

## UPDATE-ENABLED TRIPLIFICATION OF RELATIONAL DATA INTO VIRTUAL RDF STORES

SUNITHA RAMANUJAM\*, VAIBHAV KHADILKAR<sup>†</sup>, LATIFUR KHAN<sup>‡</sup>,  
MURAT KANTARCIOGLU<sup>§</sup> and BHAVANI THURASINGHAM<sup>¶</sup>

*The University of Texas at Dallas, Richardson, TX 75080, USA*

\*[sxr063200@utdallas.edu](mailto:sxr063200@utdallas.edu)

<sup>†</sup>[vvk072000@utdallas.edu](mailto:vvk072000@utdallas.edu)

<sup>‡</sup>[lkhan@utdallas.edu](mailto:lkhan@utdallas.edu)

<sup>§</sup>[murat@utdallas.edu](mailto:murat@utdallas.edu)

<sup>¶</sup>[bxt043000@utdallas.edu](mailto:bxt043000@utdallas.edu)

STEVEN SEIDA

*Raytheon Company, Garland, TX 75042, USA*

[steven\\_b\\_seida@raytheon.com](mailto:steven_b_seida@raytheon.com)

The current buzzword in the Internet community is the Semantic Web initiative proposed by the W3C to yield a Web that is more flexible and self-adapting. However, for the Semantic Web initiative to become a reality, heterogeneous data sources need to be integrated in order to enable access to them in a homogeneous manner. Since a vast majority of data currently resides in relational databases, integrating relational data sources with semantic web technologies is at the top of the list of activities required to realize the semantic web vision. Several efforts exist that publish relational data as Resource Description Framework (RDF) triples; however almost all current work in this arena is uni-directional, presenting data from an underlying relational database into a corresponding virtual RDF store in a read-only manner. An enhancement over previous relational-to-RDF bridging work in the form of bi-directionality support is presented in this paper. The bi-directional bridge proposed here allows RDF data updates specified as triples to be propagated back into the underlying relational database as tuples. Towards this end, we present various algorithms to translate the triples to be updated/inserted/deleted into equivalent relational attributes/tuples whenever possible. Particular emphasis is laid, in this paper, on the translation and update propagation process for triples containing blank nodes and reification nodes, and a platform enhanced with our algorithms, called D2RQ++, through which bi-directional translation can be achieved, is presented.

*Keywords:* Semantic web; resource description framework; relational databases; data interoperability.

### 1. Introduction

The idea of a semantically annotated Web that would bestow upon machines enhanced interpretative abilities and enable a perception of the Web as one large database, thereby facilitating large scale data integration and reuse is rapidly

gaining popularity. Endeavors to find a way to translate this idea into tangible reality resulted in the Semantic Web initiative which advocates resource description through the use of meta-data rather than keywords thereby enhancing computer understandability and reusability of information in web pages. In order to enable the definition and description of data (or resources) on the Web and the relations between them, several Semantic Web technologies such as Resource Description Framework (RDF), which is considered as one of the fundamental building blocks of the Semantic Web, RDF/S, and Web Ontology Language (OWL) have been proposed by the W3C. These technologies provide a means to integrate disparate data sources and reuse data across applications through the use of ontologies, and their flexibility and ease of adoption have resulted in their pervasive acceptance.

However, given the fact that approximately 70% of the websites that were available in 2007 were backed by relational databases [1], and the percentage can only have increased further till date, it follows that the success and longevity of adoption of the Semantic Web depends heavily on enabling access of data within these relational databases to the Semantic Web. Moreover, relational databases, by virtue of having been in existence for several decades now, have the advantage of being equipped with sound, refined, and efficient query optimization, transaction support, data concurrency and security techniques. Thus, in order to reap the benefits offered by Semantic Web technologies while continuing to exploit the advantages of well-established and scalable relational database technologies, a means to enable the integration of the two technologies so as to create a “best of both worlds” scenario needs to be established. The need to address and provide a solution to this integration problem has prompted the initiation of the RDB2RDF incubator group [2] whose primary objective is to establish mapping standards that facilitate relational database and RDF interoperability.

The problem of bridging Relational Database Management System (RDBMS) and RDF concepts has been the focus of several research efforts [3–6] currently underway, each of which attempt to transform/propagate existing as well as newly added/modified data housed in relational databases into virtual RDF stores. However, almost all current solutions offer merely a read-only view of data from one domain into another. Thus, while one can view the relational data in RDF graph form and can query the resultant RDF triples using SPARQL [7], RDF’s native query language, data in the underlying relational database cannot be added to, deleted from, or altered in any way through the virtual RDF graph corresponding to the relational database. In this paper, we propose a solution that eliminates this data modification restriction thus allowing data flow in either direction. Towards this end, we present D2RQ++,<sup>a</sup> a bi-directional data flow facilitating enhancement to an existing, extensively adopted relational-to-RDF read-only translation tool called D2RQ. The version of D2RQ++ presented in this paper builds on [8]

<sup>a</sup><http://cs.utdallas.edu/semanticweb/D2RQ-Ext/d2rq-extension.html>

by including the ability to propagate data changes specified in the form of RDF triples containing blank nodes and RDF reification nodes, in addition to regular RDF triples, back to the underlying relational database.

Blank nodes, an important component of RDF graphs, are used to represent complex data. They are neither URI references nor literals and they enable association of a set of related properties with a resource, thereby creating a composite relationship.

RDF reification is a means of validating an RDF triple based on the trust level of another statement [9] and is an important facility provided by RDF that enables users to make assertions about statements and record provenance data, thereby facilitating appropriate authentication of RDF data. No application or tool that works with RDF data is complete without RDF reification support and, hence, the original version of D2RQ++ [8] has been augmented with algorithms that permit insert/update/delete of reification data. As before, when triples cannot explicitly be translated into equivalent concepts within the underlying relational database schema, D2RQ++ continues to adhere to the Open-World Assumption by permitting those triples to be housed in a separate native RDF store. When information on a particular entity is requested, the output returned is a union of the data pertaining to the entity from the relational database as well as any triples that have the entity as the subject and that may exist in the native RDF store. Thus, RDF triples submitted for insertion/update/deletion are never rejected due to mismatches with the underlying relational schema, thereby maintaining the Open-World Assumption of the Semantic Web world while still being able to work with technologies such as RDBMSs which are based on the Closed-World Assumption. The contributions of this paper include:

- Algorithms to translate RDF update triples into equivalent relational attributes/tuples thereby enabling DML operations on the underlying relational database schema.
- Extensions to support translation of a wide variety of blank node structures to equivalent relational tuples.
- Enhancements to support and permit triples that represent the concept of RDF reification to be propagated back into the underlying relational database schema through DML operations.
- Preservation of the Open-World Assumption by maintaining a separate native RDF store to house triples that are mismatched with the underlying relational database schema.
- Incorporation of the above algorithms and extensions into D2RQ++, an enhanced version of the highly popular D2RQ open-source relational-to-RDF mapping tool, and into D2R++-Server, the update-aware front-end graphical user interface (GUI) through which users can now issue DML requests using RDF data.

The organization of this paper is as follows. Section 2 presents a brief overview of related research efforts in the Relational-to-RDF arena. The challenges involved

in bi-directional translation of relational data into RDF triples and vice versa, and, D2RQ++, our approach towards addressing these challenges are presented in Sec. 3 along with the algorithms comprising D2RQ++, with particular emphasis on blank nodes, in Sec. 4. RDF reification and its bi-directional translation methodology is presented in Sec. 5 while Sec. 6 highlights the implementation specifics of the proposed system with sample translation screenshots and performance graphs for the insert/update/delete process for databases of various sizes. Lastly, Sec. 7 concludes the paper.

## 2. Related Work

Several research efforts exist that attempt to bring relational database concepts and Semantic Web concepts together in a uni-directional, read-only manner. One such effort is the D2RQ project [3] which is essentially a mapping between relational schema and OWL/RDF-Schema (RDFS<sup>b</sup>) concepts. D2RQ takes a relational database schema as input and presents an RDF interface of the same as output. Our work in this paper is centered completely around D2RQ and attempts to extend the same to permit insert, update, and delete operations on the underlying RDBMS. The work in [4] is yet another effort that, like D2RQ, uses a declarative meta schema consisting of quad map patterns that define the mapping of relational data to RDF ontologies. RDF123 [10], an open source translation tool, also uses a mapping concept, however its domain is spreadsheet data and it attempts to achieve richer spreadsheet-to-RDF translation by allowing the users to define mappings between spreadsheet semantics and RDF graphs. The work in [11–13] describe more mapping attempts in the reverse direction. In [11] the authors use relational. OWL to extract the semantics of a relational database, automatically transform them into a machine-readable RDF/OWL ontology, and use RDQuery [12] to translate SPARQL queries to SQL. The authors in [13] also essentially perform a relational-to-ontology mapping but here, they expect to be given some target ontology and some simple correspondences between the atomic relational schema elements and the concepts in the ontology to begin the mapping process with.

