n From (`q_1`) r3 Inner Join (`q_4`) r4 On (r3.a=r4.a)`

8. Experimental study

In this section, we present our experimental study with the following two main goals:

(1) Exploring and comparing the performance of queries generated by our generic translation with the performance of queries generated by schema dependent translations implemented in existing relational RDF stores.
(2) Exploring and comparing the performance of queries generated by our original translation (*trans*) with the performance of queries generated by our simplified translation (further denoted as *trans-s*).

Towards these goals, we implemented *trans* and *trans-s* in C++ and applied them to query translation for RDF stores RDFProv [11,12], Sesame [9], and Jena [63,62] that stored RDF data in a MySQL 5.1 CE RDBMS. In addition, to test our translations over different relational query optimizers, we ran our test queries over the RDFProv store deployed with both MySQL 5.1 Community Edition and Oracle 9i Enterprise Edition.

The dataset for our experiments was obtained by extending the RDF graph in Fig. 2 to a larger one with 1,000,000 triples that captured information about persons' names, emails, phones, cell phones, and webpages. We selected nine SPARQL que-

Table 1
Evaluation times of queries Q1–Q9 over RDFProv, Sesame, and Jena (best times for each store are shown in bold).

RDF store/translation/RDBMS	Query evaluation time (s)								
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
RDFProv/*trans*/MySQL	14.73	16.63	13.81	0.05	19.92	0.03	0.03	5.89	5.73
RDFProv/*trans-s*/MySQL	5.233	4.217	**4.28**	0.02	0.08	**0.01**	**0.01**	0.198	0.05
RDFProv/*trans*/Oracle	16.06	17.05	100.2	**0.01**	**0.01**	**0.01**	**0.01**	4.04	**0.01**
RDFProv/*trans-s*/Oracle	**5.05**	**4.06**	8.03	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**
Sesame/Sesame/MySQL	**5.07**	**4.31**	**4.78**	**0.08**	4.29	**0.18**	**0.17**	**0.18**	**0.17**
Sesame/*trans*/MySQL	1371	1380	1178	2.14	1083	1.38	1.27	346.7	651.9
Sesame/*trans-s*/MySQL	618.1	635	324	1.8	**3.39**	1.24	1.19	309.2	2.65
Jena/Jena/MySQL	2357	1043	2258	0.36	1064	0.594	0.562	2.859	0.563
Jena/*trans*/MySQL	117.1	118.6	42.69	0.13	132.4	0.2	0.2	2.825	30.42
Jena/*trans-s*/MySQL	**44.4**	**44.5**	**22.2**	**0.11**	**0.25**	**0.16**	**0.16**	**2.2**	**0.22**

ries over this dataset. Queries Q1–Q5 were the ones presented in Examples 4.17, 5.12, and 6.1. Since queries Q1, Q2, Q3, and Q5 yielded large intermediate and final results ($\approx 25\%$ of the dataset size) and could not benefit from database indexes, we modified them into queries Q6, Q7, Q8, and Q9, respectively, by replacing the variable $?n$ with the literal "george". Queries Q6–Q9, as well as Q4, appeared to be much more selective and efficient.

In Table 1, we report the performance of our test queries over the generated dataset stored with RDFProv, Sesame 2.2.3, and Jena 2.5.7. The experiments were conducted on a PC with 3.00 GHz Intel Core 2 CPU, 4 GB RAM, and 750 GB disk space running MS Windows XP Professional.

Our system RDFProv used several types of relations to store RDF data, including class and property relations, and provided RDF-to-Relational mappings α and β that were generated for each query using algorithms described in [11,12]. Additional RDFProv optimizations for basic graph patterns were not applicable to our test queries and were turned off. Our simplifications (*trans-s*) showed to significantly improve query performance for all the queries evaluated over MySQL and queries Q1, Q2, Q3, and Q8 evaluated over Oracle. Oracle showed better performance than MySQL for most simplified queries (except Q3) and showed equal performance for some *trans* and *trans-s* generated queries (Q4–Q7 and Q9), which could be a result of more sophisticated query optimization techniques used by this database management system.

