R2D: A Bridge between the Semantic Web and Relational Visualization Tools

Sunitha Ramanujam¹, Anubha Gupta¹, Latifur Khan¹, Steven Seida², Bhavani Thuraisingham¹
¹The University of Texas at Dallas
Richardson, Texas, U.S.A.
{sxx063200, axg089100, lkhan, bxt043000}@utdallas.edu
²Raytheon Corporation
Garland, Texas, U.S.A.
steven_b_seida@raytheon.com

Abstract – The widespread deployment of Resource Description Framework has resulted in the emergence of a new data storage paradigm, the RDF Graph Model, which, in turn, requires a rich suite of modeling and visualization tools to aid with data management. This paper presents R2D (RDF-to-Database), an effort whose goal is to enable reusability of relational tools on RDF data. R2D aims to transform RDF data, at run-time, into an equivalent normalized relational schema, thereby bridging the gap between RDF and RDBMS concepts and making the abundance of existing relational tools available to RDF Stores. The work in this paper extends our earlier work by including the ability to map blank nodes, which are used to represent complex relationships between entities, and to perform pattern matching and aggregation functions on data. The R2D system architecture and mapping constructs, with particular emphasis on blank node handling, are presented along with descriptions of the algorithms comprising R2D. Performance graphs and screen-shots of a relational visualization tool that uses R2D to access RDF data are presented as evidence of the feasibility of our research.

Keywords: Semantic Web, Resource Description Framework, Relational Databases, Data Interoperability

I. INTRODUCTION

In today’s increasingly networked world, the need to augment human reasoning has kicked off the Semantic Web initiative, for which various standards are being developed. One such standard, the Resource Description Framework [1], is the current buzzword in the Semantic Web Community and the focus of the work in this paper. RDF’s simplicity and suitability to unstructured and semi-structured data that is typically available on the web have increased the demand for data stores that use the RDF Graph data model and offer the ability to store and query RDF data [2].

The growing number of RDF stores have, as with any data store with massive amounts of information, spawned an associated requirement of tools for the management and visualization of this data. However, most of the current data modeling, visualization, and business intelligence tools that are widely available in the market today are still based on the more mature relational models [3]. Further, small and medium-sized organizations that are resource constrained may not have the ability or inclination to take risks associated with investing in fledging technologies such as RDF and the tools for the same [4]. In order to avoid the learning curves associated with new tools and continue to leverage the advantages offered by traditional relational tools without losing out on the benefits offered by the newer web technologies and standards, the gap between the two needs to be bridged.

The motivation behind our research is to arrive at a solution to the bridging problem without the need to create an actual physical relational schema and duplicate/synchronize data. Our approach, called R2D (RDF-to-Database), provides a relational interface to data stored in the form of RDF triples. R2D, which is a relational wrapper around RDF data stores, is a bridge that hopes to enable existing relational tools to work seamlessly with RDF Stores without having to make extensive modifications or waste valuable resources by replicating data unnecessarily. This paper elaborates on [5] and extends the work in [6] by including the ability to handle blank nodes and RDF container objects. Blank nodes are nodes that are neither URI references nor literals and are typically used to associate a resource with a set of properties that together represent complex data. They are a vital component of RDF graphs and their relationalization is the primary focus of this paper. The paper also discusses enhancements to the SQL-to-SPARQL transformation that now permit pattern matching and aggregation on RDF data. Our contributions in this paper are:

- We propose a mapping scheme for the translation of RDF Graph structures to an equivalent normalized relational schema that extends the work in [6] by including the ability to process blank nodes and RDF Container objects.
- Based on the mapping file created, we propose a transformation process that presents, at run-time, a normalized, non-generic, domain-specific, virtual relational schema view of the given RDF store. The algorithm in [6] is extended through the addition of normalization rules for different blank node scenarios.
- We propose a mechanism, which now includes pattern matching and aggregation facilities, to transform any relational SQL queries issued against the virtual relational schema into the SPARQL equivalent, and return triples data to end-users in a tabular format.
- The proposed framework imposes no restrictions on the nature of RDF triples or their storage mechanisms as it is a purely virtual layer that does not involve duplication of the RDF data. Hence, data updates are immediately visible through R2D without explicit synchronization activities.
Lastly, we provide a JDBC interface that includes all of the above functionalities and that can be plugged seamlessly into existing visualization tools.

