A Relational Wrapper for RDF Reification

Sunitha Ramanujam[1], Anubha Gupta[1], Latifur Khan[1], Steven Seida[2], Bhavani Thuraisingham[1]

[1] The University of Texas at Dallas, Richardson TX 75080, U.S.A
[2] Raytheon Company, Garland TX 75042, U.S.A
{sxr063200, axg089100, lkhan, bxt043000}@utdallas.edu, steven_b_seida@raytheon.com

Abstract. The importance of provenance information as a means to trust and validate the authenticity of available data cannot be stressed enough in today's web-enabled world. The abundance of data now accessible due to the Internet explosion brings with it the related issue of determining how much of it is trustworthy. Provenance information, such as who is responsible for the data or how the data came to be, assists in the process of verifying the authenticity of the data. Semantic web technologies such as Resource Description Framework (RDF) include the ability to record such provenance information through the process of reification. RDF's popularity has resulted in a demand for modeling and visualization tools. The work presented in this paper, called R2D, attempts to address this demand by innovatively integrating existing, stable technologies such as relational systems with the newer web technologies such as RDF. The work in this paper extends our earlier work by adding support for the RDF concept of reification. Reification enables the association of a level of trust and confidence with RDF triples, thereby enabling the ranking/validation of the authenticity of the triples. Details of the algorithmic enhancements to the various components of R2D that were made to support RDF reification are presented along with performance graphs for queries executed on a database containing crime records data from a police department..

Keywords: Resource Description Framework, Data Provenance, Reification, Data Interoperability.

1 Introduction

The extensive growth of the Internet and associated web technologies has catalyzed research into the notion of a "Semantic Web". This notion is envisioned to augment human reasoning and data management abilities with automated access, extraction, and interpretation of web information. Amongst the many methodologies and standards that are being released periodically as part of the Semantic Web initiative is the Resource Description Framework (RDF) [1], a domain-independent data model that enables interoperability between applications that exchange machine-comprehendible information on the Internet. RDF records information in the form of triples, each consisting of a subject, a predicate, and an object. The predicate is typically a verb and denotes the relationship that exists between the subject and the object. RDF's rapidly increasing popularity as a web content data storage paradigm has necessitated research in the field of visualization tools to inspect and manage data stored using this model. While efforts are ongoing to develop new tools for this purpose, alternate research efforts are underway that focus on integrating benefits and features available in existing methodologies with the advantages offered by the newer web technologies.

R2D, the work presented in this paper, is one such alternative research effort the objective of which is to salvage the time, effort, and resources expended in the development of existing, stable, relational tools by reusing them for RDF data visualization purposes. The advantages of relationalizing RDF stores using applications such as R2D are manifold and include continued leveraging of the knowledge gained by relational database domain experts, reduction of learning curves associated with mastery of new tools, and availability of new technology to resource-constrained small and medium-sized organizations unwilling to invest in expensive tools for fledgling technologies such as RDF [2].

R2D enables the visualization, inspection, and examination of RDF stores using traditional and mature relational tools. The gap between the two paradigms is bridged, through R2D, using a JDBC wrapper that presents, at run-time, a virtual relational version of the RDF store, thereby eliminating the necessity to duplicate and synchronize data. This paper extends the work in [3] by incorporating support for the concept of RDF reification at every stage of R2D's deployment.

Reification is an important RDF concept that provides the ability to make assertions about statements represented by RDF triples. With the increasing number of online resources and sources of information that become available each day, the need to authenticate the available sources becomes essential in order to be able to judge the validity, reliability, and trustworthiness of the information [4]. This authentication is facilitated by augmenting the sources with provenance information, i.e., information describing the origin, derivation, history, custody, or context of a physical or electronic object [5]. RDF Reification, a means of validating a statement/triple based on the trust level of another statement [6], is the solution offered by the WWW consortium for users of RDF stores to record provenance information. Thus, RDF reification is a key component of any application requiring stringent records of the basis/foundation behind every piece of information in the data store. In particular, reification plays a critical role in security-intensive applications where it is imperative to maintain the privacy and ownership of sensitive data. The provenance information captured using reification can be used, in such applications, to monitor and control data access. The contributions of this paper are as follows.

- We propose a mapping scheme for relationalization of RDF Stores. The mapping algorithm extends the algorithm in [3] by including new constructs to handle and process reification information
- Based on the created map file, we propose a transformation process that generates a normalized, domain-specific virtual relational schema corresponding to the RDF store. The transformation algorithm in [3] is extended to include tables and relationships for reification data
- We extend the SQL-to-SPARQL translation algorithm in [3] by including the ability to optionally retrieve reification data, when present, through joins

The organization of the paper is as follows. A brief overview of related research efforts in the relational-to-rdf arena, in either direction, is provided in the following section. R2D mapping preliminaries in terms of the high-level system architecture and mapping constructs are given in section 3 while Section 4 presents detailed descriptions of the various algorithms involved in the mapping process. Section 5 highlights the implementation specifics of the proposed system with sample visualization screenshots and performance graphs for a diverse range of queries on databases of various sizes. Lastly, Section 6 concludes the paper

2 Related Work

With RDF being the current buzzword in the "Semantic Web" community, research efforts are underway in various aspects of RDF such as RDF-ising relational and legacy database systems, transforming traditional SQL queries into RDF query languages such as RDQL and SPARQL, and optimizing performance of queries issued against RDF data sources. However, the overall concept and objectives of R2D are unique since all research efforts attempt to integrate relational database concepts and Semantic Web concepts from a perspective that is opposite to that considered in our work. The only research with objectives very closely aligned with R2D that we have been able to identify till date is RDF2RDB [7] and differences between the two frameworks are tabulated in Table 1.