Sesame used the normalized database schema with one relation (*triples*) that stored subjects, predicates, and objects of all RDF triples, however URIs and literals in this relation were substituted with integer IDs. The mappings from IDs to URIs and literals were stored in relations *uri_values* and *label_values*, respectively. To deal with this schema, we created a denormalized database view that used three inner joins ($triples \bowtie_{subj=id} uri_values$, $triples \bowtie_{pred=id} uri_values$, and $triples \bowtie_{obj=id} (uri_values \cup label_values)$) to substitute IDs with actual URIs and literals. With such a view, the α and β mappings became very simple as described in the first case of Example 5.3. Sesame's native translation showed to be very efficient, however the system returned incorrect/incomplete results for queries Q2 and Q7. For example, for Q2 in Example 4.17, Sesame returned incomplete tuple $(B_3, george, \mathtt{NULL})$ instead of expected tuple $(B_3, george, www.george.edu)$ in the final result. The performance of the *trans* and *trans-s* generated queries was significantly slower than Sesame's performance for most queries, since the queries were evaluated over the view that required three additional joins; nevertheless, *trans-s* had the best time for Q5.

Jena used the denormalized database schema with one relation that was similar to the denormalized database view for Sesame. Before evaluating the *trans* and *trans-s* generated queries over this store, we had to encode URIs and literals using Jena's encoding scheme (e.g., the "*george*" literal was encoded as "*Lv:0::george:*"). We observed that the *trans-s* queries outperformed both *trans* and Jena's native translation queries in all the tests. *trans* showed to be a faster alternative than Jena's translation for all the queries except Q9.

With respect to the experimental study goals and based on our empirical data, we conclude:

(1) Our generic translation can serve as a good alternative to existing schema dependent translations to provide better query performance (as in case of Jena) or ensure query result correctness (as in case of Sesame).
(1) Our proposed simplifications to the translation can significantly improve query performance (in case of both MySQL and Oracle).

9. Conclusions and future work

In this work, 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 queries with triple patterns, basic graph patterns, optional graph patterns, alternative graph patterns, and value constraints. Our translation is generic and can be implemented in existing relational RDF stores, including Jena, Sesame, 3store, KAON, RStar, OpenLink Virtuoso, DLDB, RDFSuite, DBOWL, PARKA, RDFProv, and RDFBroker. Such a flexibility was achieved by full separation of the translation from the relational data-

base schema design. Third, we presented a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries. Fourth, we extended our semantics and translation to support the bag semantics of a SPARQL query solution. Finally, we conducted the experimental study showing that: (1) our generic translation can serve as a good alternative to existing schema dependent translations to provide better query performance and/or ensure query result correctness, and (2) our proposed simplifications to the translation can significantly improve query performance. Our work resulted in the first solution to the problem of SPARQL-to-SQL translation that has been shown to be correct. It can serve as a reference solution for researchers and developers of relational RDF stores.

We identify the following directions for future work:

- *RDFS-aware SPARQL-to-SQL translation* will be a natural extension of our research. 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 technique.
- *Translation-generated SQL query optimization* is another promising direction for future work. 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.
- *New research opportunities*, such as integration, reuse, and evaluation of RDF store schemas, automatic checking of query correctness, and distributed querying of multiple RDF stores, are becoming feasible with the generic and semantics-preserving SPARQL-to-SQL translation. We are interested in exploring some of these important challenges.