The organization of this paper is as follows. Section II presents a brief overview of related work. Section III discusses R2D’s system architecture, mapping preliminaries, and types of relationships handled in R2D. The key algorithms comprising the R2D Framework are presented in Section IV. Section V highlights the implementation details with sample visualization screenshots and performance graphs for the map file generation process and for a diverse range of queries, and lastly, section VI concludes the paper.

II. RELATED WORK

The objective of R2D is unique and has no comparable counterparts. However, several research efforts to bring relational database concepts and semantic web concepts together exist, albeit from a perspective that is opposite to that considered in our work. These include D2RQ [7] and Virtuoso RDF Views[8], which are essentially mapping efforts that take a relational schema as input and present an RDF interface of the same as output. RDF123 [9], an open source translation tool, also uses a mapping concept, however its domain is spreadsheet data. Triplify [10] 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.

One research whose objectives are very closely aligned with ours is the RDF2RDB project [3]. Like in R2D, the authors in [3] attempt to arrive at a domain-specific, meaningful relational schema equivalent for an RDF store, however, RDF2RDB involves data replication with the triples data being dumped into a relational schema, and therefore is subject to the synchronization and space issues discussed previously. Moreover, for successful mapping, RDF2RDB requires the presence of ontological information in the form of schema definitions such as rdfs:class and rdf:property. R2D, on the other hand, can arrive at mapping details with or without explicit ontology information. Furthermore, the relational mapping in [3] involves the creation of a table for each property in the RDF graph regardless of the cardinality of the relationship represented by the property. As a result, the resulting schema may not be truly normalized and may contain more tables than necessary due to the presence of properties representing 1:N or N:1 types of relationships. R2D avoids these unnecessary tables by taking such conditions into consideration. The authors in [3] also do not discuss the details of how blank nodes are handled by their research, if at all. Lastly, since RDF2RDB involves creation of an actual physical relational schema with the RDF data duplicated into the same, there is no SQL-to-SPARQL conversion component. Since R2D performs a virtual conversion at run-time the SQL-to-SPARQL transformation process is an integral component of the same and is, to the best of our knowledge, the first of its kind. The Hybrid model presented in [11] is another mapping methodology that is similar to [3] in terms of relational schema generation and, hence, has the same drawbacks as [3].

The query processing component of R2D which comprises the SQL-to-SPARQL transformation process, once again, has no comparable counterpart while many efforts, [12, 13, 14], are underway in the other direction, namely, SPARQL-to-SQL conversion. The authors in [12] discuss a translation methodology that supports integration of heterogeneous relational databases using the RDF model. An SQL-based RDF Querying Scheme is presented in [13] where the RDF querying capability is made a part of the SQL; however, the RDF data is stored in a single database table. In [14], the authors partition the RDF graph data to store sub-graph information with the objective of reducing join costs and improving query performance.

From the above discussions, it is apparent that none of the research efforts address the issue of enabling relational applications to access RDF data without data replication. Therefore, we believe R2D makes a vital contribution to the data interoperability arena.

III. R2D PRELIMINARIES

R2D’s system architecture is illustrated in Figure 1. The work presented in this research focuses on presenting, through a JDBC Interface, a tabular equivalent of the RDF triples database to the visualization tools, and on an SQL Interface that generates SPARQL versions of SQL queries and passes the same to the Query Engine layer for processing and RDF data retrieval.

The RDF Store at the bottom of Figure 1 is examined by the RDFMapFileGenerator Algorithm (Item A in Figure 1) and an RDF-to-RelationalSchema mapping file is generated by the same using the constructs discussed in Section III (A). The DBSchemaGenerator Algorithm (Item B in Figure 1) takes this mapping file as input and presents to the relational visualization tool a domain-specific, virtual relational schema corresponding to the RDF store. Alternatively, users of the visualization tool can choose to issue SQL queries against the virtual relational schema to access the RDF data. At this point R2D’s SQL-to-SPARQL Translation Algorithm (Item C in Figure 1) performs the necessary query translations, invokes the SPARQL query
engine, and returns the results to the visualization tool in a tabular format.

At the heart of the RDF-to-Database transformation is the R2D mapping language – a declarative language that expresses the mappings between RDF Graph constructs and relational database schema constructs. Figure 2 illustrates a sample scenario from which examples are used, wherever applicable, to augment the subsequent discussions on R2D constructs.