More recent research efforts in the relational-to-RDF mapping arena include the work done in [5, 14, 15]. Triplify [5] is another effort at publishing linked data from relational databases, and it achieves this by extending SQL and using the extended version as a mapping language. The View-Based Triplify method in [14] achieves the RDBMS-to-RDF transformation using a set of simple mapping rules based on which traditional relational views, each of which describes a distinct RDF class (and includes all the corresponding properties), are created. [15] is another mapping effort that uses a set of well-defined rules, like [14], to map relational schema metadata into an equivalent RDFS ontology which is then written into an RDF/XML file. Several endeavors [16–20] are underway in the data integration arena as well that

<sup>b</sup><http://www.w3.org/TR/rdf-schema/>

aim to present a semantically unified RDF model derived from multiple underlying heterogeneous databases.

Each of the efforts presented above is uni-directional as all of them just allow a read-only view of the relational database with nothing coming back into the same. ONTOACCESS [21] is the only effort we have been able to identify that attempts bi-directionality. In ONTOACCESS, the authors define a new mapping language called R3M which is very similar to the D2RQ mapping language [22], and they include support for the SPARQL/Update language [23] for data manipulation. D2RQ++, on the other hand, avoids learning curves associated with new languages by reusing, and extending when required, D2RQ's mapping language. By reusing D2RQ's mapping language, D2RQ++ also eliminates the effort and resources associated with creation of new languages. Another primary difference between ONTOACCESS and our approach, i.e. D2RQ++, is support for the Open-World Assumption. While ONTOACCESS accepts only those updates/inserts that have an equivalent relational concept in the underlying database, D2RQ++ can work with mismatched data as well (as described in the previous section), which is a key requirement of RDF's Open-World Assumption, thus proving itself to be an authentic Semantic Web application. Yet another difference between ONTOACCESS and D2RQ++ is the ability to accommodate updates/deletes of blank node and reification node structures. ONTOACCESS makes no mention of how incoming blank node structures are handled while D2RQ++ is capable, as illustrated in Secs. 4 and 5, of translating a variety of blank nodes and reification nodes, into equivalent relational structures thereby enabling the blank node contents to be transmitted to the underlying relational schema.

As can be seen from the discussions the issue of enabling bi-directional data transfer between relational and RDF applications has barely been addressed. ONTOACCESS is the only research that comes close to the objectives of D2RQ++ but it, too, has certain drawbacks as described above. Hence, to the best of our knowledge, D2RQ++ is one of the first endeavors to address the issue of bi-directionality in the relational-to-RDF mapping arena.

### 3. D2RQ++ — Our Approach

As stated earlier, the goal of our enhancement is to make the translation between RDF and RDBMS data stores bi-directional, thereby permitting update activities that can be propagated back to either the underlying RDBMS itself or to a native RDF store. We use the oldest and most recognized Employee-Department-Project relational schema depicted in Fig. 1 to reinforce concepts wherever applicable in subsequent sections. (We would like to emphasize that the Fig. 1 scenario was chosen purely for elucidation purposes. D2RQ++, however, can be used on any relational schema and is not just restricted to the scenario depicted in Fig. 1.)

There were several issues that needed to be addressed in order to achieve bi-directional translation and these are discussed in the following sub-sections.

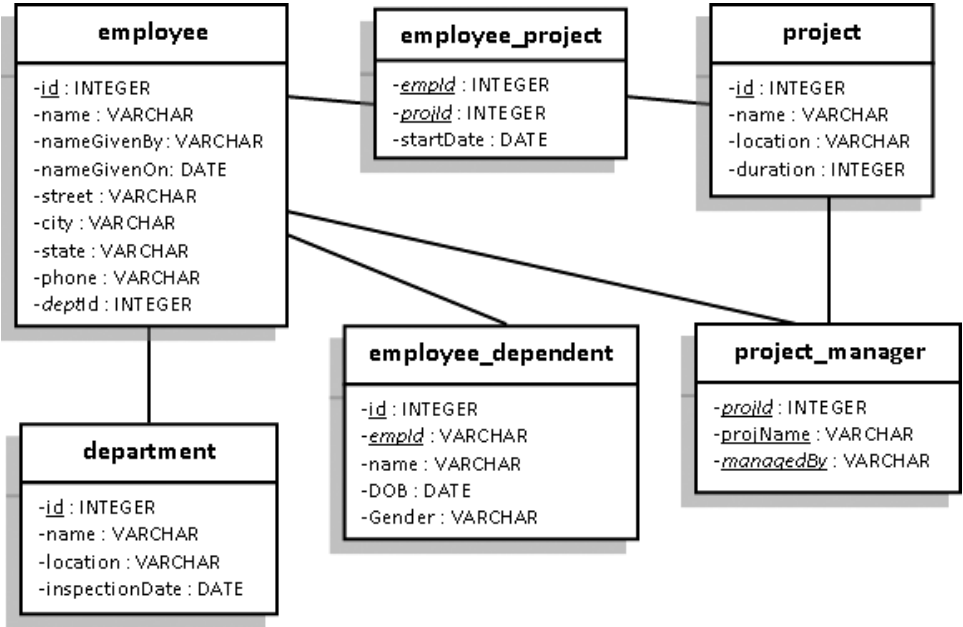


Fig. 1. Relational schema used to illustrate D2RQ++.

3.1. Persistence of unmapped and/or duplicate information

The first issue involved trying to preserve the Open-World assumption expected by Semantic Web applications and standards such as RDF, OWL, etc. during DML operations. In order to address this issue, we chose to maintain a separate native RDF store which would house all those triples that did not have an equivalent entity/attribute mapping within the underlying relational database schema. Even when an equivalent mapping exists, duplicate triples (such as a second *name* attribute value for a given employee) are housed in the native RDF store, instead of overwriting the existing value within the relational database schema and, thereby, losing the earlier information. Subsequent querying of the relevant employee information would return both *name* values (from the RDBMS and the RDF store) to the end user.

3.2. Mapping and persistence of RDF blank nodes

The second issue involved arriving at a translation process for RDF blank nodes while continuing to re-use existing mapping languages such as D2RQ since this concept does not have an equivalent relational database mapping. In order to facilitate DML operations involving blank nodes we added several new mapping constructs to D2RQ’s mapping language. These constructs are listed in Table 1.

The first three constructs essentially identify a specific concept in the relational database schema as a blank node and include information, specified through the

Table 1. Extensions to D2RQ's mapping constructs.

Construct	Description
d2q:SimpleLiteralBlankNodeProperty Bridge (SLBNPB)	Blank nodes that have only literal objects, each with a unique predicate
d2q:ComplexLiteralBlankNodeProperty Bridge (CLBNPB)	Blank nodes that have only literal objects; however, the predicates are not unique and include repetitions
d2q:ResourceBlankNodeProperty Bridge (RBNPB)	Blank nodes that have only resource objects with predicates that may or may not be unique.
d2q:belongsToBlankNode	Construct that helps link a relational attribute to the parent blank node

“d2q:pattern” construct, on the format in which object values should be specified for the blank node. These blank node classifications are similar to, and based on, the scenarios and examples discussed in [24] and more details can be found in the same. The last construct is used to identify those attributes within the relational database schema that together make up the `d2q:{SL/CL/R}BlankNodePropertyBridge`. This (last) construct is used to associate the attributes comprising a blank node with their parent blank node. In order to better understand how blank nodes are mapped in D2RQ++, let us consider the address attributes in the *employee* entity, viz., *street*, *city*, and *state*. If the end user prefers to view (or update) these address attributes in the form of a blank node that contains these attributes as objects, the following are the mapping statements the user would add to D2RQ's map file.

```
map:employee_address a
  d2q:SimpleLiteralBlankNodePropertyBridge;
  d2q:belongsToClassMap map:employee;
  d2q:property vocab:employee_address;
  d2q:propertyDefinitionLabel "employee address";
  d2q:pattern "@@employee.address_street@@/
              @@employee.address_city@@/
              @@employee.address_state@@";

map:employee_address_street a d2q:PropertyBridge;
  d2q:belongsToBlankNode map:employee_address;
  d2q:belongsToClassMap map:employee;
  d2q:property vocab:employee_address_street;
  d2q:propertyDefinitionLabel "employee address_street";
  d2q:column "employee.address_street";
```

Due to space constraints, the entry corresponding to only one of the address attributes (*street*) is shown above. Similar entries will have to be included for the other two attributes (*city*, *state*) as well. The *employee\_address* blank node is characterized as a SLBNPB as every object belonging to the blank node (i.e. *street*,

*city, state*) is a simple literal and each of these objects has a unique predicate. More details on updating blank nodes are presented in Sec. 4 along with the appropriate algorithms.

