2010 IEEE Fourth International Conference on Semantic Computing

Bi-directional Translation of Relational Data into Virtual RDF Stores

Sunitha Ramanujam*, Vaibhav Khadilkar*, Latifur Khan*, Steven Seida†,
Murat Kantarcioglu* and Bhavani Thuraisingham*
*The University of Texas at Dallas
800 W. Campbell Road
Richardson, TX 75080
†Raytheon Company
1200 Jupiter Road
Garland, TX 75042

***Abstract*—A vast majority of the world's valuable data currently exists in relational databases and other legacy storage systems. In order for Semantic Web applications to access such legacy data without replication or synchronization of the same, the gap between the two needs to be bridged. 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 the existing and new data from an underlying relational database into a corresponding virtual RDF store in a read-only manner. This paper expands on previous relational-to-RDF bridging work, by enabling the bridge to be bi-directional and allowing data updates specified as triples to be propagated back to the relational database as tuples. Algorithms to translate the triples to be updated/inserted/deleted into equivalent relational attributes/tuples whenever possible are presented. A widely embraced open-source tool called D2RQ is enhanced with these algorithms to serve as evidence of the bi-directionality of our translation process.**

***Keywords*-Semantic Web, Resource Description Framework, Relational Databases, Mapping, Data Interoperability, Data Manipulation**

I. INTRODUCTION

The ability of the World Wide Web to serve as a universal medium for the exchange of data and knowledge has increased its popularity and spawned related efforts such as the Semantic Web. The Semantic Web is a rapidly maturing initiative that is envisioned to enhance computer understandability of the subject matter of web pages through the use of meta-data rather than keywords. Further, Semantic Web technologies and specifications such as the Resource Description Framework (RDF[1]) and the Web Ontology Language (OWL[2]) provide a means to integrate disparate data sources and reuse data across applications through the use of ontologies. The above reasons, coupled with the simplicity and flexibility offered by RDF and other Semantic Web technologies, have resulted in widespread adoption of the same. On the other hand, relational databases, by virtue of having been in existence for several decades now, are the most commonly used storage solutions in production environments. Thus, in order to reap the benefits offered by Semantic Web technologies while continuing to exploit the advantages of well-established database technologies such as query optimization, data concurrency and security, transaction support, and scalability, a means to enable the two technologies to co-exist together needs to be established. The importance of this bridging problem is validated with the initiation, by the W3C, of the RDB2RDF incubator group [1] aimed at investigating the need for mapping standardization.

Several solutions to the problem of bridging Relational Database Management System (RDBMS) and RDF concepts are in existence today that are able 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 [2], [3], [4]. Thus, while one can view the relational data in RDF graph form and can query the resultant RDF triples using SPARQL [5], 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 present D2RQ++[3], an enhancement to an existing, extensively adopted relational-to-RDF read-only translation tool called D2RQ, which includes the ability to propagate data changes specified in the form of RDF triples back to the underlying relational database. Most of the current semantic infrastructures simply combine the read-only view of the relational database provided by existing tools with ontologies to infer additional knowledge. With D2RQ++ this process is taken a step further by allowing the inferred knowledge as well as newly acquired knowledge specified as triples to be updated to relational database tuples.

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

[1]http://www.w3.org/RDF/
[2]http://www.w3.org/TR/owl-features/
[3]http://cs.utdallas.edu/semanticweb

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 blank node structures to equivalent relational tuples.
- 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.

The organization of the 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 Section 3 along with the algorithms comprising D2RQ++. Section 4 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 and, lastly, Section 5 concludes the paper.

II. 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 [2] which is essentially a mapping between relational schema and OWL/RDF-Schema (RDFS[4]) 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 [3] is yet another effort that, like D2RQ, also uses a declarative meta schema consisting of quad map patterns that define the mapping of relational data to RDF ontologies. RDF123 [6], 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 [7], [8], [9] describe more mapping attempts in the reverse direction. In [7] 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 [8] to translate SPARQL queries to SQL. The authors in [9] 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. A more recent research effort in the relational-to-RDF mapping arena is Triplify [4], 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.

[4]http://www.w3.org/TR/rdf-schema/

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 [10] 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 [11], and they include support for the SPARQL/Update language [12] 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.