A. R2D Mapping Constructs

This section discusses R2D constructs specific to blank nodes and their handling. Details on non-blank-node-specific constructs such as r2d:TableMap, r2d:MultiValuedPredicate, and r2d:[MultiValued]ColumnBridge can be found in [5, 6].

r2d:SimpleLiteralBlankNode (SLBN): SLBNs help relate RDF Graph blank nodes that consist purely of distinct simple literal objects to relational database columns. 

Example: The object of the “Name” predicate in Figure 2 is an example of an SLBN which has distinct literal predicates of “First”, “Middle”, and “Last”, which are, in turn, translated into columns of the same names in the “Employee” r2d:TableMap.

r2d:MultiValuedSimpleLiteralBlankNode (MVSBN): This construct maps duplicate SLBNs and, while the processing of the predicates is identical to the (SingleValued) SLBN, this construct results in the generation of a separate r2d:TableMap with a foreign key relationships to the table representing the subject resource of the blank node. In the event the predicates leading to the blank nodes are distinct, an r2d:MultiValuedPredicate (MVP) is created and a “TYPE” column corresponding to the MVP is included in the r2d:TableMap. Example: The objects of the “HomeAddress” and the “WorkAddress” predicates in Figure 2 together form an MVSBN.

r2d:ComplexLiteralBlankNode (CLBN): This construct refers to blank nodes in the RDF Graph that have multiple literal object values for the same subject and the predicate concept associated with the blank node. An r2d:ComplexLiteralBlankNode typically results in the generation of a separate r2d:TableMap with a foreign key relationship to the table representing the subject resource of the blank node.

Example: The object of the “Phone” predicate in Figure 2 is an example of a CLBN that has multiple object (<Cell>) values for the subject (URI/EmpA) and a predicate (Cell) concept associated with the blank node.

r2d:MultiValuedComplexLiteralBlankNode (MVCLBN): This construct maps duplicate complex literal blank nodes and the processing of the predicates is identical to the (SingleValued) CLBN case except in the event the predicates leading to the blank nodes are distinct, in which case an r2d:MultiValuedPredicate (MVP) is created and a “TYPE” column corresponding to the MVP is included in the r2d:TableMap. Example: Consider a scenario where the “Phone” predicate in Figure 2 is replaced with two similar predicates, “PastPhNums” and “CurrentPhNums”, each of which are CLBNs. The objects of these two predicates together form an MVCLBN.

r2d:SimpleResourceBlankNode (SRBN): This construct helps map blank nodes that have multiple predicates leading to resource objects belonging to the same object class. SRBNs typically identify N:1 or N:M relationships between the subject resource and the object resource classes. R2D containers that represent collections of similar resource objects are represented using the SRBN construct. Example: The object of the “Projects” predicate in Figure 2 is an example of a SRBN that has multiple resource objects that are instances of the “Project” class r2d:TableMap.

r2d:ComplexResourceBlankNode (CRBN): CRBNs represent blank nodes that have distinct or non-distinct predicates leading to objects belonging to different object classes. This construct also identifies N:1 or N:M relationships between the subject resource class and each of the object classes and typically result in the creation of as many join tables as the number of distinct object classes leading off of the CRBN. RDF containers that represent collections of different types of object resources are represented using CRBNs. Example: The object of the “OtherActivities” predicate is an example of a CRBN that has multiple resource objects each of which is an instance of a different (one “Course” and one “Training”) class.

r2d:MultiValuedSimple/ComplexResourceBlankNode (MVSBN and MVRBN): Duplicate simple/complex resource blank nodes are represented using the MVSBN and MVRBN constructs respectively. Like other MultiValued constructs, the processing for these is also identical to their SingleValued counterparts except in the event the predicates leading to the blank nodes are distinct, in which case an r2d:MultiValuedPredicate (MVP) is created and a “TYPE” column corresponding to the MVP is included in the r2d:TableMap. Example: Consider a scenario where the “Projects” predicate in Figure 2 is replaced with two similar predicates, “PastProjects” and “CurrentProjects”, each of which are SRBNs. The objects of these two predicates together form an MVRBN.

r2d:MixedBlankNode: Blank Nodes consisting of a mixture of literal, resource, and other blank node objects are mapped using the r2d:MixedBlankNode construct. This construct results in the creation of a r2d:TableMap as described in Table 1.
The mapping constructs specific to single-valued and multi-valued column bridges are applicable to blank nodes as well and are discussed in [6]. The virtual relational schema generated by R2D for the sample scenario in Figure 2 is illustrated in Figure 3 and the schema generation details are discussed in Section IV (B).