### 3.3. Mapping and persistence of RDF reification nodes

The third issue, similar to the second issue above, involved arriving at a translation process for RDF reification nodes while continuing to re-use D2RQ's existing mapping language since this concept does not have an equivalent relational database mapping. RDF reification is a feature through which a triple can be encapsulated as a resource in order to enable additional statements about that resource to be made that help establish the degree of confidence and trustworthiness of the triple. This issue was resolved by extending D2RQ's mapping language with new constructs that are capable of handling reification data, and providing two ways in which reification information can be persisted using D2RQ++. The first alternative is trivial and involves storing all reification information directly into the native RDF store. This method is adopted by D2RQ++ when appropriate tables/columns to house reification information do not exist in the underlying relational database or when mapping information pertaining to reification nodes is not available in D2RQ++'s mapping file. The second option involves storing the data within the underlying relational database schema and is available if the schema includes appropriate tables and columns within which reification data can be stored. In-depth details on the mapping and persistence of reification nodes are provided in Sec. 5.

### 3.4. Maintenance of open-world assumption through periodic consolidation

The fourth issue was concerned with establishing the order of update activities (in the case of batch updates) in order to ensure that referential integrity constraints do not force an update rejection due to incorrect update sequences such as an *employee* triple of the form  $\{\langle empURI \rangle \langle DeptID \rangle \langle DepartmentID \rangle\}$  arriving before the actual *department* triple  $\{\langle deptURI \rangle \langle ID \rangle \langle DepartmentID \rangle\}$ . This issue was resolved by blindly accommodating triples that violate referential integrity constraints in the native RDF store and introducing a periodic consolidation/flush algorithm [8]. This consolidation/flush algorithm periodically validates the RDF store contents against the underlying relational database schema to identify those triples that now have parent key values in the relational schema corresponding to their object values, i.e. to identify those triples that earlier violated foreign key constraints but now no longer do because the parent key is now present in the relational schema. Once these triples are identified, the flush algorithm transfers them into the relational schema by following Algorithm 1 detailed in [8] and removes them from the native RDF store. The flush algorithm also consolidates duplicate triples in the event the underlying relational database column corresponding to the triple's predicate is updated to a null value. These duplicate triples were originally



accommodated in the native RDF store because the corresponding column in the underlying relational database had a non-null value. Whenever these columns are updated to null values, the flush algorithm consolidates duplicate triples back into the underlying database and deletes the same from the native RDF store.

The next section presents the various algorithms that were developed to address and resolve the issues presented above.

## 4. D2RQ++ Algorithms for Triples and Blank Nodes

### 4.1. *Insert/update operations on regular triples*

Insert/Update operations on regular RDF triples are fairly straightforward and are applicable to simple triples that involve a literal or resource object and that do NOT involve any blank nodes. The only time an INSERT statement is executed against the underlying relational schema is when the predicate exists as a column in the table to which the subject of the triple belongs and the subject value itself does not exist as a primary key value in the same table. When the predicate exists as a column in the table and the subject exists as a primary key value in the same table, the object value is updated (using an SQL UPDATE statement) only if the corresponding cell in the relational schema is empty. If not, under the Open-World Assumption, the object in the input triple is considered to be a duplicate value for the corresponding column and is preserved by housing the triple in the native RDF store. Subsequent querying of that column will return both values, i.e. the cell value stored in the relational database as well as the object value for the corresponding predicate stored in the native RDF store. In the event the predicate of the input triple does not map to an equivalent column in the underlying relational schema as specified in the mapping file, the input triple is always added into the native RDF store. The algorithm for insert/update operations on regular RDF triples can be found in [8].

### 4.2. *Insert/update operations on RDF blank nodes*

Triples that contain blank node objects are handled using two procedures. The first procedure, found in [8], handles insert/update operations involving simple or complex literal blank nodes while the second procedure, found in [27], deals with resource blank nodes. Due to space constraints a detailed discussion of these algorithms are omitted here and readers are referred to the appropriate references [8, 27] for more information.

### 4.3. *Consolidation procedure*

As stated earlier, the order of insert/update activities may result in certain triples being rejected from the RDBMS and being housed temporarily in the native RDF store instead due to the violation of referential integrity constraints. Periodically, the triples in the native RDF store are validated against the underlying relational database and any triple that no longer violates referential integrity constraints

is transmitted back to the relational database and deleted from the native RDF store using the Flush Algorithm [8]. This algorithm, run periodically, is also quite straightforward and applies to any triple in the native RDF store with a resource object. Whenever the object value of such a triple is found to exist as a primary key in the table corresponding to the Object Resource class, and the predicate of the triple exists as a column in the table corresponding to the subject of the triple, if the subject does not already exist in the subject table in the RDBMS, a new tuple is inserted with the subject and object values of the triple being updated in the appropriate relational columns. If the subject exists as a primary key in the underlying RDBMS table and the current predicate column value is null, it is updated and set to the object value of the triple; otherwise no update happens and the triple continues to remain in the native RDF store. In the event a successful insert/update of the triple was accomplished in the underlying relational schema, the triple is then deleted from the native RDF store. In this manner, referential integrity violations are given opportunities to return to the underlying relational schema periodically. Similar flush procedures exist for CLBNPBs and RBNPBs as well, however, due to space constraints, they are not presented here.

#### 4.4. Delete operations on RDF triples and blank nodes

The process to delete a regular triple from either the RDBMS or the native RDF store as applicable is, again, fairly straightforward and can be found in [8].

Delete procedures for {SL/CL/R}BNPBs are illustrated in Algorithms 1a and 1b below.

**Algorithm 1a: DeleteLiteralBlankNode**

**Input:** An RDF Triple with Literal Blank Node object

**Output:** A successful RDBMS/RDF delete

- |   |
|---|
| <ol style="list-style-type: none"> <li>1. If BlankNode definition not found in Map File</li> <li>2. Delete BlankNode structure from native RDF Store &amp; Return</li> <li>3. End If</li> <li>4. If BlankNode structure does not match Map File definition</li> <li>5. Delete BlankNode structure from native RDF Store &amp; Return</li> <li>6. End If</li> <li>7. If subject of BlankNode does not exist in RDBMS</li> <li>8. Delete BlankNode structure from native RDF Store &amp; Return</li> <li>9. End If</li> <li>10. If BlankNode.Type = SLBNPB then</li> <li>11. For every predicate off of the BlankNode Do</li> <li>12. If value in column (of subject table in row where PK = SLBNPB's subject) corresponding to predicate != predicate's object</li> <li>13. Delete BlankNode structure from native RDF Store &amp; Return</li> <li>14. End If</li> </ol> |
|---|

**Algorithm 1a: (Continued)**

```

15. End For
16.   UPDATE subject table set values of columns corresponding to
      SLBNPB's predicates to NULL in subject table WHERE
      PK = SLBNPB's subject
17. Else If BlankNode.Type = CLBNPB then
18.   Get RDBMS table corresponding to object of CLBNPB
19. For every predicate belonging to CLBNPB
20.   DELETE subject, predicate, and object of CLBNPB FROM RDBMS
      table from the Foreign Key, Type, and Value fields respectively & Return
21. End For
22. End If

```

**Algorithm 1b: DeleteResourceBlankNode****Input:** An RDF Triple with Resource Blank Node object**Output:** A successful RDBMS/RDF delete

```

1. If BlankNode definition not found in Map File
2.   Delete BlankNode structure from native RDF Store & Return
3. End If
4. If BlankNode structure does not match Map File definition
5.   Delete BlankNode structure from native RDF Store & Return
6. End If
7. If subject of BlankNode does not exist in RDBMS
8.   Delete BlankNode structure from native RDF Store & Return
9. End If
10. For every predicate belonging to RBNPB Do
11.   Get tables corresponding to resource subject and object classes
      in RDBMS
12. If object value does not exist in object RDBMS table
13.   Delete RBNPB structure from native RDF store & Return
14. End If
15. If RBNPB's subject and current object class relationship is 1:N
16.   Get object table's field corresponding to subject
17. If field value != subject value WHERE table.PK = object value
18.   Delete RBNPB structure from native RDF store & Return
19. End If
20. End If
21. End For
22. For every predicate belonging to RBNPB Do
23. If RBNPB subject and predicate object relationship = 1:N
24.   UPDATE object table SET field corresponding to subject to NULL
      WHERE object table PK = object value of predicate

```

**Algorithm 1b: (Continued)**

```

25. Else
26.   DELETE subject and object values FROM N:M (join) table
27. End If
28. End For

```

The next section discusses the mapping language extensions and algorithms that enable inserts, updates, and deletes of reification information to be either propagated back to the relational database schema or stored in a native RDF store as applicable.