Another difference between ONTOACCESS and D2RQ++ is the ability to accommodate updates/deletes of blank node structures. Blank Nodes are used to represent complex relationships between entities and are an integral component of the RDF specification. ONTOACCESS makes no mention of how incoming blank node structures are handled while D2RQ++ is capable, as illustrated in Section 3, of translating a variety of blank 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, none of the existing research efforts, except one, address the issue of enabling bi-directional data transfer between relational and RDF applications. 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 the first endeavor to address the issue of bi-directionality in the relational-to-RDF mapping arena.

III. 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 Figure 1 to reinforce concepts wherever applicable in subsequent sections. (We would like to emphasize that the Figure 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 Figure 1). There were several issues that needed to be addressed in order to achieve bi-directional translation. 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.

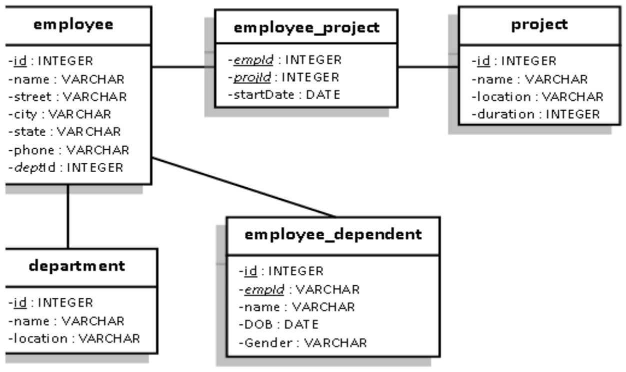


Fig. 1. Relational Schema used to illustrate D2RQ++

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 D2RQs 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 "d2rq: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 [13] 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 d2rq:{SL/CL/R}BlankNode PropertyBridge. This (last) construct is used to associate the attributes comprising a blank node with their parent blank node.

TABLE I
EXTENSIONS TO D2RQ'S MAPPING CONSTRUCTS

Construct	Description
d2rq:SimpleLiteralBlank NodePropertyBridge (SLBNPB)	Blank Nodes that have only literal objects, each with a unique predicate
d2rq:ComplexLiteralBlank NodePropertyBridge (CLBNPB)	Blank Nodes that have only literal objects; however, the predicates are not unique and include repetitions
d2rq:ResourceBlankNode PropertyBridge (RBNPB)	Blank Nodes that have only resource objects with predicates that may or may not be unique
d2rq:BelongsToBlankNode	Construct that helps link a relational attribute to the 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 D2RQs map file.

$$
\begin{aligned}
&map : employee\_address\ a\\
&\quad d2rqrw : SimpleLiteralBlankNodePropertyBridge;\\
&\quad d2rq : belongsToClassMap\ map : employee;\\
&\quad d2rq : property\ vocab : employee\_address\\
&\quad d2rq : propertyDefinitionLabel\ \text{“}employee\ address\text{”}\\
&\quad d2rq : pattern\ \text{“}@@employee.address\_street@@/\\
&\qquad\qquad @@employee.address\_city@@/\\
&\qquad\qquad @@employee.address\_state@@\text{”};\\
&\quad .
\end{aligned}
$$

$$
\begin{aligned}
&map : employee\_address\_street\ a\ d2rq : PropertyBridge;\\
&\quad d2rq : belongsToClassMap\ map : employee\_address;\\
&\quad d2rq : property\ vocab : employee\_address\_street\\
&\quad d2rq : propertyDefinitionLabel\\
&\qquad\qquad \text{“}employee\ address\_street\text{”}\\
&\quad d2rq : column\ \text{“}employee.address\_street\text{”};
\end{aligned}
$$

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 Section 3-A along with the appropriate algorithm.

The third issue we had 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 {<*empURI*> <*DeptID*> <*DepartmentID*>} arriving before the actual department triple {<*deptURI*> <*ID*> <*DepartmentID*>}. 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. This 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 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 them from the native RDF store. Both the above use cases arise since RDF knowledge stores allow violation of integrity constraints but they are not permitted by a RDBMS. The next sub-section presents the various algorithms that were developed to address and resolve the issues presented above.

A. *D2RQ++ Algorithms*

Algorithm 1 is fairly straightforward and is used on simple triples that involve a literal or resource object and that do NOT involve any blank nodes. As can be seen in the algorithm, 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.