Figure 3: Equivalent Relational Schema for the Sample Scenario in Figure 2

B. Types of Blank Nodes and Relationships

Table 1 summarizes the blank node constructs that are provided by R2D and the RDBMS relationships corresponding to them in the virtual relational schemata. It also provides appropriate examples from Figure 2 wherever applicable. RDBMS relationships corresponding to non-blank-node entities in the RDF graph can be found in our earlier work in [6].

### Table 1. Mapping Between R2D and RDBMS Terms

<table>
<thead>
<tr>
<th>R2D Constructs</th>
<th>RDBMS Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2d:ColumnBridge</td>
<td>Column (Example: <code>&lt;Nickname&gt;</code>)</td>
</tr>
<tr>
<td>r2d:SimpleLiteral BlankNode</td>
<td>Column (Ex: <code>&lt;First&gt;</code>, <code>&lt;Middle&gt;</code>, <code>&lt;Last&gt;</code>)</td>
</tr>
<tr>
<td>r2d:Complex LiteralBlankNode</td>
<td>Multi Valued Attribute (resulting in a new table that includes a TYPE column)</td>
</tr>
<tr>
<td>r2d:[Simple/Complex] ResourceBlank Node (Ex: <code>&lt;Phone&gt;</code>)</td>
<td>Primary-Key/Foreign-Key relationship. Either a Column in parent table (1:N relationship) or a Column in a new join table (N:M relationship)</td>
</tr>
<tr>
<td>r2d:MultiValued {Simple/Complex} (LITERAL/Resource) BlankNode (Ex: Home/WorkAddress)</td>
<td>If no references to other table – Multi-valued Attribute (resulting in new table for 1:N relationship); Else Column in a new join table (N:M relationship)</td>
</tr>
<tr>
<td>r2d:MultiValued MixedBlankNode</td>
<td>Multi-valued Attribute (results in the creation of a r2d:TableMap which contains as fields every literal or resource leaf node object that is an element of the tree rooted at the r2d:MultiValuedBlankNode)</td>
</tr>
<tr>
<td>r2d:refersToTableMap</td>
<td>Foreign Key (Ex: <code>&lt;DeptID&gt;</code>)</td>
</tr>
<tr>
<td>r2d:MultiValued Predicate</td>
<td>&quot;Type&quot; column in parent table (Ex: <code>Phone_Type</code> for <code>&lt;Phone&gt;</code>)</td>
</tr>
</tbody>
</table>

IV. R2D: A PROTOTYPE DESIGN

In keeping with the objectives of this research, several RDF-to-RDBMS bridging algorithms were designed and developed in addition to the design of the RDF-to-Relational mapping language discussed in Section III. The following subsections discuss these algorithms.

A. RDFMapFileGenerator

The first step in the R2D Framework is map file generation realized through the RDFMapFileGenerator algorithm that automatically generates an RDF-to-Relational mapping file through extensive examination of RDF data.

Table 2 lists the relationship between some key OWL/RDFS Ontology terminologies and R2D constructs to relational concepts.

### Table 2. RDFS/OWL V/S R2D: NOTIONAL MAPPING

<table>
<thead>
<tr>
<th>OWL/RDFS Constructs</th>
<th>RELATIONAL CONCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdfs:class</td>
<td>Table</td>
</tr>
<tr>
<td>rdf:property</td>
<td>Column</td>
</tr>
<tr>
<td>rdf:domain</td>
<td>Table that the rdf:property is a column of</td>
</tr>
<tr>
<td>rdf:range</td>
<td>Datatype of the column</td>
</tr>
<tr>
<td>rdf:type predicate</td>
<td>Values of Primary Key column of the table</td>
</tr>
</tbody>
</table>

However, the transformation process is not always as straightforward or well-defined as Table 2 suggests. There are currently many RDF Graphs in existence that either do not have any, or have incomplete structural information included along with the data. RDFMapFileGenerator works on RDF Stores with or without such structural information. A high-level discussion of the algorithm is provided below.