### 5. Bi-Directional Translation of RDF Reification Nodes

RDF reification, as described in Sec. 3.3, is an important feature that enables the establishment of the degree of confidence and trustworthiness of a triple. RDF reification nodes consist of four mandatory properties: `rdf:subject`, which identifies the subject in the original triple being reified, `rdf:predicate`, which identifies the property in the statement being reified, `rdf:object`, which identifies the object of the statement being reified, and, lastly, `rdf:type`, which is always `rdf:statement` since all reified statements are instances of `rdf:Statement`. These four properties are called the reification quad and these can be accompanied by one or more non-quad properties that detail the actual provenance information. These non-quad predicates can have one or more literal, resource, or blank node objects as illustrated in the sample scenario in Fig. 2.

In the graph, every solid node with outgoing edges, such as *URI/Emp1* and *URI/Dept1*, represents a subject/resource. Solid edges, such as *Street*, *City*, *State*, and *Phone*, represent predicates and the solid nodes at the end of the edges, such as *<Street>*, *<City>*, *<State>*, and *<Phone>*, represent objects. Empty solid nodes, such as the node at which the *Projects* predicate terminates represent blank nodes. The nodes in dashed lines with the “s”, “p”, “o”, and “t” predicates, amongst others, represent reified nodes. “s”, “p”, “o”, and “t” represent the “`rdf:subject`”, “`rdf:predicate`”, “`rdf:object`”, and the “`rdf:type`” predicates of the reification quad. Other predicates of the reification nodes (other than “s”, “p”, “o”, and “t” predicates) represent non-quad predicates (NQPs). Empty nodes in dashed lines (other than the reified nodes) that are the objects of non-quad reification predicates, such as the object of the *ManagedBy*, and *NameDetails* predicates, represent reification blank nodes. The non-quad reification properties chosen in this example may not represent actual provenance information. They were primarily chosen to illustrate proof of concept. Elements of Fig. 2 are used, wherever applicable, to facilitate better comprehension of the information presented in subsequent sections.

Reification information can be persisted using D2RQ++ in two ways as detailed in Sec. 3.3. As can be seen in Fig. 2, reification nodes comprise different types of non-quad predicates. We have classified reification nodes into five broad categories

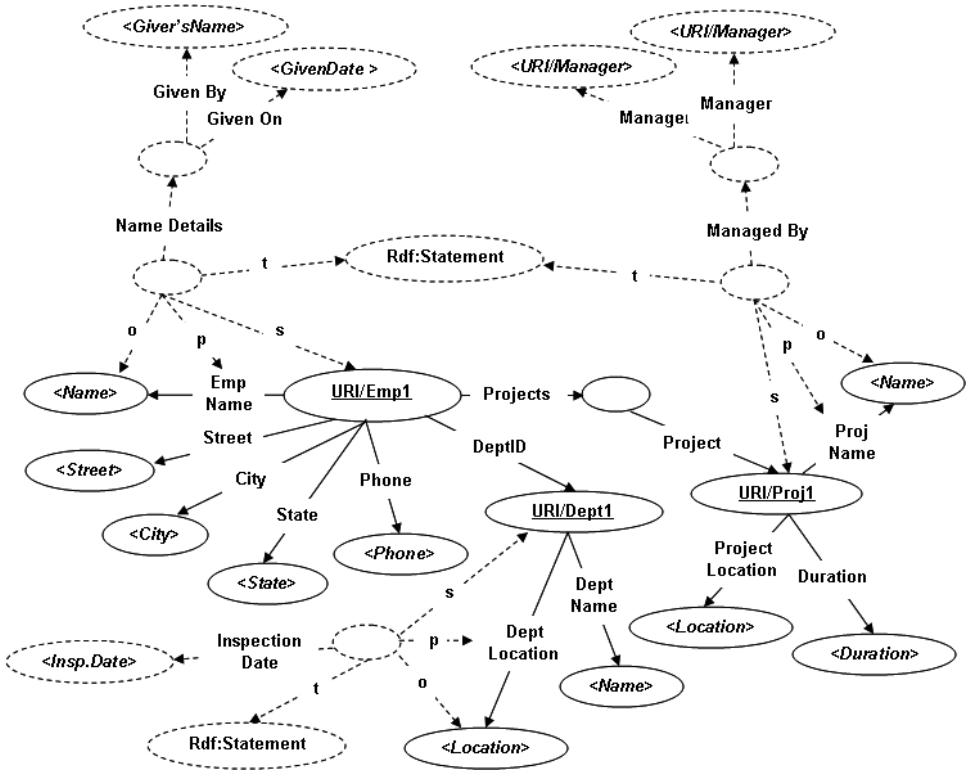


Fig. 2. Sample RDF scenario with reification nodes.

based on the type of non-quad predicates they contain and these categories are described in the following sub-section.

### 5.1. Reification node categories and their relationalization

#### (a) Reification Nodes (RNs) with non-repeating literal/resource NQPs

As the category implies, these RNs contain one or more predicates, each of which is unique, and each of which leads to a literal or resource object. An example of these kinds of RNs is the RN with the *InspectionDate* NQP, leading to a literal object, in Fig. 2.

- (i) *Relational Storage of RNs with literal rdf:objects*: Storage of this kind of reification information when the RN reifies a literal object (i.e. *rdf:object* is a literal) within a relational database requires columns for each of the non-repeating NQPs in the table corresponding to the *rdf:subject* of the RN, and storing the NQP objects in those columns. Thus, the object of the *InspectionDate* NQP in Fig. 2 is stored within the *inspectionDate* column in the *department* table (corresponding to the *URI/Dept1* *rdf:subject*).

- (ii) *Relational Storage of RNs with resource rdf:objects*: In the case of resource reification objects (i.e. resource rdf:objects), if the relationship between rdf:subject and rdf:object of the RN is 1:N the NQPs correspond to columns in the table on the N-side of the relationship; if the relationship is N:M the NQPs correspond to columns in the join table that results from the translation of this N:M relationship.

(b) *RNs with single or multiple groups of repeating literal/resource NQPs*

These RNs also have NQPs leading to literal or resource objects; however, the RNs NQPs need not be unique and can repeat for multiple object values. For example, if a department location has been inspected multiple times, the corresponding reification information in Fig. 2 will include multiple NQPs, with a repeating predicate of *InspectionDate*, each of which has a unique inspection date object. This is an example of a single group of repeating literal NQPs (since only one predicate is repeated. Multiple groups of repeating predicates are scenarios where more than one predicate, each of which are repeated a number of times, exist).

- (i) *Relational Storage of RNs with literal rdf:objects*: RNs in this category with literal rdf:objects are stored within a relational database in a table with the following columns (*rdf:subject Table PK*, *rdf:predicate column*, *type*, *value*) with the primary key comprising all four columns. The first column is a foreign key that references the primary key of the table corresponding to the RN's rdf:subject (the *department* table in our example), the second column is the one in the rdf:subject's table that corresponds to the RN's rdf:predicate (the value stored in this field is the rdf:object value), the third column (*type*) stores the name of each NQP belonging to the RN, and the fourth column (*value*) stores the object value of each of the NQPs.
- (ii) *Relational Storage of RNs with resource rdf:objects*: To store RNs containing resource rdf:objects in a relational schema, the repeating NQPs and their objects are stored within the *type* and *value* fields in the join table (which also contains the (*rdf:subject table PK*, *rdf:objectTablePK*) foreign key fields) for the {rdf:subject, rdf:object} pair.

(c) *RNs with Simple Literal Blank Nodes (SLBNs)*

These are RNs whose non-quad predicates comprise simple literal blank nodes and an example of one such RN is the one with the *NameDetails* NQP in Fig. 2.

- (i) *Relational Storage of RNs with literal rdf:objects*: Relational storage of RNs in this category with literal rdf:objects involves translating each of the predicates comprising the SLBN non-quad predicate into appropriate columns in the table corresponding to the RN's rdf:subject (similar to the relationalization process in category (a) (i) above). Thus, information pertaining to the *NameDetails* literal SLBN NQP in Fig. 2 is stored within the columns *name-GivenBy* and *nameGivenOn* belonging to the *employee* (rdf:subject) table in the relational schema in Fig. 1.

- (ii) *Relational Storage of RNs with resource rdf:objects*: The procedure for storing RNs with resource rdf:objects in this category within a relational database is identical to the process detailed in category (a) (ii) above.

(d) *RNs with Complex Literal Blank Nodes (CLBNs)*

RNs whose NQPs comprise of CLBNs fall into this category. A scenario where the *NameDetails* reification SLBN NQP in Fig. 2 consists of two, instead of one, *GivenBy* predicates and one *GivenOn* predicate is an example of a CLBN NQP since there are multiple non-unique (repeating) predicates belonging to the *NameDetails* blank node.

*Relational Storage*: Relational storage of predicates belonging to CLBN NQPs of RNs with literal and resource rdf:objects in this category requires the same procedures as detailed in category (b) (i) and (ii) respectively.

(e) *RNs with Resource Blank Nodes (RBNs)*

This category comprises RNs whose NQPs consist of resource blank nodes. An example of such an RN in Fig. 2 is the RN with the *ManagedBy* NQP which is an RBN consisting of multiple *Manager* entities which are resources, possibly of type *employee*.