Triples that contain blank node objects are handled using two procedures the first procedure, illustrated in Algorithm 2, deals with simple or complex literal blank nodes, and the second procedure deals with resource blank nodes.

Algorithm 1 INSERT/UPDATETRIPLE()

Input: An RDF triple
Output: A successful RDBMS/RDF Store update

```
1: Identify table assoicated with triple's subject
2: if triple is a Blank Node then
3:   Call Insert/UpdateLiteral/ResourceBlankNodeTriple
4: else
5:   if sub does not exist in RDBMS then
6:     if pred exists in sub's table in RDBMS then
7:       Insert triple as new tuple in RDBMS and return
8:     else
9:       Add triple to native RDF Store and return
10:    end if
11:  else
12:    if pred exists in subject's table in RDBMS then
13:      if (obj.isLiteral()) or (obj.isResource() and exists(obj as PK in another table)) then
14:        if RDBMS table cell value is NULL then
15:          Update triple's obj value in column and return
16:        end if
17:      end if
18:    end if
19:    Add triple to native RDF Store and return
20:  end if
21: end if
```

SLBNPBs translate into a collection of simple attributes that belong to the same table in the underlying relational schema. An example of an SLBNPB has been discussed in the previous sub-section. SLBNPBs are inserted/updated into the underlying relational database only when every attribute value comprising the SLBNPB is empty. Even if one attribute value comprising the SLBNPB has a non-null value in the relational schema (for example, if the *employee* table has, for a particular employee, a non-null value for the *state* column while the *street* and the *city* columns are null), or if the input SLBNPB has a structure that is different from the specification included in the map file (i.e., it has less/more constituent attributes than the map file specification) the entire SLBNPB structure is housed in the native RDF store instead. However, this condition (of requiring all constituent attributes to be null in order to achieve a successful insert or update activity) is a design choice adopted by us and can be changed into other suitable choices as required by the end-user of the system.

CLBNPBs represent 1:N relationships between a subject and an object concept within the underlying relational database. An example of such a relationship is the *employee_phone* relationship where an employee has 1 or more phone numbers while a phone number belongs to one and only one employee. CLBNPBs map to normalized tables (such as an *employee_phone* table with possible attributes of *empID*,

Algorithm 2 INSERT/UPDATELITERALBLANKNODE()

Input: An RDF triple with BlankNode object
Output: A successful RDBMS/RDF Store update

```
1:  if BlankNode definition not found in Map File then
2:      Add BlankNode to native RDF Store and return
3:  end if
4:  if BlankNode.Type == SLBNPB then
5:      for all every pred off of the blank node do
6:          if pred exists as column in sub table in RDBMS then
7:              if corresponding column value is NOT NULL then
8:                  Add BlankNode to native RDF Store and return
9:              end if
10:         else
11:             Add triple to native RDF store and return
12:         end if
13:     end for
14:     if sub of SLBNPB does not exist in RDBMS then
15:         Insert sub and obj of SLBNPB as a new tuple in RDBMS and return
16:     else
17:         Update the obj values of SLBNPB in corresponding columns against sub tuple and return
18:     end if
19: else
20:     if sub of CLBNPB does not exist in RDBMS then
21:         Add CLBNPB to native RDF store and return
22:     else
23:         Get RDBMS table corresponding to obj of CLBNPB
24:         for all every pred belonging to CLBNPB do
25:             Insert sub, pred off of CLBNPB, and obj of CLBNPB into RDBMS table in the foreign key, type and value fields respectively
26:         end for
27:     end if
28: end if
```

phoneType, *phoneNumber*, all of which form a combined primary key with the *empID* being a foreign key that references the parent *employee* table) that result from such 1:N relationships. CLBNPBs are inserted into the underlying relational table iff the subject of the CLBNPB exists in the parent table (In our *employee_phone* example, phone details are inserted into the *employee_phone* table iff the corresponding employee URI exists in the parent *employee* table). For every predicate belonging to the CLBNPB a new tuple is inserted into the underlying relational table with the CLBNPB subject, the predicate of the CLBNPB, and the object forming the values of the foreign key, the type, and the value columns of the relational table. In the event the subject of the CLBNPB does not exist in the parent table, the entire CLBNPB structure is housed in the native RDF store and is consolidated into the relational database by the periodic flush algorithm when the CLBNPBs subject is added into the parent table.