The data structure discovery process is as follows. When structural information about the RDF database is available, the algorithm discovers schema definitions and creates appropriate Table and Column structures as per the mappings in Table 2. Next, instance data is processed, using three procedures, to identify and account for those predicates that may not have been defined through explicit rdf:property definitions.

The first procedure, ProcessLiteralPredicate, is used to process predicates that have literal objects. For every literal predicate that does not have a column corresponding to itself, a new column is added to the TableMap corresponding to the resource to which the predicate belongs. If the resource contains more than one such predicate (i.e. the resource contains multiple literal object
values for the same predicate), then the column type of the corresponding column is set to r2d:MultiValuedColumnBridge, otherwise it is a simple r2d:ColumnBridge.

The second procedure, ProcessResourcePredicate, handles predicates that have resource objects. A new potential column is added for every resource predicate that belongs to the subject resource. After all resource predicates are processed duplicate predicates (i.e., predicates that have objects belonging to the same object class) are examined and eliminated. During this consolidation process, any potential columns that refer to the same object resource class are combined and set to r2d:MultiValuedColumnBridges while columns referring to distinct object resource classes are set to r2d:ColumnBridge. This consolidation is mandatory in order to arrive at a normalized and logically sound relational schema. In cases where the objects belong to the same object class but the predicates have distinct predicate names, a MultiValuedPredicate object is created which reflects this fact. These MultiValuedPredicates typically become “TYPE” fields in the corresponding r2d:TableMaps in the relational schema.

Blank node predicates, handled through the third procedure, ProcessBlankNode, are processed and classified into the categories described in Section III (A) depending on whether the blank node objects are literals, resources, blank nodes, or a combination of the same. If every predicate off of the blank node contains a literal object (such as the Name and Phone blank nodes in Figure 2) then, for each predicate off of the blank Node, the ProcessLiteralPredicate procedure is called which works as described earlier. If every column generated through the ProcessLiteralPredicate procedure is a simple r2d: ColumnBridge (such as the Name blank node) then the BlankNode is set to r2d:SimpleLiteralBlankNode. If any of the columns are r2d:MultiValuedColumnBridges (such as the Phone blank node) then the BlankNode is set to r2d:ComplexLiteralBlankNode. If no such blank node has been previously encountered, this blank node is added to the set of blank nodes. If a similar blank node is already an element of the set of blank nodes, the blank node type is set to r2d:MultiValuedSimpleLiteralBlankNode (such as the blank nodes corresponding to the HomeAddress and WorkAddress predicates in Figure 2) or r2d:MultiValuedComplexLiteralBlankNode respectively.

In case of blank nodes that contain only resource objects, every predicate off of such blank nodes is processed using the ProcessResourcePredicate procedure, also discussed above. As before, the consolidation process is carried out after all predicates off of the blank nodes are processed. If the number of consolidated columns is equal to 1 (such as in the case of the Projects blank node), the blank node type is set to r2d:SimpleResourceBlankNode, otherwise (as in the case of the OtherActivities blank node) it is set to r2d:ComplexResourceBlankNode. As in the previous case, if a similar blank node exists, the blank node type is set to r2d:MultiValuedSimpleResourceBlankNode or r2d:MultiValuedComplexResourceBlankNode respectively; otherwise, the blank node is added to the set.

Blank nodes that contain a mixture of literal objects, resource objects, and other blank nodes, are considered to be of type r2d:MixedBlankNodes and they are processed using the Depth-First-Search graph algorithm. The topmost blank node is considered the root of the tree and the procedure is as follows. For every literal or resource predicate off of a blank node, a column is created and added to the blank node entity. Additionally, for every blank node predicate off of a blank node, a new Blank Node entity is created and added to an array of blank nodes and is also added as a column to the original blank node. This way, the hierarchy of the tree rooted at the topmost blank node is maintained. This hierarchy is required during the SQL-to-SPARQL conversion to retrieve data associated with blank nodes appropriately.

Every unique processed blank node is added to the set of blank nodes for further processing by the DBSchemaGenerator algorithm described next.

B. DBSchemaGenerator

The map file generation process is followed by the actual relational schema generation process which is the next stage in the R2D process and is achieved using the DBSchemaGenerator algorithm. This algorithm takes the RDF-to-Relational Schema mapping file generated in Section IV (A) and returns a virtual, appropriately normalized relational database schema consisting of entities/tables and the relationships between them. A description of the algorithm follows.