Table 1: Comparison between RDF2RDB and R2D

RDF2RDB	R2D
Involves data replication resulting in resource wastage and synchronization issues	No data replication/ synchronization issues since relational schema is virtual
Requires presence of ontological information (rdfs:class, rdf:property) for successful mapping	No ontological information required. Mapping discovered through extensive examination of triple patterns
Schema may have unnecessary tables and may not be truly normalized	No unnecessary tables created for to 1:N or N:1 relationships
No details on blank nodes or reification data handling	Meaningful transformations included for blank nodes and reification nodes
No SQL-to-SPARQL transformation	Since relational schema is only virtual, comprehensive

	SQL-to-SPARQL transformation algorithm is included

The D2RQ project [8], an extensively adopted open source project is another significant player in the RDBMS-RDF mapping arena. The goals of D2RQ are the exact reverse of our goals. They attempt to create a mapping from relational databases to RDF Graphs, and transform RDF queries into corresponding SQL queries, thereby making relational data accessible through RDF applications. Our goal, on the other hand, is to enable RDF triples to be accessed through relational applications. RDF123 [9], an open source translation tool, also uses a mapping concept in the spreadsheet domain where the users define mappings between the spreadsheet semantics and RDF graphs for richer translation.

Other efforts in the reverse direction include [10] where Perez and Conrad use relational.OWL to extract the semantics of a relational database and automatically transform them into a machine-readable and understandable RDF/OWL ontology. A few contributions that actually consider the mapping process from the same perspective as our research (i.e., from RDF to relational model) are the ones listed in [11]. However, all models are very generic, involving non-application-specific tables such as resources, literals, statements etc. that would make the determination of the problem domain addressed by the model difficult without examining the actual data. Further, none of the models discuss the concept of RDF reification and the relational transformation of the same. In contrast, R2D details a mapping scheme for representing provenance information in a relational format and enables the users to actually arrive at a complete Entity-Relationship Diagram.

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. Chebotko, et. al. [12] propose an algorithm to translate SPARQL queries with arbitrary complex optional patterns to an equivalent SQL statement. Chen, et. al. [13] discuss a methodology that supports integration of heterogeneous relational databases using the RDF model. An SQL-based RDF Querying Scheme is presented in [14] where the RDF querying capability is made a part of the SQL. The current research efforts presented above indicate that no current solutions address the issue of enabling relational applications to access RDF data without data replication. Hence, to the best of our knowledge, R2D is unprecedented.

3 R2D Architecture and Preliminaries

Figure 1 illustrates the architecture of the proposed system along with the specific R2D modules that are responsible for each function provided by R2D. R2D's primary objective is to present, through a JDBC interface, a relational equivalent of RDF triples stores to visualization tools that are based on a relational model. It also provides an SQL Interface that generates SPARQL versions of SQL queries and passes the same to the SPARQL Query Engine layer for processing and RDF data retrieval.

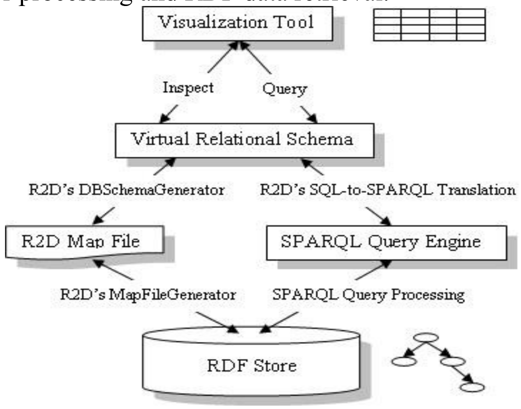


Figure 1: R2D System Architecture and Modules

At the heart of the RDF-to-Relational transformation process is the R2D mapping language – a declarative language that expresses the mappings between RDF Graph constructs and relational database constructs. In order to better understand the constructs comprising the R2D mapping language, let us consider the sample scenario in Figure 2.

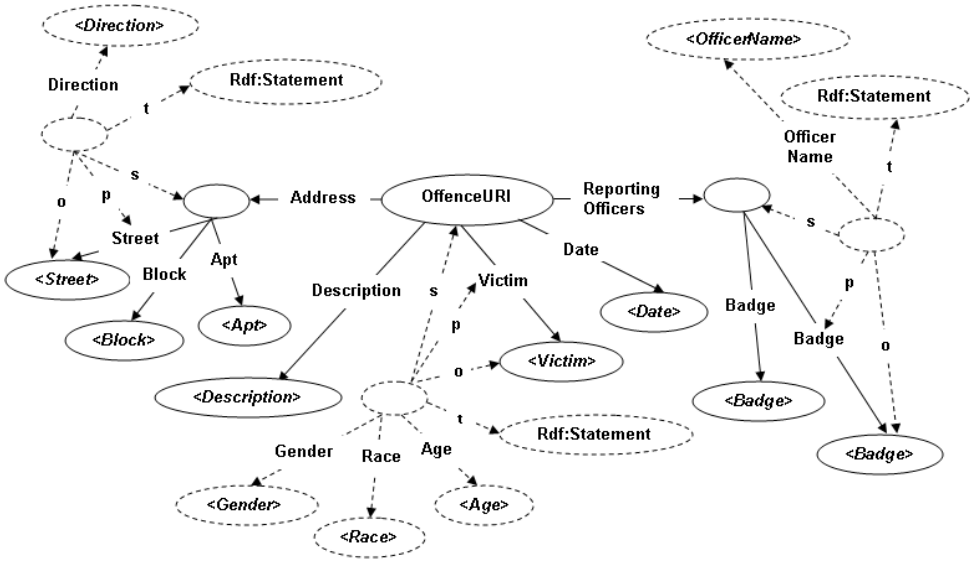


Figure 2: Sample Scenario involving Crime Data

Every solid node with outgoing edges, such as *OffenceURI,* represent a subject/resource. Edges, such as *Address, Description,* and *Victim,* represent predicates and the solid nodes at the end of the edges, such as *<Street>, <Description>, and <Victim>,* represent objects. Empty solid nodes, such as the nodes at which the *Address* and *ReportingOfficer* predicates terminate, represent blank nodes.The nodes in dashed lines represent reified nodes with the "s", "p", "o", and "t" representing 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. 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 Figure 2 are used, wherever applicable, to facilitate better comprehension of the mapping constructs which are discussed in the remainder of the section. Table 2 lists the mappings between some key OWL/RDFS ontology terminologies and RDF concepts to appropriate R2D constructs and their relational equivalents.