ResourceBlankNodePropertyBridges are blank nodes that comprise only of resource objects that may or may not belong to the same object class. In the case of RBNPBs the objects of the blank node are inserted into the underlying relational database tables iff the RBNPB subject and every one of the RBNPB object values exist as primary key values in their corresponding tables in the underlying relational schema; otherwise the entire RBNPB structure is added into the native RDF store. Two example RBNPBs are illustrated in Figure 2.

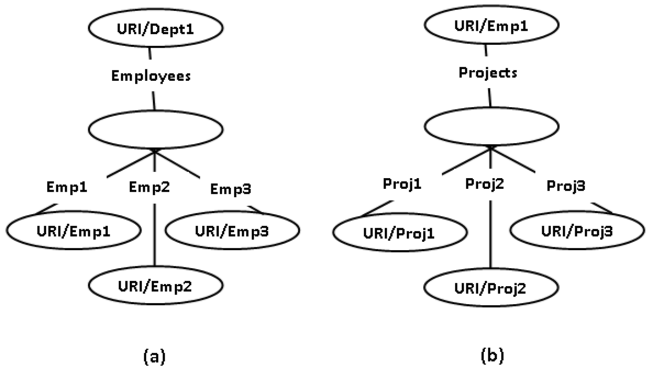


Fig. 2. Sample RBNPB Scenarios

If the RBNPB subject and object share a 1:N relationship as illustrated in Figure 2(a) between *employee* and *department* entities, for every predicate of the RBNPB, the tuple with the object value is located in the object table (i.e., the *employee* table) and the subject value (i.e., *departmentID*) is updated into the appropriate field (*deptId* in *employee* table) in the N-side table (i.e., in the *Employee* table) . If they share an N:M relationship like the one between the *employee* and *project* entities illustrated in Figure 2(b), a new tuple is inserted with the subject and object URIs as values in the respective columns (i.e., in *empID* and *projID* columns) in the join table (i.e. *employee_project* table) representing the N:M relationship. Due to space constraints the algorithm for inserting/updating resource blank nodes is omitted from this paper and can be found, instead, in our technical report [14].

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. 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

Algorithm 3 FLUSH ALGORITHM()

Input: Triples in native RDF Store

Output: Possible insert/update in RDBMS and assoicated delete from RDF store

```
1: for all triples in RDF Store do
2:   if obj.isResouce() then
3:     if exists(pred in sub table) then
4:       if exists(obj value in obj table) then
5:         if exists(sub value in sub table) then
6:           if column corresponding to obj == NULL in sub table then
7:             Update obj value in column in sub table
8:             Delete triple from native RDF Store and return
9:           end if
10:        else
11:          Insert sub and obj values in appropriate columns in sub table
12:          Delete triple from native RDF Store and return
13:        end if
14:      end if
15:    end if
16:  end if
17: end for
```

Algorithm 4 DELETETRIPLE()

Input: An RDF triple

Output: Possible successful RDBMS/RDF Store delete

```
1: if not (exists(sub)) in sub table in RDBMS then
2:   Find and delete triple from native RDF Store and return
3: else
4:   if not (exists(pred)) in sub table in RDBMS then
5:     Find and delete triple from native RDF Store and return
6:   else
7:     if sub referenced by another RDBMS table then
8:       Deny delete and return
9:     else
10:      if obj value exists in pred column in sub table then
11:        if pred column == Primary Key column (PK) then
12:          Delete from sub table where PK = sub and return
13:        else
14:          Update sub table set pred column = NULL where PK = sub and return
15:        end if
16:      else
17:        Find and delete triple from native RDF Store and return
18:      end if
19:    end if
20:  end if
21: end if
```

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.

Algorithm 4 highlights the process to delete a regular triple from either the RDBMS or the native RDF store as applicable. The only scenario in which a regular triple deletion fails is if the subject value of the triple is referenced as a foreign key by another relational database table. If either the subject or the predicate of the triple do not exist as a primary key value or column, respectively, in the underlying relational database, the triple is assumed to be housed in the native RDF store and is deleted from that store. If the column corresponding to the triple predicate has a value that differs from the triples object value, this implies that the triples object value is a secondary value for that column. In this case as well the triple is housed in the native RDF store and, hence, is deleted from the RDF store. If the column value matches the triples object value and the column corresponding to the triples predicate is not the primary key column, the tuple corresponding to the subject row is updated with a null value for the column; otherwise the tuple corresponding to the subject row is deleted completely from the underlying table. Delete procedures exist for {SL/CL/R}BNPBs as well, however space restrictions prevent us from including the details here.