[6] describes how entries of type r2d:TableMap, r2d:ColumnBridge, r2d:MultiValuedColumnBridge, and r2d:MultiValuedPredicate are handled.

The processing of non-nested blank nodes of various kinds is as follows. For SLBNs (such as the blank node object of the Name predicate) every r2d:ColumnBridge entry that belongs to the blank node is simply added as a column to the table to which the SLBN belongs (as indicated by the r2d:belongsToTableMap construct for the blank node). Blank nodes of type CLBN (such as the object of the Phone predicate) result in the creation of a new table that represents a 1:N relationship between the subject and the objects of the blank node. In addition, CLBN tables invariably include a “TYPE” column associated with the r2d:MultiValuedPredicate that is typically a part of the blank node. Entries of type SRBNs and CRBNs (such as objects of the Projects and OtherActivities predicates respectively) typically result in creation of join tables with the primary keys of tables corresponding to the subject resource and the object resource included as fields in the join table. Further, if the predicates corresponding to the column bridges belonging to these blank nodes are MultiValued, an additional “TYPE” column is created and added to the join table.

The processing of MVSLBN results in the creation of a new table, contrary to the SLBN scenario. This table has as columns the primary key of the table corresponding to the blank node’s r2d:belongsToTableMap value, and all the r2d:ColumnBridges that belong to the MVSLBN. The processing of MVCLBN and r2d:MultiValued
{Simple/Complex}ResourceBlankNode is very similar to their SingleValued counterparts with the only difference being the inclusion of an additional field in the event the predicate corresponding to the blank node is an “MVP”.

Blank nodes of type r2d:MixedBlankNode result in tables which have as columns the primary key column of the table corresponding to the r2d:belongsToTableMap construct of the topmost blank node, and the literal and resource objects that are at the leaf nodes of the tree rooted at the topmost mixed/nested blank node. These leaf nodes are discovered using a recursive procedure which explores the predicates in a depth-first manner.

C. SQL-to-SPARQL Translation

This algorithm corresponds to the final phase of the R2D transformation process where the SQL-to-SPARQL translation is performed. The algorithm, which takes an SQL Statement as input and returns an appropriate SPARQL equivalent as output, is an enhancement over the work in [6] with functionalities added to process queries involving underlying blank nodes, and to provide pattern matching and data aggregation abilities. The algorithmic details follow.

First, the input query is parsed to identify the tables, fields, and Where and Group By clauses, if present. The parsed query is then transformed into its SPARQL form and executed. Any data aggregation is achieved by appending an ORDER BY clause to the transformed SPARQL query. The actual group functions are calculated on the data obtained through the execution of the appended SPARQL query and the aggregated results are returned to the relational tool in a tabular format. In order to better understand the transformation procedure let us consider the following query based on the sample scenario illustrated in Figure 2 in Section III.

```
SELECT Name_First, Name_Last, Phone_Value, department_name FROM employee, employee_Phone, phone WHERE employee.employee_PK = employee_Phone.employee_PK and employee_Phone.Phone_Type = 'Cell' and employee.department_id = department.department_id AND (name_First LIKE 'ABC%' OR employee_ppk = 'null');
```

The SPARQL SELECT is generated by adding a variable for every field (including aggregated fields, and fields in the SQL WHERE clauses) in the SQL SELECT list. After this step the SPARQL SELECT list for our example is as follows:

```
SparqlSELECT = SELECT ?name_First ?name_Last ?Phone_Value ?department_name
```

The SQL WHERE clauses are added, with minor modifications, to the FILTER clause of the SPARQL statement. If the field in the SQL WHERE clause is a primary key, the field name is replaced with the “?subject<tableIndex>” variable where tableIndex corresponds to the table, or the parent table in the case of derived tables (corresponding to blank nodes) to which the field belongs. WHERE clauses involving non-primary key fields are added directly to the SPARQL FILTER clause. In the case of the LIKE operator, the value on the right-hand-side is converted to an equivalent regular expression construct (by appropriately using the “\%”, “\”, “.”, and “\.” special characters in place of the “\%” and “\?” characters used in the LIKE expression) and the “regex” function is used on this converted expression in the FILTER clause.