Table 2: Notional Mapping between OWL/RDFS concepts, R2D constructs, and Relational concepts

OWL/RDFS/RDF concepts	R2D Mapping Constructs	Relational Equivalent
rdfs:class	r2d:TableMap	Table
rdf:property	r2d:ColumnBridge	Column
rdfs:domain		Table that the rdf:Property is a column of
rdfs:range	r2d:Datatype	Datatype of the column
rdf:type predicate	r2d:KeyField	Values of the primary key column of the table
Blank Node	r2d:SimpleLiteralBlankNode	Columns in parent table
	r2d:ComplexLiteralBlankNode	Columns in a new join table (symbolizing N:M relationship)
	r2d:{Simple/Complex} resourceBlankNode	Depending on cardinality, either columns in the parent table (1:N relationship) or columns in a new join table (N:M relationship)
Reification	r2d:ReificationNode	Columns in either the parent table or in a new join table

The constructs listed in Table 2 above are described in more detail below along with some of the R2D mapping constructs pertaining to regular resources and blank nodes that are essential in order to effortlessly comprehend the work in this paper. A complete list of mapping constructs can be found in [3].

r2d:TableMap: The r2d:TableMap construct refers to a table in a relational database. In most cases, each rdfs:class object will map to a distinct r2d:TableMap, and, in the absence of rdfs:class objects, the r2d:TableMaps are inferred from the instance data in the RDF

Store. Typically, every solid node with multiple predicates in an RDF graph maps into an r2d:TableMap if a similar TableMap does not already exist.

Example: The RDF graph in Figure 2 results in the creation of a TableMap called "*Offence*".

r2d:ColumnBridge: r2d:ColumnBridges relate single-valued RDF Graph predicates to relational database columns. Each rdf:Property object maps to a distinct column attached to the table specified in the rdfs:domain predicate. In the absence of rdf:property/domain information, they are discovered by exploration of the RDF Store data.

Example: The *Description*, *Victim*, and *Date* predicates in Figure 2 become r2d:ColumnBridges belonging to the *Offence* r2d:TableMap.

r2d:SimpleLiteralBlankNode: r2d:SimpleLiteralBlankNodes help relate RDF Graph blank nodes that consist purely of distinct simple literal objects to relational database columns. Predicates off of an r2d:SimpleLiteralBlankNode become columns in the table corresponding to the subject of the blank node.

Example: The object of the *Address* predicate in Figure 2 is an example of an r2d:SimpleLiteralBlankNode which has distinct literal predicates of *Street*, *Block*, and *Apt*, which are, in turn, translated into columns of the same names in the *Offence* r2d:TableMap.

r2d:ComplexLiteralBlankNode: This construct refers to blank nodes in an RDF Graph that have multiple object values for the same subject and predicate concept associated with the blank node. An r2d:ComplexLiteralBlankNode 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 *ReportingOfficers* predicate in Figure 2 is an example of an r2d:ComplexLiteralBlankNode that has multiple object (*Badge*) values for the subject (*OffenceURI*) and predicate (*ReportingOfficers*) concept associated with the blank node. The relational transformation for *ReportingOfficers* involves the generation of an r2d:TableMap of the same name. This *ReportingOfficers* r2d:TableMap includes as columns a *Type* field that holds the values of the predicates off of the CLBN (in our sample scenario, the *Type* field will hold a value of *"Badge"*), and a *Value* field that holds the object values of the predicates off of the CLBM. Additionally, the r2d:TableMap also includes, as foreign key, the *Offence_PK* column which references the primary key of the *Offence* r2d:TableMap.

The concept of reification is supported using many of these previously defined constructs along with a few new constructs that are described below.

r2d:ReificationNode: The r2d:ReificationNode construct is used to map blank nodes associated with "reification quads". Under certain scenarios an r2d:ReificationNode results in the generated of a new "reification" r2d:TableMap. These scenarios are discussed in detail in Section 4.2. The mapping constructs specific to r2d:ReificationNodes are discussed next.

Example: The non-solid nodes corresponding to the *Address-Street* predicate, the *Victim* predicate, and the *ReportingOfficers-Badge* predicate in Figure 2 are examples of r2d:ReificationNodes named *Address_Street_Reif*, *Victim_Reif*, and *ReportingOfficers_Badge_Reif* respectively.

r2d:BelongsToTableMap: This constructs connects an r2d:ReificationNode to the r2d:TableMap corresponding to the resource associated with "rdf:subject" of the r2d:ReificationNode. This information is recorded in the R2D Map File for use during the SQL-to-SPARQL translation.
Example: OffenceURI is the value of the *rdf:subject* predicate of the *Victim_Reif* r2d:ReificationNode. The r2d:TableMap corresponding to *OffenceURI* is *Offence*. Hence, the r2d:BelongsToTableMap construct corresponding to *Victim_Reif* is set to a value of *Offence,* thereby connecting the reification node to a relational table.

r2d:BelongsToBlankNode: This construct connects an r2d:ReificationNode to the r2d: [Simple/Complex][Literal/Resource]BlankNode corresponding to the blank node associated with the "rdf:subject" of the r2d:ReificationNode.
Example: The *rdf:subject* of the *Address_Street_Reif* reification node in Figure 2 consists of a blank node resource called *Address,* which is an r2d:SimpleLiteralBlankNode. Hence, for this reification node the r2d:BelongsToBlankNode construct is used to associate *Address_Street_Reif* to the *Address* blank node.

NOTE: Since the *rdf:subject* of a reification node can either refer to a proper resource or a blank node, the r2d:BelongsToTableMap and r2d:BelongsToBlankNode constructs are mutually exclusive. These are primarily required to enable the generation of appropriate SPARQL WHERE clauses during SQL-to-SPARQL translation.

r2d:ReifiedPredicate: This construct is used to record the predicate corresponding to the *"rdf:predicate"* property of the reification quad mapped by the r2d:ReificationNode construct. This information is, again, required for appropriate SPARQL query generation.
Example: The complete URI of the *Victim* predicate of *OffenceURI* is recorded under the *Victim_Reif* reification node using the r2d:ReifiedPredicate construct.