Each of the algorithms described in the previous subsections 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 [15], 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.

IV. IMPLEMENTATION RESULTS

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

A. Experimental Platform

Ubuntu 9.04 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[5] 0.7 to perform the uni-directional translation of a relational database schema to an equivalent virtual RDF store, MySQL[6] 5.1.37 to store the relational database schema to be translated into an equivalent RDF store, and Jena[7] 2.6.2 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 for the development of the algorithms and procedures detailed in Section 3, and Java 1.6 for development of the Section 3's algorithms and procedures.

B. Experimental Dataset

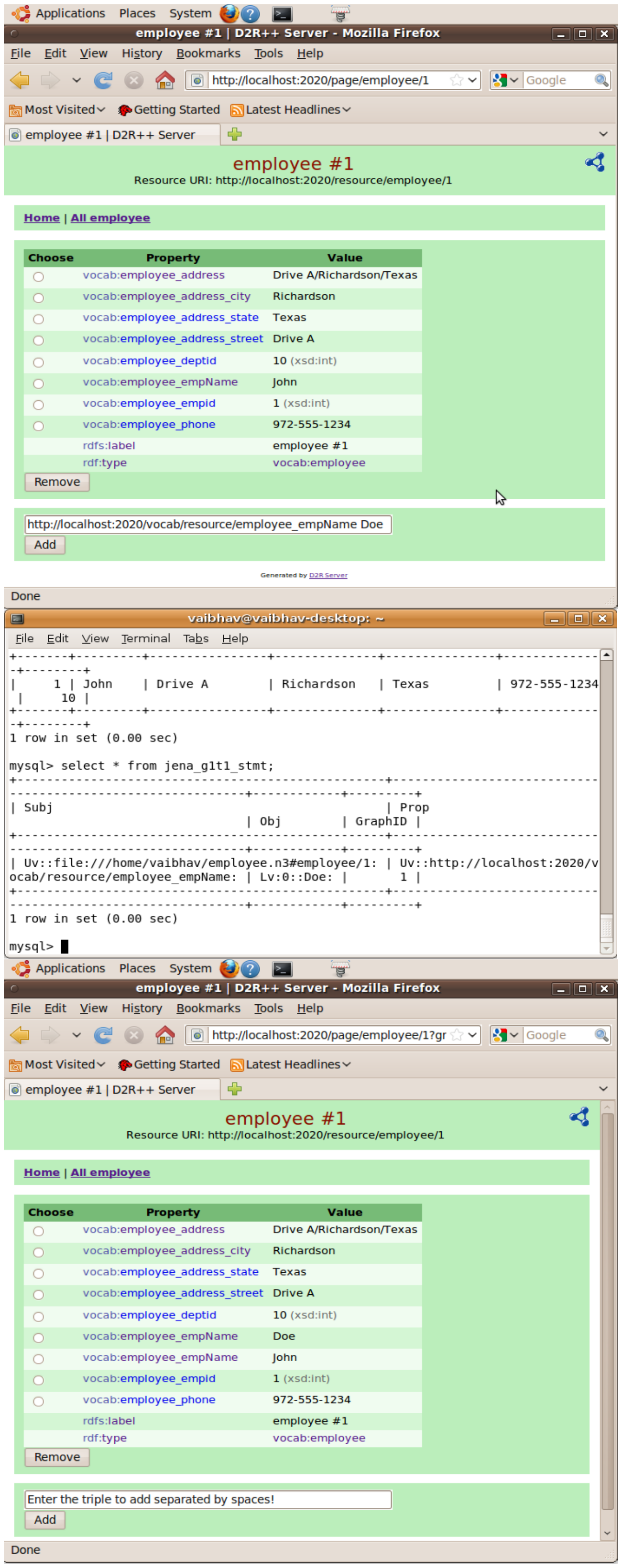
The performance experiments conducted and the D2RQ GUI outputs presented below are based on the Employee-Department-Project scenario illustrated in Figure 1. Synthetic RDF triple datasets of various sizes corresponding to the relational schema as defined by D2RQs mapping file were created through a data 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.