References

[1] D.J. Abadi, A. Marcus, S. Madden, K.J. Hollenbach, Scalable Semantic Web data management using vertical partitioning, in: Proc. of the International Conference on Very Large Data Bases (VLDB), 2007, pp. 411–422.
[2] R. Agrawal, A. Somani, Y. Xu, Storage and querying of e-commerce data, in: Proc. of the International Conference on Very Large Data Bases (VLDB), 2001, pp. 149–158.
[3] S. Alexaki, V. Christophides, G. Karvounarakis, 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, pp. 43–48.
[4] G. Antoniou, A. Bikakis, N. Dimaresis, M. Genetzakis, G. Georgalis, G. Governatori, E. Karouzaki, N. Kazepis, D. Kosmadakis, M. Kritsotakis, G. Lilis, A. Papadogiannakis, P. Pediaditis, C. Terzakis, R. Theodosaki, D. Zeginis, Proof explanation for a nonmonotonic Semantic Web rules language, Data Knowledge Eng. 64 (3) (2008) 662–687.
[5] K. Anyanwu, A. Maduko, A. Sheth, SPARQ2L: towards support for subgraph extraction queries in RDF databases, in: Proc. of the International World Wide Web Conference (WWW), 2007, pp. 797–806.
[6] D. Beckett, J. Grant, SWAD-Europe Deliverable 10.2: Mapping Semantic Web data with RDBMSes. Technical Report, 2003. <http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report>.
[7] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American, May 2001.
[8] C. Bizer, A. Seaborne, D2RQ – treating non-RDF databases as virtual RDF graphs, in: Proc. of the International Semantic Web Conference (ISWC), 2004, Poster presentation.
[9] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: a generic architecture for storing and querying RDF and RDF Schema, in: Proc. of the International Semantic Web Conference (ISWC), 2002, pp. 54–68.
[10] A. Chebotko, M. Atay, S. Lu, 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), 2007, pp. 428–439.
[11] A. Chebotko, X. Fei, C. Lin, S. Lu, F. Fotouhi, Storing and querying scientific workflow provenance metadata using an RDBMS, in: Proc. of the IEEE International Workshop on Scientific Workflows and Business Workflow Standards in e-Science, 2007, pp. 611–618.
[12] A. Chebotko, X. Fei, S. Lu, F. Fotouhi, Scientific workflow provenance metadata management using an RDBMS-based RDF store, Technical Report TR-DB-092007-CFLF, Wayne State University, September 2007. <http://www.cs.wayne.edu/~artem/main/research/TR-DB-092007-CFLF.pdf>.
[13] A. Chebotko, S. Lu, M. Atay, F. Fotouhi, Efficient processing of RDF queries with nested optional graph patterns in an RDBMS, Int. J. Semantic Web Inform. Syst. 4 (4) (2008) 1–30.
[14] A. Chebotko, S. Lu, H.M. Jamil, F. Fotouhi, Semantics preserving SPARQL-to-SQL query translation for optional graph patterns, Technical Report TR-DB-052006-CLJF, Wayne State University, May 2006. <http://www.cs.wayne.edu/~artem/main/research/TR-DB-052006-CLJF.pdf>.
[15] A. Chebotko, Querying and Managing Semantic Web Data and Scientific Workflow Provenance using Relational Databases, Ph.D. Dissertation, Department of Computer Science, Wayne State University, USA, 2008.
[16] E.I. Chong, S. Das, G. Eadon, J. Srinivasan, An efficient SQL-based RDF querying scheme, in: Proc. of the International Conference on Very Large Data Bases (VLDB), 2005, pp. 1216–1227.
[17] E.F. Codd, A relational model of data for large shared data banks, Commun. ACM 13 (6) (1970) 377–387.
[18] E.F. Codd, Extending the database relational model to capture more meaning, ACM Trans. Database Syst. 4 (4) (1979) 397–434.
[19] E.F. Codd, The Relational Model for Database Management, Version 2, Addison-Wesley, 1990.
[20] R. Cyganiak, A relational algebra for SPARQL, Technical Report HPL-2005-170, Hewlett-Packard Laboratories, 2005. <http://www.hpl.hp.com/techreports/2005/HPL-2005-170.html>.
[21] L. Ding, K. Wilkinson, C. Sayers, 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.
[22] O. Erling, Implementing a SPARQL compliant RDF triple store using a SQL-ORDBMS. Technical Report, OpenLink Software Virtuoso, 2001. <http://virtuoso.openlinksw.com/wiki/main/Main/VOSRDFWP>.
[23] G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, G. Antoniou, Ontology change: classification and survey, Knowledge Eng. Rev. 23 (2) (2008).
[24] Y. Guo, J. Heflin, Z. Pan, Benchmarking DAML+OIL repositories, in: Proc. of the International Semantic Web Conference (ISWC), 2003, pp. 613–627.
[25] Y. Guo, Z. Pan, J. Heflin, LUBM: a benchmark for OWL knowledge base systems, J. Web Semant. 3 (2–3) (2005) 158–182.
[26] Y. Guo, A. Qasem, Z. Pan, J. Heflin, A requirements driven framework for benchmarking Semantic Web knowledge base systems, IEEE Trans. Knowledge Data Eng. 19 (2) (2007) 297–309.
[27] S. Harris, N. Gibbins, 3store: efficient bulk RDF storage, in: Proc. of the International Workshop on Practical and Scalable Semantic Systems (PSSS), 2003, pp. 1–15.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