Predicates of r2d:ReificationNodes are mapped using the r2d:ColumnBridge construct described earlier in this section. Some of the important mapping constructs specific to r2d:ColumnBridges include:

r2d:BelongsToReificationNode: This construct connects an r2d:ColumnBridge to an r2d:ReificationNode entity and is a mandatory component of r2d:ColumnBridges belonging to reification nodes.
Example: The r2d:BelongsToReificationNode associated with the *Victim_Gender* r2d:ColumnBridge is assigned a value of *Victim_Reif*, thereby linking the *Victim_Gender* column with its reification node.

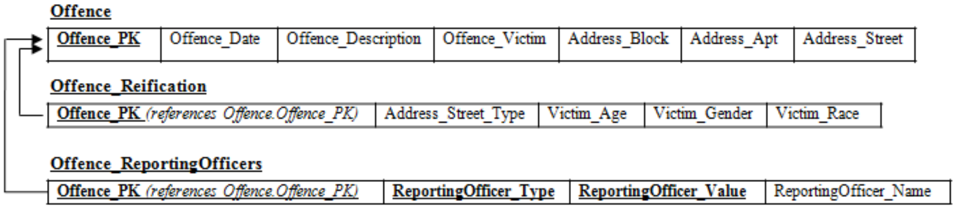
r2d:DataType: This construct specifies the datatype of the r2d:ColumnBridge to which it is associated and comes into play when the structure of the virtual relational database schema objects is examined.

Example: The *Address_Block* column bridge may have an r2d:DataType of *Integer* while the *Victim_Gender* column bridge has an r2d:DataType of *String*.

r2d:Predicate: This construct is used to store the fully qualified property name of the predicate which is associated with the reification r2d:ColumnBridge. This information is used during the SQL-to-SPARQL translation to generate the SPARQL WHERE clauses required to obtain the value of the r2d:ColumnBridge

Example: The complete URI of the *Victim_Gender* predicate of the *Victim_Reif* reification node is recorded using the r2d:Predicate construct.

Figure 3 illustrates the relational schema that is inferred using the above mapping constructs.



Offence

Offence_PK | Offence_Date | Offence_Description | Offence_Victim | Address_Block | Address_Apt | Address_Street

Offence_Reification

Offence_PK *(references Offence.Offence_PK)* | Address_Street_Type | Victim_Age | Victim_Gender | Victim_Race

Offence_ReportingOfficers

Offence_PK *(references Offence.Offence_PK)* | ReportingOfficer_Type | ReportingOfficer_Value | ReportingOfficer_Name

Figure 3: Equivalent Relational Schema for Sample Scenario involving Crime Data

The following sections describe how each of the above mentioned R2D constructs is utilized to transform provenance information available in RDF stores through the reification concept into their relational equivalents.

4 Reification within the R2D Framework

In order to bring to fruition R2D's vision and objectives, various algorithms were designed and developed to implement each component, highlighted in Figure 1, within the R2D framework. The algorithmic details of each R2D module for translation of regular resources and blank nodes are described in depth in [3] and are omitted from this paper due to space constraints. The following sections discuss the algorithmic aspects specifically associated with the presentation of a relational view of RDF **reification** data.

4.1 Mapping Reification Nodes – RDFMapFileGenerator

The RDFMapFileGenerator is the first component in the R2D transformation framework. It is responsible for the generation of a map file containing the correlations between meta-data gleaned from the input RDF store and their relational schema equivalent.

The reification data processing component of the RDFMapFileGenerator is quite straightforward. Every blank node corresponding to a “reification quad” is mapped using the r2d:ReificationNode construct. If the “rdf:subject” property of the “reification quad” mapped by the r2d:Reification construct is a resource, the r2d:BelongsToTableMap construct is used to associate the “reification quad” with the r2d:TableMap corresponding to the resource. If the “rdf:subject” property is a blank node, the r2d:BelongsToBlankNode construct is used to associate the “reification quad” to the r2d:[Simple/Complex][Literal/Resource]BlankNode associated with the “rdf:subject” blank node. Further, if the rdf:object property of the “reification quad” refers to another resource, then r2d:RefersToTableMap construct is used to store this relationship. This information is used in the case of 1:N relationships between two TableMap entities during the SQL-to-SPARQL transformation. Column 1 of Table 3 is the mapping file excerpt for the *Victim_Reif* and the *Address_Street_Reif* reification nodes from Figure 2.

Every non-quad predicate of the reification blank node is mapped using the r2d:ColumnBridge construct and is associated with its reification node using the r2d:BelongsToReificationNode construct. Furthermore, the datatype of the object corresponding to the non-quad predicate is mapped using the r2d:Datatype construct and the URI of the non-quad predicate itself is recorded using the r2d:Predicate construct, for use during the SQL-to-SPARQL transformation. An excerpt from the mapping file that includes information for the *Victim_Gender* and the *Address_Street_Direction* properties of the corresponding reification nodes from Figure 2 is listed in Column 2 of Table 3.

Table 3: Mapping of Reification Nodes and their Predicates in the R2D Map File

Map File Excerpt for Reification Nodes	Map File Excerpt for Predicates of Reification Nodes

map:Victim_Reif a r2d:ReificationNode;
r2d:belongsToTableMap map:Offence;
r2d:datatype xsd:String;
r2d:reifiedPredicate <http://Victim>;
.
map: Address_Street_Reif a
r2d:ReificationNode;
r2d:belongsToBlankNode map: Address;
r2d:datatype xsd:String;
r2d:reifiedPredicate <http://Address/Street>;
.
	map: Victim_Gender a r2d:ColumnBridge;
	r2d:belongsToReificationNode map: Victim_Reif;
	r2d:datatype xsd:String;
	r2d:predicate <http:// Reification/Gender>;
	.
	map: Address_Street_Direction a r2d:ColumnBridge;
	r2d:belongsToReificationNode map:Address_Street_Reif;
	r2d:datatype xsd:String;
	r2d:predicate <http://Reification/StreetDirection>;
	.