C. 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 D2R-Server [15] front-end GUI application to D2R++-Server which includes provisions to *add/remove* RDF triples. Figures 3 and 4 illustrate the enhanced D2R++-Server's *add/remove* extensions and D2RQ++'s ability to propagate RDF triples, as new tuples or updates to existing tuples, back to the underlying relational database or native RDF store, as applicable. The data in the *employee* table prior to any new insert/update activities included a single record for an employee named "John". Figure 3 illustrates a scenario when a duplicate value (a second *empName* for $empID = 1$) is specified for an existing column in the relational database. Under the Closed-World Assumption, this value is either rejected or replaces the original value thereby resulting in loss of information. In

[5]http://www4.wiwiss.fu-berlin.de/bizer/d2rq/
[6]http://www.mysql.com/
[7]http://jena.sourceforge.net/



http://localhost:2020/page/employee/1

employee #1 | D2R++ Server

employee #1
Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Remove

http://localhost:2020/vocab/resource/employee_empName Doe
Add

Generated by D2R Server

vaibhav@vaibhav-desktop: ~

```
+-------+--------+-----------------+----------------+----------------+------------
-+---------+
|     1 | John    | Drive A        | Richardson    | Texas          | 972-555-1234
 |      10 |
+-------+--------+-----------------+----------------+----------------+------------
-+---------+
1 row in set (0.00 sec)

mysql> select * from jena_g1t1_stmt;
+--------------------------------------------------------+--------------------------
--------------------------------+------------+---------+
| Subj                                                   | Prop
                                | Obj          | GraphID |
+--------------------------------------------------------+--------------------------
--------------------------------+------------+---------+
| Uv::file:///home/vaibhav/employee.n3#employee/1: | Uv::http://localhost:2020/v
ocab/resource/employee_empName: | Lv:0::Doe: |       1 |
+--------------------------------------------------------+--------------------------
--------------------------------+------------+---------+
1 row in set (0.00 sec)

mysql> 
```

http://localhost:2020/page/employee/1?gr

employee #1 | D2R++ Server

employee #1
Resource URI: http://localhost:2020/resource/employee/1

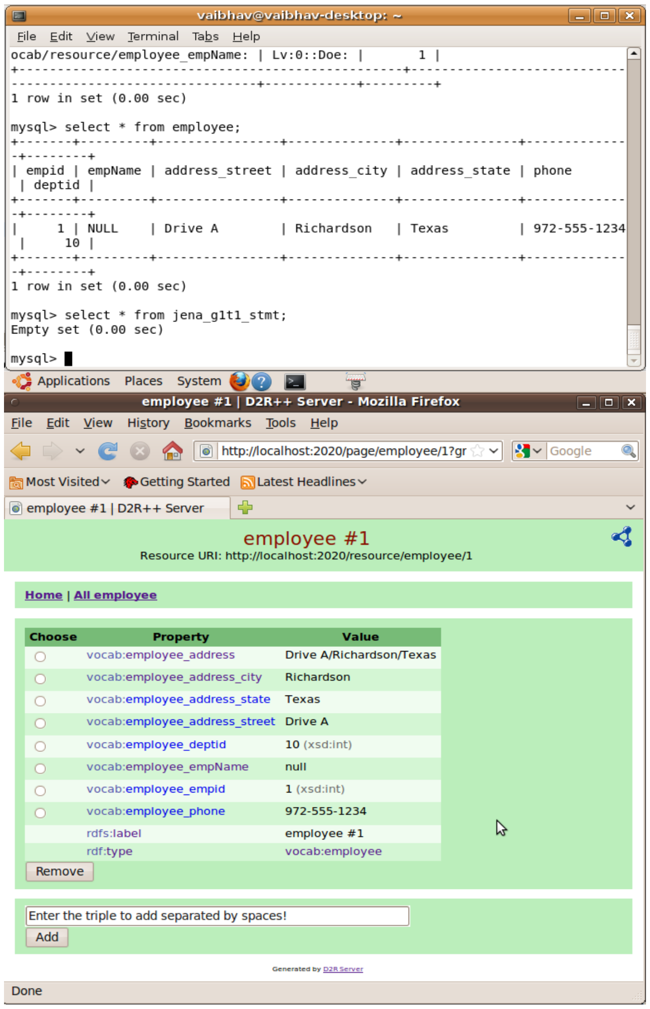
Home | All employee

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	Doe
○	vocab:employee_empName	John
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Remove