Upon completion of the SQL WHERE clause processing, the FILTER clause for our example is:

```
SparqlFILTER = FILTER (?Phone_Type = 'Cell' && employee_employee_pk = employee_Phone_employee_PK)
```

The WHERE clause corresponding to employee.employee_PK = employee_Phone.employee_PK is eliminated in the SPARQL equivalent since employee_Phone is a derived table corresponding to the employee resource itself. Further, since the primary key field refers to the subject resource, the primary key fields associated with the employee and department tables are replaced with the corresponding ?subjecti variables where i is the unique tableIndex associated with the tables to which the primary keys belong.

The SPARQL WHERE clause is generated as follows. For non-derived tables and derived tables corresponding to multi-attributes clauses of the form ?subject<tableIndex> <Field.Predicate> ?<Field.Name> are added for every field in the table. For derived tables corresponding to blank nodes and for fields belonging indirectly to non-derived tables (i.e. SimpleLiteralBlankNode fields), clauses of the form ?subject<tableIndex> <BlankNode.Predicate> ?<BlankNode.Name> and ?<BlankNode.Name> <Field.Predicate> ?<Field.Name> are added to the SPARQL WHERE clause. The SPARQL WHERE clause after the processing of predicates associated with the non-derived table, Department, and the processing of fields belonging indirectly to the non-derived table, Employee (caused by the SimpleLiteralBlankNode corresponding to the multi-valued attribute, Name), is as follows:

```
SparqlWHERE = WHERE { ?subject0 <employee/Name> ?employee_name.
employee_Name <employee/First> ?name_First.
employee_Name <employee/Last> ?name_Last.
?subject0 <department/department_id> ?employee_department_id.
?subject1 <department/employee_department_id> ?employee_department_id.}
```

Since a field cannot be specified in the FILTER clause without being a part of the SPARQL WHERE clause, the field employee_department_id is added to the SPARQL WHERE clause above despite not being a part of the SPARQL or SQL SELECT list.

For derived tables corresponding to multi-valued attributes or non-mixed blank nodes that contain multi-valued predicates, such as EmployeePhone, SPARQL where clauses of the form

```
?subject<tableIndex> ?<MVPColumn.Name>
?<NonMVPColumn.Name> and
?<BlankNode.Name> ?<MVPColumn.Name>
?<NonMVPColumn.Name>
```

are added, respectively. Further, for every predicate belonging to the multi-valued predicate field, a clause of the form ?MVPColumn.Name = <PredicateName> is added to the SPARQL FILTER clause. The processing of predicates associated with the derived table, Employee_Phone, containing a multi-valued predicate column called
Phone_Type results in the following additions to the SPARQL WHERE clause:
SparqlWHERE = SparqlWHERE + ?subject0 http://empl/Phone ?employee_Phone.
?employee_Phone ?Phone_Type ?Phone_Value.

Lastly, in the case of mixed blank nodes, for each field belonging to the mixed blank node table, the sequence of predicates leading from the topmost subject (of the mixed blank node) to the field are obtained by traversing the tree structure stored during the MapFileGeneration process and a Where clause is added to the SPARQL WHERE for each of the predicates in the sequence.

The SPARQL WHERE and FILTER clauses are added to the SPARQL Query and the final query is:
SparqlQUERY = SparqlSELECT + SparqlWHERE + SparqlFILTER

This transformed query is executed by the SQL-to-SPARQL-Translation Algorithm using the SPARQL Query Engine and the retrieved data is returned in relational format seamlessly.

V. IMPLEMENTATION SPECIFICS

The hardware used in the implementation of R2D was a computer running Windows Vista with 2 GB RAM and 2.00 GHz Intel Core2 Duo Processor. The software platforms and tools used include MySQL 5.0 to house the relational equivalent of the given RDF store, Jena 2.5.6 [http://jena.sourceforge.net/index.html] to manipulate the RDF triples, Java 1.5 for development of the algorithms detailed in Section IV, and DataVision v1.2.0 [http://datavision.sourceforge.net/] to visualize/generate reports based on RDF data.

A. Experimental Dataset

Two datasets were used in the experimentation process. The optimized version of the map file generation process was executed against the first dataset which is based on the publications domain described in [6] in order to enable an apples-to-apples performance comparison against the earlier work. The second dataset is a subset of the scenario in Figure 2 and includes the “Employee”, “Department”, and “Project” resources along with the blank nodes for “Name”, “Phone” and “Projects”. The query performance experiments and reporting tool outputs presented here are based on this second dataset.