- (i) *Relational Storage of RNs with literal rdf:objects*: These RNs are stored within a relational database in a table with the following columns: (*rdf:subject Table PK*, *rdf:predicate column*, *predicate of RN's RBN NQP*) where the third column is a foreign key referencing the table corresponding to the objects of the RBN. Thus, the relational translation of the RN with the *ManagedBy* RBN NQP results in the addition of a new tuple in the *project.manager* table with the fields (*projId*, *projName*, *managedBy*) where *projId* is a foreign key referencing the primary key of the *project* table and *managedBy* is a foreign key referencing the *employee* table.
- (ii) *Relational Storage of RNs with rdf:objects*: Relationalization of RBN RNs with resource rdf:objects involves identifying the join table (containing the (*rdf:subject table PK*, *rdf:objectTablePK*) foreign key fields) representing the relationship between the rdf:subject and rdf:object classes, identifying the column that is a foreign key referencing the table corresponding to the resource objects of the RBN NQP, and adding the objects of the RBN NQP as new tuples in the identified table and column.

There are several other categories of reification nodes such as RNs that contain multiple object classes, with or without unique predicates, RNs that contain nested blank nodes, and RNs that contain a mixture of literals, resources, and blank nodes. Further, several other scenarios where reification can be applied also exist such as reification of multivalued attributes, reification of triples where the subject or object is a blank node. However, the scope of this paper is restricted to the five categories described above as space constraints prohibit us from including these extended scenarios. We plan to include the extended scenarios in subsequent publications.

### 5.2. Mapping language extensions for reification support

For SPARQL queries and other tools that work with the virtual RDF store generated by D2RQ++ to distinguish between regular tables/columns and tables/columns that cater exclusively to reification data a mapping scheme that permits differentiation between the two kinds of tables/columns needs to be established. In order to achieve this requirement, we further extended D2RQ's mapping language by adding several mapping constructs specifically for reification support. These constructs are listed in Table 2.

The first two constructs are used to identify provenance data stored within the relational database schema and map the same as reification nodes to ensure that the appropriate reification quads are generated in the virtual RDF graph generated through the translation of the relational schema. More details on reification nodes and their classifications can be found in [25]. The last construct is used to identify those columns in the relational schema that correspond to a reification node's non-quad predicates and associate those columns with the parent reification node.

In order to better understand the mapping details pertaining to reification information, let us consider the attributes *employee.nameGivenBy*, *employee.name-GivenOn*, *department.inspectionDate*, and *project.manager.managedBy* in Fig. 1 that correspond to NQPs in reification nodes. The first two attributes are examples of an RN with an SLBN NQP, the third attribute is an example of an RN with a simple literal NQP, and the last attribute is an example of an RN with an RBN NQP. These attributes are translated into appropriate reification node predicates in the equivalent virtual RDF store using the following mapping statements in D2RQ++'s mapping file.

```
map:employee_name_reif a d2rqrw:ReificationNode;
  d2rq:belongsToClassMap map:employee;
  d2rqrw:reifiedPropertyBridge map:employee.name;

map:employee_name_nameDetails a
  d2rqrw:SimpleLiteralBlankNodePropertyBridge;
  d2rqrw:belongsToReificationNode map:employee_name_reif;
  d2rq:property vocab:employee_name_nameDetails;
  d2rq:propertyDefinitionLabel "employee Name NameDetails";
```

Table 2. RDF reification extensions to D2RQ's mapping constructs.

Construct	Description
d2rqrw:ReificationNode (RN)	Construct used to map reification nodes
d2rqrw:ReifiedPropertyBridge	Construct used to associate relational attribute corresponding to the predicate of the triple that is being reified (by the reification node) to the reification node
d2rqrw:belongsToReificationNode	Construct that helps associate a relational attribute corresponding to the reification node's non-quad predicate to the reification node



```
d2rq:pattern "@@employee.nameGivenBy@@/
@@employee.nameGivenOn@@";
```

```
map:employee_nameGivenBy a d2rq:PropertyBridge;
d2rqrw:belongsToBlankNode map:employee_name_nameDetails;
d2rq:belongsToClassMap map:employee;
d2rq:property vocab:employee_nameGivenBy;
d2rq:propertyDefinitionLabel "employee nameGivenBy";
d2rq:column "employee.nameGivenBy";
```

Due to space constraints, the entry corresponding to *nameGivenOn* is omitted as it is similar to the *nameGivenBy* entry above. The mapping entry for *department.inspectionDate* is very similar to the mapping for *employee.nameGivenBy* entry above except that the SLBNPB entry is excluded here since the *inspectionDate* NQP is a direct predicate to its RN without any intermediate blank nodes. The mapping entries for RNs with RBN NQPs such as the one corresponding to the *project\_manager.managedBy* attribute are given below.

```
map:project_manager_projName_reif a d2rqrw:ReificationNode;
d2rq:belongsToClassMap map:project_manager;
d2rq:refersToClassMap map:employee;
d2rqrw:reifedPropertyBridge map:project_manager.name;

map:project_manager_name_managedBy a
d2rqrw:ResourceBlankNodePropertyBridge;
d2rqrw:belongsToReificationNode map:project_manager_projName_reif;
d2rq:property vocab:project_manager_name_managedBy;
d2rq:propertyDefinitionLabel "project_manager name managedBy";
d2rq:pattern "@@project_manager.manager@@/
@@ project_manager.manager @@";

map:project_manager_manager a d2rq:PropertyBridge;
d2rqrw:belongsToBlankNode map: project_manager_name_managedBy;
d2rq:belongsToClassMap map:project_manager;
d2rq:refersToClassMap map:employee;
d2rq:property vocab:project_manager_manager;
d2rq:propertyDefinitionLabel "Project_Manager manager";
d2rq:column "project_manager.manager";
```

The above mappings enable reification information stored in relational database schemas to be mapped to appropriate reification nodes with all intermediate edges (such as blank node NQP edges) maintained, thus enabling accurate RDF graph transformations of relational reification attributes. Algorithms to insert, update, and delete information in RDF reification nodes within the underlying relational database schema are presented in the following subsection.

### 5.3. D2RQ++ algorithms for reification nodes

Insert and Update operations on RNs belonging to categories (a) and (c) in Sec. 5.1 are detailed in Algorithm 2. As can be seen in the algorithm, if the incoming reification node is not mapped appropriately in D2RQ++'s mapping file or if there is a mismatch in the map file definition and the actual node structure, the reification node is stored in the native RDF store instead of in the relational database schema. This is also the case when the `rdf:predicate` of the RN does not exist as a column in the table corresponding to the `rdf:subject`, when the `rdf:object` value does not exist in the column corresponding to the RN's `rdf:predicate`, when any one of the NQP predicates (in case (a)) or predicates of the SLBN NQP (in case (b)) do not exist as columns in the `rdf:subject`'s table, or when any of the NQP predicates or predicates of the SLBN NQP have non-null values.

When none of the above conditions exist and if the `rdf:subject` value of the RN does not exist as a value in the `rdf:subject`'s table, an INSERT statement is issued as illustrated in line 5 of Algorithm 2. If the `rdf:subject` value exists in the appropriate table then an UPDATE statement is issued as illustrated in line 11 of Algorithm 2.

**Algorithm 2: Insert/UpdateLiteral/SLBNReificationNode**

**Input:** An RDF Reification Node with literal/resource or SLBN NQPs

**Output:** A successful RDBMS/RDF Insert/Update

1. If ReificationNode (RN) definition not found in Map File OR  
RN structure does not match Map File definition OR  
RN's `rdf:predicate` column not present in table corresponding to RN's  
`rdf:subject` OR  
One or more fields corresponding to RN's NQPs or SLBN NQP's  
predicates not present in table
2.     Add RN to native RDF Store & Return
3. End If
4. If RN's `rdf:subject` value does not exist in table corresponding to RN's  
`rdf:subject`
5.     INSERT `rdf:subject` value, `rdf:object` value, and values of all NQP  
objects or objects of SLBN NQPs into `rdf:subject` Table PK,  
column corresponding to RN's `rdf:predicate`, and columns corresponding  
to non-quad predicates of RN & Return
6. End If
7. If RN's `rdf:subject` value exists but `rdf:object` value does not exist in  
`rdf:predicate` column of table
8.     Add RN to native RDF Store & Return
9. End If
10. If every column corresponding to RN's NQP or SLBN NQP's predicates is  
NULL

**Algorithm 2: (Continued)**

11. UPDATE NQP columns SET values = objects of NQPs in rdf:subject table WHERE PK = rdf:subject value & Return
12. Else
13. Add RN to native RDF & Return
14. End If

Algorithm 3 highlights the process to delete reification nodes belonging to categories (a) and (c) from either the RDBMS or the native RDF store as applicable.