Enter the triple to add separated by spaces!
Add

Fig. 3. Addition of a Second *empName* Field Value



```
ocab/resource/employee_empName: | Lv:0::Doe: |        1 |
+-----------------------------------------------------+------------------------------
---------------------------------+-------------+---------+
1 row in set (0.00 sec)

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

mysql> select * from jena_g1t1_stmt;
Empty set (0.00 sec)

mysql> 
```

employee #1 | D2R++ Server - Mozilla Firefox

http://localhost:2020/page/employee/1?gr

employee #1

Resource URI: http://localhost:2020/resource/employee/1

Home | All employee

Choose	Property	Value
○	vocab:employee_address	Drive A/Richardson/Texas
○	vocab:employee_address_city	Richardson
○	vocab:employee_address_state	Texas
○	vocab:employee_address_street	Drive A
○	vocab:employee_deptid	10 (xsd:int)
○	vocab:employee_empName	null
○	vocab:employee_empid	1 (xsd:int)
○	vocab:employee_phone	972-555-1234
	rdfs:label	employee #1
	rdf:type	vocab:employee

Enter the triple to add separated by spaces!

Generated by D2R Server

Done

vaibhav@vaibhav-desktop: ~
File Edit View Terminal Tabs Help
Applications Places System
File Edit View History Bookmarks Tools Help
Google
Most Visited
Getting Started
Latest Headlines
employee #1 | D2R++ Server
Remove
Add

Fig. 4. Deletion of both *empName* Field Values

D2RQ++, which adheres to the Open-World Assumption, this duplicate value is housed in the native RDF store instead, thereby preserving both the original value as well as the new value as can be seen in the second front-end screenshot illustrated in Figure 3.

Figure 4 illustrates the *remove* operation which translates, wherever applicable, to either a simple *remove* operation in the native RDF store (as in the scenario where the second *empName* value of "Doe" is removed from $empID = 1$) or to an *update* (as in the scenario where the original *empName* value of "John" is removed from $empID = 1$) or *delete* operation in the underlying relational database schema (illustrated in [14]). In the first case, since the triple to be removed exists in the RDF store, it is simply removed from the store; in the second case, the triple translates into a specific row and column in the underlying relational database and, hence, the *remove* operation is propagated to the database as an update operation that sets the appropriate column value to *null*. Due to space constraints screenshots of only a few DML operations are