[28] S. Harris, 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, pp. 235–244.
[29] A. Harth, S. Decker, Optimized index structures for querying RDF from the Web, in: Proc. of the Latin American Web Congress (LA-WEB), 2005, pp. 71–80.
[30] O. Hartig, R. Heese, The SPARQL query graph model for query optimization, in: Proc. of the European Semantic Web Conference (ESWC), 2007, pp. 564–578.
[31] E. Hung, Y. Deng, V.S. Subrahmanian, RDF aggregate queries and views, in: Proc. of the International Conference on Data Engineering (ICDE), 2005, pp. 717–728.
[32] E. Kontopoulos, N. Bassiliades, G. Antoniou, Deploying defeasible logic rule bases for the Semantic Web, Data Knowledge Eng. 66 (1) (2008) 116–146.
[33] C.P. de Laborda, S. Conrad, Bringing relational data into the Semantic Web using SPARQL and Relational.OWL, in: Proc. of the ICDE Workshops, 2006, p. 55.
[34] A. Magkanaraki, V. Tannen, V. Christophides, D. Plexousakis, Viewing the Semantic Web through RVL lenses, J. Web Semant. 1 (4) (2004) 359–375.
[35] L. Ma, Z. Su, Y. Pan, L. Zhang, 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), 2004, pp. 484–491.
[36] D.L. McGuinness, P.P. da Silva, Explaining answers from the Semantic Web: the inference Web approach, J. Web Semant. 1 (4) (2004) 397–413.
[37] S. Narayanan, T.M. Kurc, J.H. Saltz, DBOWL: towards extensional queries on a billion statements using relational databases. Technical Report OSUBMI_TR_2006_n03, Ohio State University, 2006. <http://bmi.osu.edu/resources/techreports/osubmi.tr.2006.n3.pdf>.
[38] Z. Pan, J. Heflin, DLDB: extending relational databases to support Semantic Web queries, in: Proc. of the International Workshop on Practical and Scaleable Semantic Web Systms (PSSS), 2003, pp. 109–113.
[39] J. Perez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, in: Proc. of the International Semantic Web Conference (ISWC), 2006, pp. 30–43.
[40] J. Perez, M. Arenas, C. Gutierrez, Semantics of SPARQL, Technical Report, 2006. <http://ing.utalca.cl/~jperez/papers/sparql_semantics.pdf>.
[41] A. Polleres, From SPARQL to rules (and back), in: Proc. of the International World Wide Web Conference (WWW), 2007, pp. 787–796.
[42] E. Prud'hommeaux, Optimal RDF access to relational databases, Technical Report, 2004. <http://www.w3.org/2004/04/30-RDF-RDB-access/>.
[43] E. Prud'hommeaux, Notes on adding SPARQL to MySQL, Technical Report, 2005. <http://www.w3.org/2005/05/22-SPARQL-MySQL/>.
[44] S. Schenk, S. Staab, Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the Web, in Proc. of the International World Wide Web Conference (WWW), 2008, pp. 585–594.
[45] S. Schenk, A SPARQL semantics based on Datalog, in: Proc. of the KI 2007, Annual German Conference on AI, 2007, pp. 160–174.
[46] G. Serfiotis, I. Koffina, V. Christophides, V. Tannen, Containment and minimization of RDF/S query patterns, in: Proc. of the International Semantic Web Conference (ISWC), 2005, pp. 607–623.
[47] N. Shadbolt, T. Berners-Lee, W. Hall, The Semantic Web revisited, IEEE Intell. Syst. 21 (3) (2006) 96–101.
[48] M. Sintek, M. Kiesel, RDFBroker: a signature-based high-performance RDF store, in: Proc. of the European Semantic Web Conference (ESWC), 2006, p. 363–377.
[49] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, D. Reynolds, SPARQL basic graph pattern optimization using selectivity estimation, in: Proc. of the International World Wide Web Conference (WWW), 2008, pp. 595–604.
[50] K. Stoffel, M.G. Taylor, J.A. Hendler, Efficient management of very large ontologies, in: Proc. of the American Association for Artificial Intelligence Conference (AAAI), 1997, pp. 442–447.
[51] L. Stojanovic, Methods and Tools for Ontology Evolution, Ph.D. Dissertation, University of Karlsruhe, Germany, 2004. <digbib.ubka.uni-karlsruhe.de/volltexte/documents/1241>.
[52] Y. Theoharis, V. Christophides, G. Karvounarakis, Benchmarking database representations of RDF/S stores, in: Proc. of the International Semantic Web Conference (ISWC), 2005, pp. 685–701.
[53] O. Udrea, A. Pugliese, V.S. Subrahmanian, GRIN: a graph based RDF index, in: Proc. of the American Association for Artificial Intelligence Conference (AAAI), 2007, pp. 1465–1470.
[54] R. Volz, D. Oberle, B. Motik, 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.
[55] R. Volz, D. Oberle, R. Studer, Implementing views for light-weight Web ontologies, in: Proc. of the International Database Engineering and Applications Symposium (IDEAS), 2003, pp. 160–169.
[56] W3C, RDF primer, in: F. Manola, E. Miller (Eds.), W3C Recommendation, 10 February 2004. <http://www.w3.org/TR/rdf-primer/>.
[57] W3C, RDF vocabulary description language 1.0: RDF schema, in: D. Brickley, R.V. Guha (Eds.), W3C Recommendation, 10 February 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
[58] W3C, Resource description framework (RDF): concepts and abstract syntax, in: G. Klyne, J.J. Carroll, B. McBride (Eds.), W3C Recommendation, 10 February 2004. <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
[59] W3C, SPARQL query language for RDF, in: E. Prud'hommeaux, A. Seaborne (Eds.), W3C Recommendation, 15 January 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
[60] W3C, OWL Web Ontology Language Reference, in: M. Dean, G. Schreiber (Eds.), W3C Recommendation, 10 February 2004. <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>.
[61] C. Weiss, P. Karras, A. Bernstein, Hexastore: sextuple indexing for Semantic Web data management, Proc. of the VLDB Endowment (PVLDB) 1 (1) (2008) 1008–1019.
[62] K. Wilkinson, C. Sayers, H.A. Kuno, D. Reynolds, L. Ding, Supporting scalable, persistent Semantic Web applications, IEEE Data Eng. Bull. 26 (4) (2003) 33–39.
[63] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds, Efficient RDF storage and retrieval in Jena2, in: Proc. of the International Workshop on Semantic Web and Databases (SWDB), 2003, pp. 131–150.
[64] K. Wilkinson, Jena property table implementation, in: Proc. of the International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS), 2006.
[65] F. Zemke, Converting SPARQL to SQL, Technical Report, October 2006. <http://lists.w3.org/Archives/Public/public-rdf-dawg/2006OctDec/att-0058/sparql-to-sql.pdf>.
[66] RuleML: The RuleML Initiative website. <http://www.ruleml.org/>.