Complex reification nodes, such as ones that contain one or more blank node predicates, are processed using the Depth-First-Search tree algorithm (similar to mixed blank nodes processing [3]). Every blank node encountered during DFS is mapped using the r2d:SimpleLiteralBlankNode construct. Every predicate of the blank node is mapped using the r2d:ColumnBridge construct and is linked to it’s parent blank node using the r2d:BelongsToBlankNode construct. Every complex reification node is mapped using the r2d:ComplexReificationNode construct. Blank node objects belonging to an r2d:ComplexReificationNode are connected to the r2d:ComplexReificationNode using the r2d:BelongsToReificationNode construct.

4.2 Relationalizing Reification Data – DBSchemaGenerator

The second stage of the R2D transformation framework, the DBSchemaGenerator, involves the actual virtual, normalized, relational schema generation for the input RDF store based on information in the map file created in stage one. Details of the algorithm pertaining to the relational transformation of reification data are discussed below.

Case (a) For every r2d:TableMap in the virtual relational schema corresponding to an RDF store an additional r2d:TableMap (i.e., a virtual relational table) of type "ReificationTable" is created in the schema if any of the following conditions hold:

a) An r2d:ColumnBridge corresponding to a predicate of a resource that maps to the r2d:TableMap is reified
b) A r2d:MultiValuedColumnBridge (MVCB) that results in the addition of a column to this r2d:TableMap is reified
c) A predicate corresponding to an r2d:SimpleLiteralBlankNode (SLBN) associated with a resource that maps to the r2d:TableMap is reified
d) An r2d:ColumnBridge associated with a predicate of an r2d:SimpleLiteralBlankNode (SLBN) object is reified.

This additional reification table houses the columns corresponding to every single-valued property (other than the 4 properties comprising the quad) of the "reification quads" arising from the 4 conditions described above. In order to better understand the intricacies of the algorithm let us consider the scenario depicted in Figure 2.

The reification of the *Victim* predicate in Figure 2 is an example of condition (a) above while reification of the *Street* predicate of the *Address* SLBN is an example of condition (d). The relational transformation of these reification nodes results in the creation of a new virtual relational table (called *Offence_Reification*) with the following columns (corresponding to the predicates of the reification quads): *Address_Street_Direction*, *Victim_Gender*, *Victim_Race*, and *Victim_Age*.

Case (b) In the case of reification of MultiValuedColumnBridges that result in the creation of a new join table and reification of other kinds of blank nodes other than SLBNs (more details on the various blank node types and their relational representations can be found in [3]), no new reification table is created. Non-quad properties corresponding to such reifications are added as columns to the existing r2d:TableMaps resulting from relationalization of the MVCBs and blank nodes. Reification of the *Badge* predicate of the ComplexLiteralBlankNode (CLBN) *ReportingOfficers* in Figure 2 is one such example where an *OfficerName* column (corresponding to the non-quad predicate of the reification node for *Badge*) is added to the *Offence_ReportingOfficers* TableMap that results from the relational transformation of the *ReportingOfficers* CLBN.

Complex reification nodes are nodes where non-quad predicates include one or more (nested) blank nodes. Due to the numerous types of such mixed combinations that are possible, it would be nearly impossible to arrive at an accurate normalized representation of the same. Hence, r2d:ComplexReificationNodes are processed by flattening their relational equivalents. Depending on whether Case (a) or Case (b) is applicable to the r2d:ComplexReificationNode, either a new or an existing table houses the reification columns. Predicates of literal and resource objects that are at the leaf nodes of the tree rooted at the r2d:ComplexReficationNode are translated into columns in that table.

4.3 Querying Reification Data – SQL-to-SPARQL Translation

The final stage of the R2D transformation framework involves the translation of SQL statements issued against the virtual relational schema generated by stage 2 into equivalent SPARQL queries that are executed against the actual RDF store. This is achieved through the translation algorithm, which also ensures that triples retrieved from the RDF store are returned to the relational visualization tool in the expected tabular format. The translation algorithm presented here extends the earlier version [3] by including the ability to translate queries issued against the virtual tables corresponding to **reification** data.

The SQL-toSPARQL translation process transforms single or multiple table queries with or without multiple where clauses (connected by AND, OR, or NOT operators) and Group By clauses. Due to space constraints, only a high level description of the algorithm is discussed below along with examples to illustrate the translation process.

In order to understand the intricacies of the translation algorithm, let us consider the following SQL query based on the scenario depicted in Figure 2.

```
SELECT address_street, address_street_direction, address_block, victim_gender,
reportingOfficers_badge, reportingOfficers_name FROM Offence, Offence_Reification,
Offence_ReportingOfficers where Offence.Offence_pk = Offence_Reification.Offence_pk AND
Offence.Offence_pk = Offence_ReportingOfficers.Offence_pk WHERE address_block = ‘1100’;
```

The first step in the translation process involves the generation of the SPARQL SELECT clause. For every field in the original SQL SELECT list, a variable is added to the SPARQL SELECT list. The SPARQL SELECT list after fields processing is:

```
SPARQLSelect = SELECT ?address_street, ?address_street_direction, ?address_block, , ?
victim_gender, ?reportingOfficers_badge, ?reportingOfficers_badge_name
```

The processing of regular columns for generation of SPARQL WHERE and FILTER clauses is described in [3]. The resulting SPARQL WHERE clause after processing of regular, non-reification columns as detailed in [3] is as follows:

```
SPARQLWhere = WHERE {
   ?Offence <http://Offence/Address> ?Offence_Address .
   ?Offence_Address <http://Offence/Address/Street> ? address_street .
   ?Offence_Address <http://Offence/Address/Block> ? address_block .
   ?Offence <http://Offence/ReportingOfficers> ?Offence_ReportingOfficers .
   ?Offence_ReportingOfficers http://Offence/ReportingOfficers/Badge ?reportingOfficers_badge
   FILTER (?address_block = ‘1100’ ) }
```