**Algorithm 3: DeleteLiteral/SLBNReificationNode**

**Input:** An RDF Reification Node with literal/resource or SLBN NQPs

**Output:** A successful RDBMS/RDF Update/Removal

1. If ReificationNode (RN) definition not found in Map File OR  
RN structure does not match Map File definition OR  
RN's rdf:predicate column not present in table corresponding to RN's  
rdf:subject OR  
One or more fields corresponding to RN's NQPs or SLBN NQP's  
predicates not present in table
2. Remove RN from native RDF Store & Return
3. End If
4. If RN's rdf:subject value does not exist in table corresponding to RN's  
rdf:subject
5. Remove RN from native RDF Store & Return
6. End If
7. If RN's rdf:subject value exists but rdf:object value does not exist in  
rdf:predicate column of table
8. Remove RN from native RDF Store & Return
9. End If
10. If every column corresponding to RN's NQP or SLBN NQP's predicates  
has value = corresponding NQP's object value
11. UPDATE NQP columns SET values = NULL in rdf:subject table  
WHERE PK = rdf:subject value & Return
12. Else
13. Remove RN from native RDF Store & Return
14. End If

The only situation when reification data is removed from the underlying relational database is when every column value in columns corresponding to the RN's NQPs or SLBN NQP's predicates is equal to the object values of the corresponding predicates in the reification node. Under such a scenario, the appropriate columns in the table corresponding to the RN's rdf:subject are updated to NULL. In every other situation, the RN is stored in the native RDF store and hence, is deleted

there. The DELETE DML operation is never executed for reification nodes since RNs represent additional information about a triple and deleting reification information does not delete the triple which was reified. Thus, the triple, whose subject is stored as the primary key value in the appropriate table continues to exist and, consequently, the corresponding tuple in the relational table continues to exist as well. Thus, though the algorithm implements the DELETE operations, deletion never occurs in the relational database schema.

Insert/Update and Delete procedures exist for RNs belonging to the other categories described in Sec. 5.1 as well, however space restrictions prevent us from including the details here. Since each of the algorithms presented in Secs. 4 and 5 operate on individual triples, the time complexity of algorithms pertaining to regular triples is a constant,  $a$ , as the algorithms do not involve looping processes. Algorithms pertaining to blank nodes are dependent on the number of predicates that belong to the blank nodes and, hence, have a time complexity that is slightly higher and can be represented as  $a + bn$ , where  $n$  is the number of predicates belonging to the blank node of interest, and  $b$  is the time taken to insert or update or delete the information in each predicate of the blank node into the appropriate database (i.e. either the RDBMS or the RDF store).

Each of the algorithms described in the previous sections have been implemented as a wrapper around the original D2RQ application in order to make the relational-to-RDF transformation bi-directional. Screenshots of D2R++-Server, an enhanced version of D2R-Server [26], which includes the ability to receive insert/update/delete requests from the end users are presented in the next section as evidence of the bi-directionality of the transformation process.

## 6. Implementation Results

The hardware and software platforms used in the implementation of the various algorithms discussed in the previous sections, and the performance experiments conducted using the same, are described below.

### 6.1. Experimental platform

Ubuntu 10.04 with 3 GB RAM and 2.00 GHz Intel Processor was used as the operating system for our experiments. The translation and database tools used include D2RQ<sup>c</sup> 0.7 to perform the uni-directional translation of a relational database schema to an equivalent virtual RDF store, MySQL<sup>d</sup> 5.1.37 to store the relational database schema to be translated into an equivalent RDF store, and Jena<sup>e</sup> 2.6.3 to house the native RDF store that stores the RDBMS-rejected insert/update triples. Software development platforms used include the Eclipse 3.4.0 IDE and Java 1.6 for the development of the algorithms and procedures detailed in Secs. 4 and 5.

<sup>c</sup><http://www4.wiwiw.fu-berlin.de/bizer/d2rq/>

<sup>d</sup><http://www.mysql.com>

<sup>e</sup><http://jena.sourceforge.net>

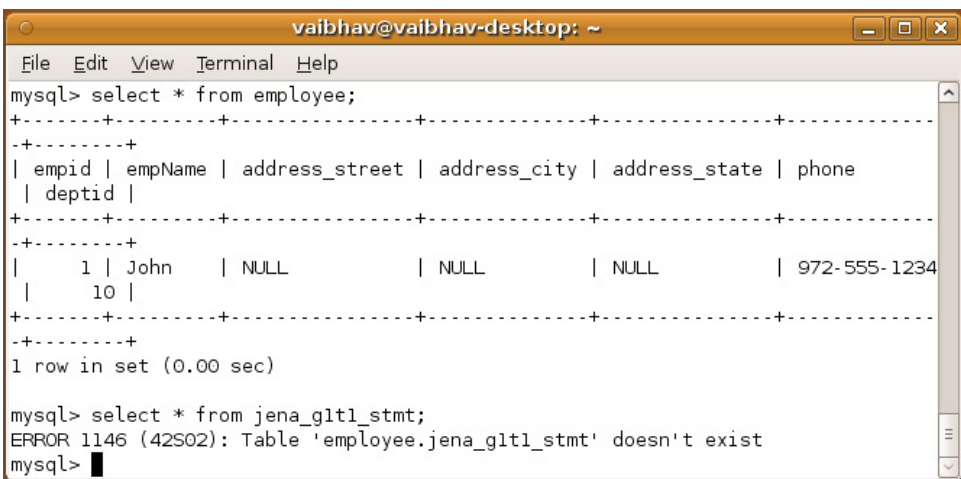
## 6.2. Experimental dataset

The performance experiments conducted and the D2R++-Server GUI outputs presented below are based on a subset of the Employee-Department-Project scenario illustrated in Fig. 1. Synthetic RDF triple datasets of various sizes corresponding to the relational schema as defined by D2RQ's mapping file were created through a data loading program and populated using a Semantic Web Toolkit, Jena, in order to evaluate performance of the insert/update operations performed by our bi-directional algorithms.

## 6.3. Experimental results

For the purposes of our experimentation and proof of viability, we used Jena's RDB Model as the native RDF store that houses the RDBMS-rejected insert/update triples. Further, in our experiments, the RDB Model is housed in the same MySQL database that houses the actual relational schema. However, in a production environment, the native RDF store will, in all probability, be housed in a completely separate MySQL database. Additionally, the system administrator may also prefer to use an in-memory model rather than a persistent model for the native RDF store. These are design decisions that are left at the administrator's discretion.

In addition to extending the D2RQ application by including our insert/update algorithms, we also extended the D2RServer [26] front-end GUI application to D2R++-Server which includes provisions to *add/remove* RDF triples. Figures 3–7 illustrate the enhanced D2R++-Server's *add/remove* extensions and D2RQ++'s ability to propagate RDF blank nodes and reification nodes, as new tuples or updates to existing tuples, back to the underlying relational database or native



The screenshot shows a terminal window titled 'vaibhav@vaibhav-desktop: ~'. The terminal displays the output of two MySQL queries. The first query, 'mysql> select \* from employee;', returns a single row of data for an employee named John. The second query, 'mysql> select \* from jena\_glt1\_stmt;', results in an error message: 'ERROR 1146 (42S02): Table 'employee.jena\_glt1\_stmt' doesn't exist'.

```

mysql> select * from employee;
+-----+-----+-----+-----+-----+-----+
| empid | empName | address_street | address_city | address_state | phone |
| deptid |
+-----+-----+-----+-----+-----+-----+
|      1 | John    | NULL          | NULL         | NULL          | 972-555-1234 |
|      10 |         |               |              |               |              |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from jena_glt1_stmt;
ERROR 1146 (42S02): Table 'employee.jena_glt1_stmt' doesn't exist
mysql>

```

Fig. 3. Initial data in *employee* RDBMS table.

RDF store, as applicable. Due to space constraints screenshots of only a few DML operations are included here. More screenshots, illustrating a wider variety of insert/update/delete operations, can be found in [27]. Data in the *employee* table prior to any new insert/update activities included a single record for an employee named “John” with no address information as can be seen in Fig. 3.

Figures 4 and 5 illustrate the addition of address information in the form of an SLBN through the D2R++-Server application, along with the corresponding updates within the appropriate RDBMS schema or native RDF Store as applicable. Since the columns corresponding to the predicates of the SLBN are all null initially, the objects of the SLBN are updated against the “John” tuple within the relational database schema table, *employee*, as illustrated in the MySQL Window in Fig. 4.

Figure 5 illustrates the addition of a second address for “John” (through the D2R++-Server application). As can be seen from the appropriate backend database queries in the MySQL window, since the relational table *employee* has an address for “John” already, the new address is added to the native RDF store.

Figure 6 illustrates D2RQ++’s reification support through the output from a simple SPARQL query to list all triples. As shown, the output includes data from the relational database schema (triplified) as well as from the native RDF store.