Artem Chebotko received the PhD degree in computer science from Wayne State University in 2008, MS and BS degrees in management information systems and computer science from Ukraine State Maritime Technical University in 2003 and 2001. He is currently an assistant professor in the Department of Computer Science, University of Texas – Pan American. His research interests include Semantic Web data management, scientific workflow provenance, XML databases, and relational databases. He has published more than 20 papers in refereed international journals and conference proceedings. He currently serves as a program committee member of several international conferences and workshops on Semantic Web and scientific workflows. He is a member of the IEEE.

Please cite this article in press as: A. Chebotko et al., Semantics preserving SPARQL-to-SQL translation, Data Knowl. Eng. (2009), doi:10.1016/j.datak.2009.04.001

Shiyong Lu received the PhD degree from the State University of New York at Stony Brook in 2002. He is currently an assistant professor in the Department of Computer Science, Wayne State University, and the director of the Scientific Workflow Research Laboratory. His research interests include scientific workflows and databases. He has published more than 70 refereed international journal and conference papers in the above areas. He is the founder and currently a cochair of the *IEEE International Workshop on Scientific Workflows*, an editorial board member for *International Journal of Medical Information Systems and Informatics*, and for *International Journal of Semantic Web and Information Systems*. He also serves as a program committee member for several top-tier IEEE conferences including SCC and ICWS. He is a member of the IEEE.

Farshad Fotouhi received the PhD degree in computer science from Michigan State University in 1988. In August 1988, he joined the faculty of Computer Science at Wayne State University, where he is currently a professor and the chair of the department. His major areas of research include XML databases, semantic Web, multimedia systems, and query optimization. He has published more than 100 papers in refereed journals and conference proceedings, served as a program committee member of various database-related conferences. He is on the Editorial Boards of the *IEEE Multimedia Magazine* and *International Journal on Semantic Web and Information Systems*. He is a member of the IEEE.