B. Experimental Results

The relational equivalent of the second dataset was generated using the algorithms detailed in Section IV. The open source visualization tool DataVision, which expects a relational schema as input, was used to view the virtual relational schema generated, query the data using SQL statements, and generate reports off of the data. The times taken by the map file generation process, with and without data sampling, for RDF stores with and without ontological information are illustrated in Figure 4. The process is especially time-intensive for large databases without structural information (which is the case with our experimental data set) but this is only to be expected since RDFMapFileGenerator has to explore every resource to ensure that no property is left unprocessed. The sampling techniques applied improved the performance of the algorithm by a large factor.

Although applying sampling techniques typically result in a reduction in accuracy, we did not encounter this problem in our experiments. The reason is that the RDF data sets we used in our experiments (including the synthetic data set used in this paper, and the LUBM [http://swat.cse.lehigh.edu/projects/lubm/] dataset) did not have too much variance in the predicates of each resource class. For example, the lecturer resource in the LUBM dataset had the same set of predicates irrespective of the number of such resources that existed in the RDF store. Thus, sampling of one Lecturer resource resulted in the same relational entity (and attributes) as the entity generated after the processing of multiple Lecturer resources. Most of the other resources in our datasets also exhibited similar structural properties and hence accuracy continued to remained intact and independent of the sampling techniques as well as the sample sizes used in our experiments.

The rest of the experiments and results presented in this section use the second dataset described earlier. The “Fields” Window in Figure 5 is a screenshot of the relational database schema as seen by DataVision, populated through the JDBC GetDatabaseMetaData Interface which executes the DBSchemaGenerator Algorithm.
As shown, the r2d:SimpleLiteralBlankNode, Employee-Name, is resolved into columns belonging to the Employee table, the r2d:ComplexLiteralBlankNode associated with Employee-Phone is resolved into a 1:N table called Phone, and the r2d:SimpleResourceBlankNode associated with Employee-Projects is resolved into a N:M table called Projects. As stated before, this schema is populated through the GetDatabaseMetaData Interface in the Connection class of the JDBC API within which the two algorithms, RDFMapFileGenerator and DBSchemaGenerator, are triggered.

The “DataVision Report Designer” Window in Figure 5 shows DataVision’s query building process for a sample query involving a GROUP BY clause. At this juncture, the Statement, Prepared Statement, and ResultSet JDBC Interfaces are invoked, which trigger the SQL-to-SPARQL Translation algorithm and return the obtained results to DataVision in the expected tabular format. DataVision, like any other relational reporting/visualization tool, has options to specify aggregation and grouping conditions and functions, the DataVision support group has, for various reasons that are not applicable to our academic test environment, disabled the GROUP BY facility. For the purposes of our research, we have enabled the functionality and the results, appropriately grouped per the desired aggregate function, are as displayed in Figure 5.

In order to compare the performance of queries executed using the virtual relational schema offered by R2D against the query performance achieved through existing RDF visualization tools, a selection of four queries were run against databases of various sizes using R2D and Allegrograph’s Gruff [http://agraph.franz.com/gruff/], a grapher-based triple-store browser, and the results are displayed in Figure 6.

As can be seen, R2D’s performance was far superior to that of Gruff’s. This could be because Gruff persists data on the hard disk in a proprietary manner, requiring additional time/resources for disk I/O, while R2D utilizes Jena’s in-memory store to house the RDF data. The time taken for SQL-to-SPARQL conversion is negligible and nearly constant. Thus, R2D does not add any overheads to the SPARQL query performance and offers an avenue for users to continue to take advantage of readily available visualization tools without data replication or synchronization issues.

VI. CONCLUSION

The stimulus behind the research in this paper is a dearth in the number and variety of data modeling and visualization tools for RDF graph data. The types of RDF Graphs and SQL queries handled and transformed by the current implementation of R2D were expanded from the previous version [6] by including the ability to handle different kinds of blank nodes. Pattern matching and data aggregation functionalities were also added to R2D. With skilled database administrators becoming rarer and more expensive, the importance of applications such as R2D becomes more pronounced as they offer a means to bypass the requirement of databases and their management. Future directions for R2D include support for reification concepts, improving the normalization process for mixed blank nodes, and translation rules for nested/correlated SQL sub-queries.
Figure 6. Response Times for Chosen Queries

REFERENCES