The output includes the *inspectionDate* reification data added to *deptId* 10 in the *Department* table through Jena API. Since the relational schema used in our experimentation did not include a column to store the reification data, and, consequently, there was no mapping in the D2RQ++ map file that corresponds to *inspectionDate*, this reification information is stored in the native RDF store instead of in the relational database schema as illustrated in Fig. 7.

The time taken for the insert/update/delete algorithms for regular RDF triples is illustrated in Fig. 8. For each operation (i.e. insert, update, and delete) performance statistics for three different scenarios were generated. In the first scenario, the operation under consideration only affects data in the underlying relational database; in the second scenario, the only affected data are the ones that exist in the native RDF store; and the third scenario is one in which 50% of the affected data are housed in the underlying relational database while the other 50% is housed in the native RDF store. Since the concept of update in RDF stores is implemented through a delete-insert combination, performance statistics for the second scenario (affected data existing only in the native RDF store) in the second graph (performance statistics for update operations) are obtained by adding the second scenario statistics from graphs 1 (insert) and 3 (delete).

As can be seen from Fig. 8, the algorithms perform well for small datasets but are somewhat time intensive for larger number of records. Further, while the performance of Insert and Delete operations are similar for smaller datasets, the performance of the Insert operation degrades faster than that of the Delete operation. This difference in time arises because the Insert and Update algorithms need to check more conditions per triple than the Delete algorithm. Further, as the database

The screenshot displays a web browser window titled "employee #1 | D2R++ Server - Mozilla Firefox". The address bar shows "http://localhost:2020/page/employee/1". The page content includes a header for "employee #1" with the Resource URI "http://localhost:2020/resource/employee/1". Below this is a navigation bar with "Home" and "All employee". A table lists properties and their values for the employee:

Choose	Property	Value
<input type="radio"/>	vocab:employee_address	null
<input type="radio"/>	vocab:employee_address_city	null
<input type="radio"/>	vocab:employee_address_state	null
<input type="radio"/>	vocab:employee_address_street	null
<input type="radio"/>	vocab:employee_deptid	10 (xsd:int)
<input type="radio"/>	vocab:employee_empName	John
<input type="radio"/>	vocab:employee_empid	1 (xsd:int)
<input type="radio"/>	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Below the table is a "Remove" button. Further down, a section for PREFIX definitions is shown, including:

```

PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema#
PREFIX db: http://localhost:2020/resource/
PREFIX owl: http://www.w3.org/2002/07/owl#
PREFIX xsd: http://www.w3.org/2001/XMLSchema#
PREFIX map: file:/home/vaibhav/employee.n3#
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
PREFIX vocab: http://localhost:2020/vocab/resource/

```

A text input field contains "address~~map:employee\_address@@Drive A/Richardson/Texas" with an "Add" button below it. The browser window is generated by "D2R++ Server".

Below the browser window is a terminal window titled "vaibhav@vaibhav-desktop: ~". It shows the following SQL queries and results:

```

mysql> select * from employee;
+-----+-----+-----+-----+-----+-----+
| empid | empName | address_street | address_city | address_state | phone |
| deptid |
+-----+-----+-----+-----+-----+-----+
| 1 | John | Drive A | Richardson | Texas | 972-555-1234 |
| 10 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from jena_glt1_stmt;
Empty set (0.00 sec)

mysql>

```

Fig. 4. Addition of the first address (input as SLBN) to "John"'s record.



Fig. 5. Addition of a second address (input as SLBN) to "John"'s record.



SPARQL results:

s	p	o
_:b0	<a href="#">rdf:type</a>	<a href="#">rdf:Statement</a>
_:b0	<a href="#">rdf:object</a>	"Research"
_:b0	<a href="#">rdf:predicate</a>	<a href="#">vocab:department_deptName</a>
_:b0	<a href="#">rdf:subject</a>	<file:///home/vaibhav/employee.n3#department/10>
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_phone</a>	"972-555-1234"
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_empName</a>	"John"
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_empid</a>	1
<a href="#">db:employee/1</a>	<a href="#">rdf:type</a>	<a href="#">vocab:employee</a>
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_address_state</a>	"Texas"
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_address_city</a>	"Richardson"
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_address</a>	_:b1
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_deptid</a>	10
<a href="#">db:employee/1</a>	<a href="#">rdfs:label</a>	"employee #1"
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_address_street</a>	"Drive A"
<a href="#">db:employee/1</a>	<a href="#">vocab:employee_address</a>	_:b2
_:b1	<a href="#">vocab:employee_address_street</a>	"Drive A"
_:b1	<a href="#">vocab:employee_address_city</a>	"Richardson"
_:b1	<a href="#">vocab:employee_address_state</a>	"Texas"
<a href="#">db:department/10</a>	<a href="#">rdf:type</a>	<a href="#">vocab:department</a>
<a href="#">db:department/10</a>	<a href="#">vocab:department_deptName</a>	"Research"
<a href="#">db:department/10</a>	<a href="#">vocab:department_deptLocation</a>	"Eric Johnsson Bldg"
<a href="#">db:department/10</a>	<a href="#">vocab:department_deptid</a>	10
<a href="#">db:department/10</a>	<a href="#">rdfs:label</a>	"department #10"
_:b0	<a href="#">vocab:inspectionDate</a>	"11-09-2010"
<a href="#">rdfs:label</a>	<a href="#">rdf:type</a>	<a href="#">rdf:Property</a>

Powered by [D2R++ Server](#)

Done

Fig. 6. SPARQL query output including *inspectionDate* reification information.

```

vaibhav@vaibhav-desktop: ~
File Edit View Terminal Help
mysql> select * from employee;
+-----+-----+-----+-----+-----+-----+
| empid | empName | address_street | address_city | address_state | phone       | deptid |
+-----+-----+-----+-----+-----+-----+
| 1     | John   | Drive A       | Richardson   | Texas        | 972-555-1234 | 10     |
+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from jena_glt1_stmt;
+-----+-----+-----+-----+-----+-----+
| Subj | GraphID | Prop | Obj |
+-----+-----+-----+-----+-----+
| Uv::http://localhost:2020/resource/employee/1: | 1 | Uv::http://localhost:2020/vocab/resource/employee_address: | Bv::map:employee_address@ABC/PQR/XYZ: |
| Bv:::37c07bf6:12c37bfd74:-7ff5: | 1 | Uv::http://localhost:2020/vocab/resource/inspectionDate: | Lv:0::11-09-2010: |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from jena_glt0_reif;
+-----+-----+-----+-----+-----+-----+
| Subj | GraphID | Stmt | Prop | HasType |
+-----+-----+-----+-----+-----+
| Uv::file:///home/vaibhav/employee.n3#department/10: | 1 | Uv::http://localhost:2020/vocab/resource/department_deptName: | Lv:0::Research: | T |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>

```

Fig. 7. Addition of *inspectionDate* reification information to *Department 10*.

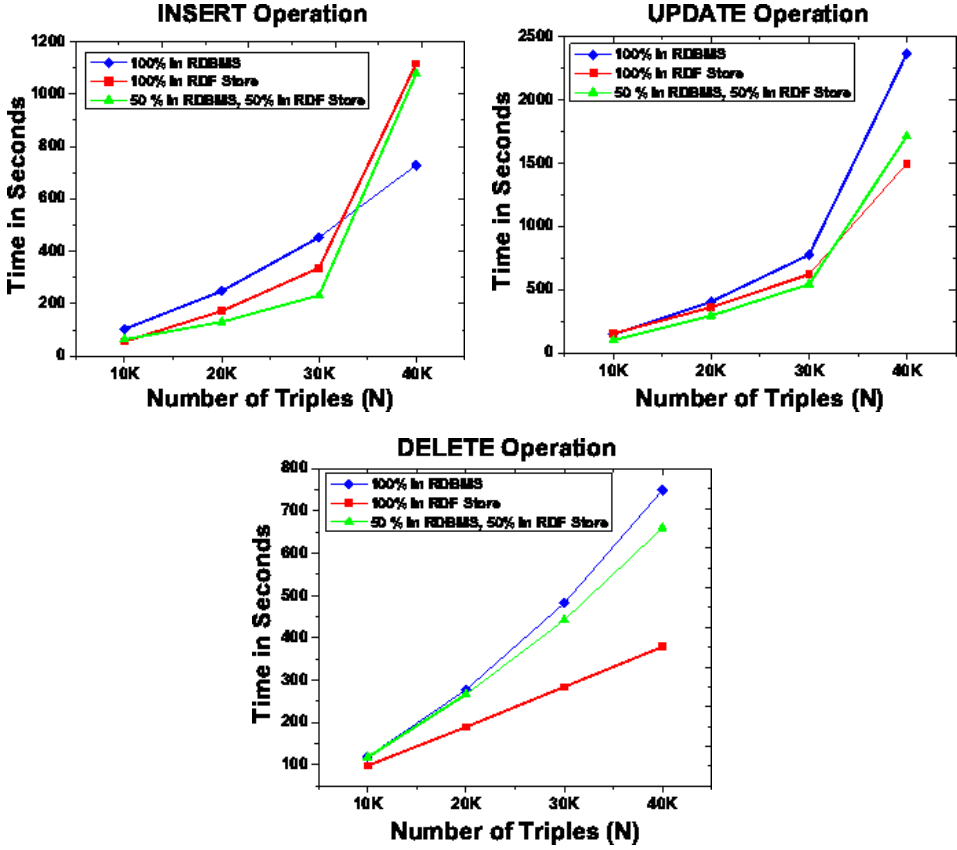


Fig. 8. Performance of DML operations.