(a) For fields belonging to tables of type “ReificationTable” corresponding to non-complex reification nodes, if the reification quad to which the field belongs reifies a resource (and not a blank node), clauses of the form *[OPTIONAL] { ?reificationQuad <rdf:subject> ?resourceTableMap . ?reificationQuad <rdf:predicate> ?reificationQuad.r2d:ReifiedPredicate . ?reificationQuad <non-quadPredicate> ?reificationColumn . ?reificationQuad <rdf:object> ?reifiedObjectField .}* are added to the SPARQL WHERE clause. The reification quad corresponding to the *victim_gender* column is one such reification. The *OPTIONAL* keyword is optional and is only required for queries involving outer joins. Also, if the field corresponding to the object being reified is not part of the SPARQL WHERE clause, an appropriate selection clause is added to the same. The SPARQL WHERE clauses resulting from the processing of the *victim_gender* column are:

REIFClause1 = ?Offence <http://Offence/Victim> >offence_victim .

*?Victim_Reif <rdf:subject> ?Offence . ?Victim_Reif <rdf:Predicate> <http://Offence/Victim> . ?Victim_Reif <rdf:Object> ?offence_victim . ?Victim_Reif <http://Offence/Victim/*Gender*> ?victim_gender.*

Processing of reification columns belonging to {Literal/Resource}MultiValuedColumnBridge ({L/R}MVCB) tables is similar to the above case with an additional step to identify the parent table from which the {L/R}MVCB table is derived through normalization.

In the case of RMVCB tables where the rdf:object of the reification quad is a resource that maps to another r2d:TableMap (through the r2d:refersToTableMap construct), an additional clause of the form *?subjectResourceTableMap <reificationQuad.r2d:ReifiedPredicate> ?objectResourceTableMap* . is added to the SPARQL WHERE clause.

(b) For fields belonging to tables of type "ReificationTable", if the reification quad to which the field belongs reifies a blank node, clauses of the form given below are added to the SPARQL WHERE clause. Further, if the rdf:object of the reification quad is a resource mapping to another r2d:TableMap then the following additional clause of the form *?BlankNode <reificationQuad.r2d:ReifiedPredicate> ?objectResourceTableMap* . is appended to the SPARQL WHERE Clause.

?ParentTableofBlankNode <BlankNodePredicate> ?BlankNode . [OPTIONAL] {?reificationQuad <rdf:subject> ?BlankNode . ?reificationQuad <rdf:predicate> ?reificationQuad.r2d:ReifiedPredicate . {?reificationQuad <rdf:object> ?reifiedObjectField .?reificationQuad <non-quadPredicate> ?reificationColumn}

The *address_street_direction* reification column belonging to the *"Address"* SLBN in Figure 2 is an example such a reification and the addition to the SPARQL WHERE clause after processing of the same is as given below.

REIFClause2 = ?Address_Street_Reif <rdf:subject> ?Offence_Address . ?Address_Street_Reif <rdf:Predicate> <http://Offence/Address/Street> . ?Offence_Address <rdf:Object> ?address_street . ?Address_Street_Reif <http://Offence/Address/Street/Direction> ?address_street_direction .

Reification columns belonging to CLBNs are processed in a manner very similar to the previous scenario (Scenario (b)). The reification column *ReportingOfficers_Badge_Name* belonging to the *"ReportingOfficers"* CLBN in Figure 2 falls in this category and the SPARQL WHERE clauses for this reification are as follows.

REIFClause3 = ?ReportingOfficers_Reif <rdf:subject> ?Offence_ReportingOfficers . ? ReportingOfficers_Reif <rdf:Predicate> <http://Offence/ReportingOfficers/Badge> . ? ReportingOfficers_Reif <rdf:Object> ?reportingOfficers_badge . ?ReportingOfficers_Reif <http:// Offence/ReportingOfficers/Badge/Name> ?reportingOfficers_badge_name .

Reification columns belonging to r2d:TableMaps corresponding to all other kinds of blank nodes are processed using either scenario (a) or (b) depending on the whether the "rdf:subject" of the reification node is a resource or a blank node.

(c) For fields derived from complex reification nodes, the sequence of predicates leading from the reification node to the (leaf) field are obtained by traversing the tree structure stored during the map file generation process. A WHERE clause is added to the SPARQL WHERE for each of the predicates in sequence.

After the translation procedures described above are applied to the given example SQL statement, the final transformed SPARQL Query is:

SPARQL Statement = SPARQLSelect + SPARQLWhere + REIFClause1 + REIFClause2 + REIFClause3

The transformed SPARQL Query is executed and the retrieved data is returned in relational format seamlessly.

5 Experimental Results

The hardware used for our simulation exercises was a Windows machine with 4GB RAM and 2 GHz Intel Dual Core processor. The software platforms and tools used include Jena 2.5.6 to manipulate the RDF triples data, MySQL 5.0 to house the RDF data in a persistent manner, and DataVision v1.2.0, an open source relational tool, [http://datavision.sourceforge.net/], to visualize, query, and generate reports based on the RDF data. Lastly, BEA Workshop Studio 1.1 Development Environment along with Java 1.5 was used for the development of the algorithms and procedures detailed in Section 4.

5.1 Experimental Datasets

The dataset used in the experiments below is a subset of crime data downloaded from a police department website. The data has triples pertaining to cities and zip codes where crimes were committed, and details of committed crimes as illustrated in Figure 2. While the DataVision screenshots include actual, valid crime data, the voluminous datasets used in the query performance evaluations was artificially generated through a data loading program. However, the structure of the simulated data was kept identical to that of the actual crime dataset and, hence, the results obtained can be directly applied to actual crime data of those volumes. For query performance experiments, Jena's in-memory model was used to load and query the data.

5.2 Simulation Results

The relational equivalent of the crime data was generated using the algorithms detailed in Sections 4.1 and 4.2. The time taken by the map file generation process without any data sampling incorporated for RDF stores of various sizes, with and without reification information, was compared with time taken for the same process when two sampling methods were applied and the results are illustrated in Figure 4. Reified versions of the crime dataset were created by adding reification information to the *Address (Address_Type)* and *Victim (Gender, Race, Age)* objects in Figure 2. This reification information was created for 50% of the offence data in the data stores.