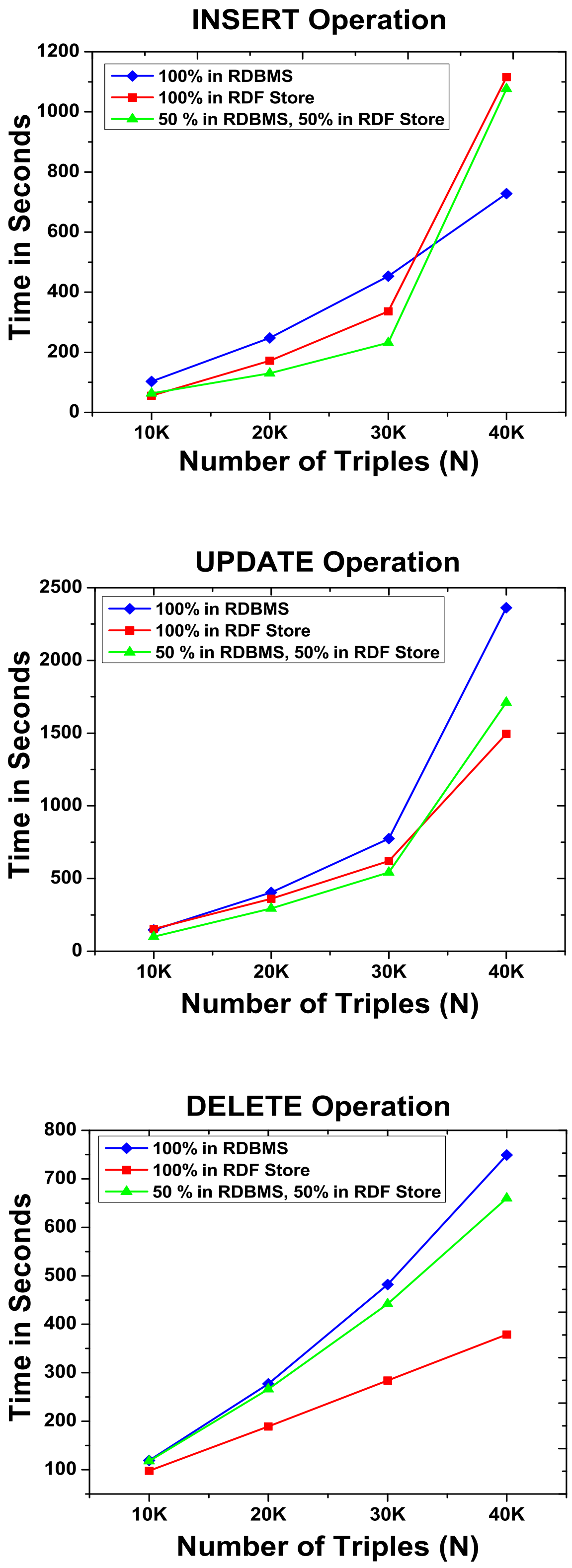


Fig. 5. Performance of DML operations

included here. More screenshots, illustrating a wider variety of insert/update/delete operations, can be found in [14].

The time taken for the insert/update/delete algorithms detailed in Section 3 is illustrated in Figure 5. 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 Figure 5, the algorithms perform well for small datasets but are somewhat time intensive for larger number of records. 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 the objectives of D2RQ++. 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++.

V. CONCLUSION

A bi-directional translational mechanism between relational databases and RDF data stores, D2RQ++, was presented in this paper. This work was motivated by a need to enable DML operations to be propagated back to the underlying relational database whenever possible and to continue to maintain the Open-World Assumption during the propagation process. 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. The various algorithms comprising D2RQ++ were presented and the feasibility of the proposed framework was demonstrated through a variety of experimental results in the form of screenshots and performance graphs.

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 addition of update-aware translation support for the RDF concept of reification.

REFERENCES

[1] A. Malhotra. W3C RDB2RDF Incubator Group Report. http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/, 2009.

[2] C. Bizer and A. Seaborne. D2RQ - Treating Non-RDF Databases as Virtual RDF Graphs. Poster at 3^{rd} International Semantic Web Conference, 2004.

[3] O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In S. Auer, C. Bizer, C. Müller, and A. V. Zhdanova, editors, *CSSW*, volume 113 of *LNI*, pages 59–68. GI, 2007.

[4] S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: light-weight linked data publication from relational databases. In J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, editors, *WWW*, pages 621–630. ACM, 2009.

[5] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query, January 2008.

[6] 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, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 451–466. Springer, 2008.

[7] C. P. de Laborda and S. Conrad. Bringing Relational Data into the SemanticWeb using SPARQL and Relational.OWL. In R. S. Barga and X. Zhou, editors, *ICDE Workshops*, page 55. IEEE Computer Society, 2006.

[8] C. P. de Laborda, M. Zloch, and S. Conrad. RDQuery - Querying Relational Databases on-the-fly with RDF-QL. Poster at 15^{th} International Conference on Knowledge Engineering and Knowledge Management, 2006.

[9] Y. An, A. Borgida, and J. Mylopoulos. Discovering the Semantics of Relational Tables Through Mappings. 4244:1–32, 2006.

[10] M. Hert, G. Reif, and H. Gall. Updating relational data via SPARQL/update. In F. Daniel, L. M. L. Delcambre, F. Fotouhi, I. Garrigós, G. Guerrini, Jose-Norberto Mazón, M. Mesiti, S. Müller-Feuerstein, J. Trujillo, T. Marius Truta, B. Volz, E. Waller, L. Xiong, and E. Zimányi, editors, *EDBT/ICDT Workshops*, ACM International Conference Proceeding Series. ACM, 2010.

[11] C. Bizer. D2R MAP - A Database to RDF Mapping Language. In *WWW (Posters)*, 2003.

[12] A. Seaborne et al. SPARQL Update - A Language for Updating RDF Graphs. http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/, 2008.

[13] 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):471–498, 2009.

[14] 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.

[15] C. Bizer and R. Cyganiak. D2R Server - Publishing Relational Databases on the Semantic Web. Poster at 5^{th} International Semantic Web Conference, 2006.