size grows, the time taken to identify whether a particular subject or predicate exists in the RDBMS or the RDF store increases due to the increased number of records. In the case of the Delete operation, as the triples are deleted the database size decreases and, therefore, the subject and predicate searches operate on fewer triples which, in turn, reduces the search time. Thus, the performance degradation is not as severe as in the Insert scenario. In either case however, since the main purpose of D2RQ++ is to provide insert/update/delete functionality in OnLine Transaction Processing (OLTP) kind of applications with few insert/update/delete operations per transaction and not to provide bulk data loading functionality, this performance degradation with increasing data volumes is of no consequence. For the same reason, generating performance statistics for DML operations on larger datasets is also considered unnecessary and beyond D2RQ++'s objectives. It is expected that for bulk data loading, performance-optimized data loader utilities provided with the underlying relational databases will be used rather than D2RQ++.

## 7. Conclusions

This paper presented a bi-directional translational mechanism, D2RQ++, between relational databases and RDF data stores, with particular emphasis on translation procedures for RDF blank nodes and reification nodes. One of the requirements of our bi-directional translation work was to reuse and extend existing uni-directional solutions in the translational arena in order to avoid reinventing the wheel wherever possible. Thus, D2RQ, a highly popular and widely adopted uni-directional translational tool was chosen as the foundation upon which our bi-directional algorithms were to be built. D2RQ++ is, thus, essentially a wrapper around D2RQ that transforms the latter from a read-only application to a read-write application.

Future directions for D2RQ++ include improvisation of the mapping and bi-directional translation process for mixed blank nodes, which consist of both literal as well as resource objects, and incorporation of support for SPARQL/Update. At the current time, neither D2RQ's nor D2RQ++'s mapping languages support the translation of relational data such as mixed/nested blank nodes. D2RQ++ is an application that accepts triples for update into the corresponding underlying relational database. Therefore, it becomes necessary to arrive at an appropriate translation and update process for such mixed/nested blank nodes. Mapping constructs need to be designed to map, translate, and update such incoming nodes into the underlying relational database. SPARQL/Update, a companion to the original (read-only) SPARQL Query Language for RDF Stores, is a data manipulation language that enables insert, update, and delete operations on RDF graphs. The SPARQL/Update specification is currently under review with the World Wide Web Consortium and is expected to become a recommendation in due course. As a result, providing support for SPARQL/Update statements in subsequent versions of D2RQ++ is a priority. From an implementation perspective, the present version of D2RQ++ is tightly integrated with Jena and its associated API for RDF triples storage, and MySQL for housing the underlying relational schema. Future directions for D2RQ++ include providing support for various other platforms such as Sesame and HBase for RDF Storage and Oracle, DB2, Sybase, and MS SQL Server for relational data storage.

## References

- [1] B. He, M. Patel, Z. Zhang and K. C.-C. Chang, Accessing the deep Web, in *Communications of the ACM* **50** (2007) 94–101.
- [2] A. Malhotra, W3C RDB2RDF Incubator Group Report, <http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/>, 2009.
- [3] C. Bizer and A. Seaborne, D2RQ — Treating non-RDF databases as virtual RDF graphs, Poster at *3rd International Semantic Web Conference*, 2004.
- [4] O. Erling and I. Mikhailov, RDF support in the virtuoso DBMS, in S. Auer, C. Bizer, C. Müller and A. V. Zhdanova (eds.), *CSSW*, LNI, Vol. 113, 2007, pp. 59–68.
- [5] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann and D. Aumueller, Triplify: Lightweight linked data publication from relational databases, in J. Quemada, G. Le'on, Y. S. Maarek and W. Nejdl (eds.), *WWW*, 2009, pp. 621–630.

- [6] F. Cerbah, Learning highly structured semantic repositories from relational databases — RDBToOnto Tool, in 5th *European Semantic Web Conference (ESWC)*, Spain, 2008.
- [7] E. Prud'hommeaux and A. Seaborne, SPARQL query language for RDF, <http://www.w3.org/TR/rdf-sparql-query>, January 2008.
- [8] S. Ramanujam, V. Khadilkar, L. Khan, S. Seida, M. Kantarcioglu and B. Thiraisingham, Bi-directional translation of relational data into virtual RDF stores, in 4th *IEEE International Conference on Semantic Computing (ICSC)*, Pittsburg, 2010.
- [9] S. Powers, *Practical RDF* (O'Reilly Media, 2003).
- [10] L. Han, T. Finin, C. S. Parr, J. Sachs and A. Joshi, RDF123: From spreadsheets to RDF, in A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin and K. Thirunarayan (eds.), *International Semantic Web Conference*, LNCS Vol. 5318, 2008, pp. 451–466.
- [11] C. P. de Laborda and S. Conrad, Bringing relational data into the semantic web using SPARQL and relational.OWL, in R. S. Barga and X. Zhou (eds.), *ICDE Workshops*, IEEE Computer Society, 2006, p. 55.
- [12] C. P. de Laborda, M. Zloch and S. Conrad, RDQuery — Querying relational databases on-the-fly with RDF-QL, poster at 15th *International Conference on Knowledge Engineering and Knowledge Management*, 2006.
- [13] Y. An, A. Borgida and J. Mylopoulos, Discovering the semantics of relational tables through mappings, *Journal on Data Semantics* (2006), pp.1–32.
- [14] L. Chen and N. Yao, Publishing linked data from relational databases using traditional views, in 3rd *IEEE International Conference on Computer Science and Information Technology (ICCSIT)*, 6 (2010), pp. 9–12.
- [15] H.-Y. Ling and S.-F. Zhou, Translating relational databases into RDF, in *International Conference on Environmental Science and Information Application Technology (ESIAT)*, Vol. 3, 2010, pp. 464–467.
- [16] M. A. Ismail, M. Yaacob and S. A. Kareem, Integration of heterogeneous relational databases: RDF mapping approach, in *International Symposium on Information Technology (ITSim)*, Vol. 3, 2008, pp. 1–7.
- [17] J. Wang, Z. Miao, Y. Zhang and B. Zhou, Querying heterogeneous relational database using SPARQL, in 8th *IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, 2009, pp. 475–480.
- [18] J. Wang, Z. Miao, Y. Zhang and B. Zhou, Integrating heterogeneous data source using ontology, in *Journal of Software* 4 (2009), pp. 843–850.
- [19] C. P. Cheong, C. Chatwin and R. Young, A RDF-based semantic schema mapping transformation system for localized data integration, in 3rd *International Conference on Anti-Counterfeiting, Security, and Identification in Communication*, 2009, pp. 144–147.
- [20] I. Myroshnichenko and M. C. Murphy, Mapping ER schemas to OWL ontologies, in *IEEE International Conference on Semantic Computing (ICSC)*, 2009, pp. 324–329.
- [21] M. Hert, G. Reif and H. Gall, Updating relational data via SPARQL/update, in F. Daniel et al. (eds.), *EDBT/ICDT Workshops*, ACM Intl. Conference Proceeding Series, 2010.
- [22] C. Bizer, D2R MAP — A database to RDF mapping language, in *WWW (Posters)*, 2003.
- [23] A. Seaborne et al., SPARQL Update — A language for updating RDF graphs. <http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/>, 2008.

- [24] S. Ramanujam, A. Gupta, L. Khan, S. Seida and B. Thuraisingham, R2D: A framework for the relational transformation of RDF data, *Int. J. Semantic Computing* **3**(4) (2009) 471–498.
- [25] S. Ramanujam, A. Gupta, L. Khan, S. Seida and B. Thuraisingham, Relationalization of provenance data in complex RDF reification nodes, *Electronic Commerce Research Journal* **10** (2010) 389–421.
- [26] C. Bizer and R. Cyganiak, D2R Server — Publishing relational databases on the semantic web, poster at *5th International Semantic Web Conference*, 2006.
- [27] S. Ramanujam, V. Khadilkar, L. Khan, S. Seida, M. Kantarcioglu and B. Thuraisingham, Bi-directional translation of relational data into virtual RDF stores, *Technical Report UTDCS-13-10*, The University of Texas at Dallas, 2010. Available at [www.utdallas.edu/~sxr063200/D2RQ++Ver1.pdf](http://www.utdallas.edu/~sxr063200/D2RQ++Ver1.pdf).