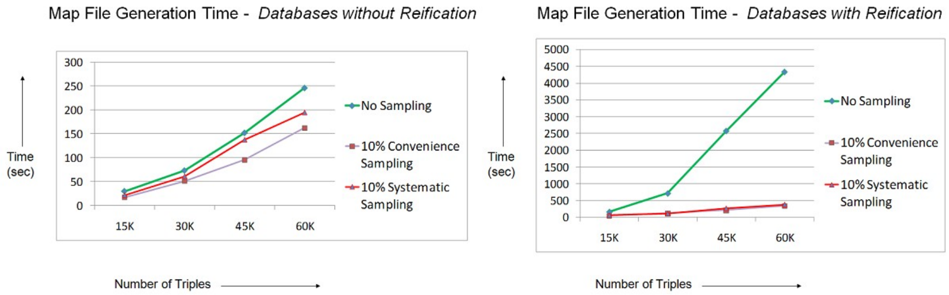
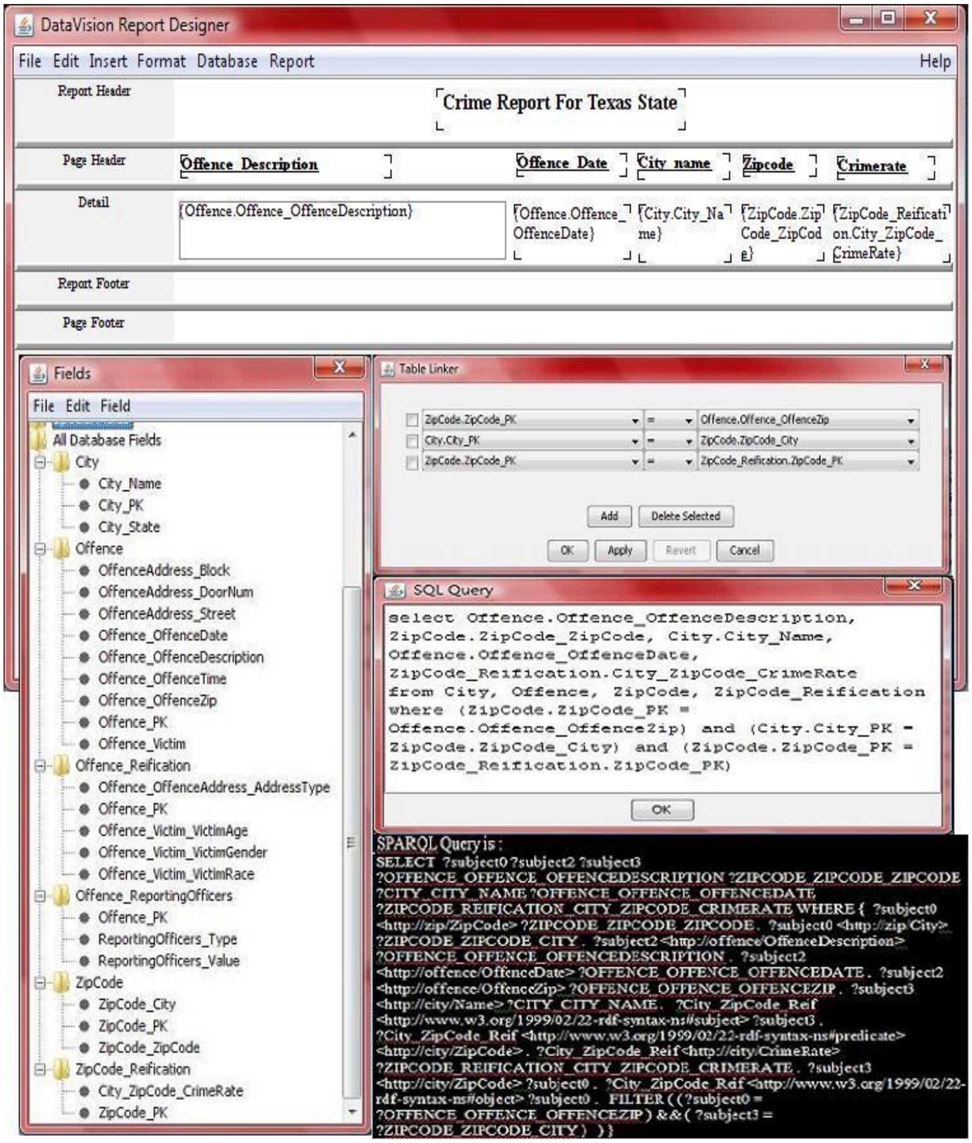


Figure 4: Map File Generation Times with/without Sampling for reified/un-reified data

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 the RDFMapFileGenerator has to explore every resource to ensure that no property is left unprocessed. Furthermore, since even adding reification information for only 50% of the triples in the RDF store resulted in a 25% increase in the size of the data store, the increase in map file generation time for databases with reification information is also predictable. However, the sampling techniques applied improved the performance of the algorithm by a large factor.

Figure 5 is a screenshot of DataVision's Report Designer along with an inset of the database schema as seen by DataVision. The r2d:SimpleLiteralBlankNode associated with *Offence-Address* is resolved into columns belonging to the *Offence* table, and the r2d:ComplexLiteralBlankNode associated with *Offence-ReportingOfficers* is resolved into a 1:N table of the same name. Reification columns are segregated into corresponding reification tables. 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. At this juncture, the Statement, the Prepared Statement, and the ResultSet JDBC Interfaces are invoked, which in turn trigger the SQL-to-SPARQL translation algorithm and return the obtained results to DataVision in the expected tabular format.



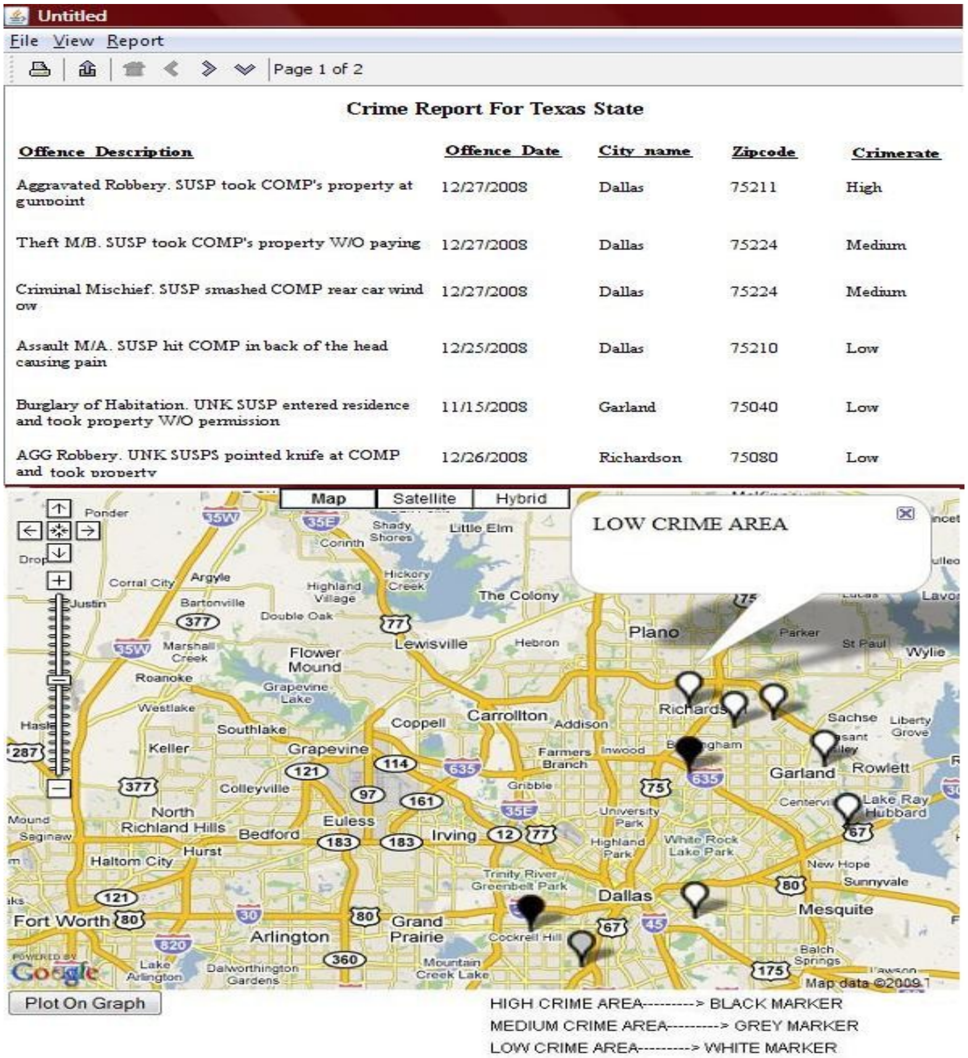
□ ZipCode.ZipCode_PK = Offence.Offence_OffenceZip
□ City.City_PK = ZipCode.ZipCode_City
□ ZipCode.ZipCode_PK = ZipCode_Reification.ZipCode_PK

```
select Offence.Offence_OffenceDescription,
ZipCode.ZipCode_ZipCode, City.City_Name,
Offence.Offence_OffenceDate,
ZipCode_Reification.City_ZipCode_CrimeRate
from City, Offence, ZipCode, ZipCode_Reification
where (ZipCode.ZipCode_PK =
Offence.Offence_OffenceZip) and (City.City_PK =
ZipCode.ZipCode_City) and (ZipCode.ZipCode_PK =
ZipCode_Reification.ZipCode_PK)
```

```
SPARQL Query is :
SELECT ?subject0 ?subject2 ?subject3
?OFFENCE_OFFENCE_OFFENCEDESCRIPTION ?ZIPCODE_ZIPCODE_ZIPCODE
?CITY_CITY_NAME ?OFFENCE_OFFENCE_OFFENCEDATE
?ZIPCODE_REIFICATION_CITY_ZIPCODE_CRIMERATE WHERE { ?subject0
<http://zip/ZipCode> ?ZIPCODE_ZIPCODE_ZIPCODE . ?subject0 <http://zip/City>
?ZIPCODE_ZIPCODE_CITY . ?subject2 <http://offence/OffenceDescription>
?OFFENCE_OFFENCE_OFFENCEDESCRIPTION . ?subject2
<http://offence/OffenceDate> ?OFFENCE_OFFENCE_OFFENCEDATE . ?subject2
<http://offence/OffenceZip> ?OFFENCE_OFFENCE_OFFENCEZIP . ?subject3
<http://city/Name> ?CITY_CITY_NAME . ?City_ZipCode_Reif
<http://www.w3.org/1999/02/22-rdf-syntax-ns#subject> ?subject3 .
?City_ZipCode_Reif <http://www.w3.org/1999/02/22-rdf-syntax-ns#predicate>
<http://city/ZipCode> . ?City_ZipCode_Reif <http://city/CrimeRate>
?ZIPCODE_REIFICATION_CITY_ZIPCODE_CRIMERATE . ?subject3
<http://city/ZipCode> ?subject0 . ?City_ZipCode_Reif <http://www.w3.org/1999/02/22-
rdf-syntax-ns#object> ?subject0 . FILTER ((?subject0 =
?OFFENCE_OFFENCE_OFFENCEZIP ) && ( ?subject3 =
?ZIPCODE_ZIPCODE_CITY ) )}
```

Figure 5: DataVision Report Designer, Relational Schema, and Query Processing

An excerpt from the output returned to DataVision by the SQL-to-SPARQL translation algorithm for the SQL statement in Figure 5 is shown in Figure 6. Selected fields from this output were utilized by another independent application to plot the crime details on Google maps as also illustrated in Figure 6.



Crime Report For Texas State

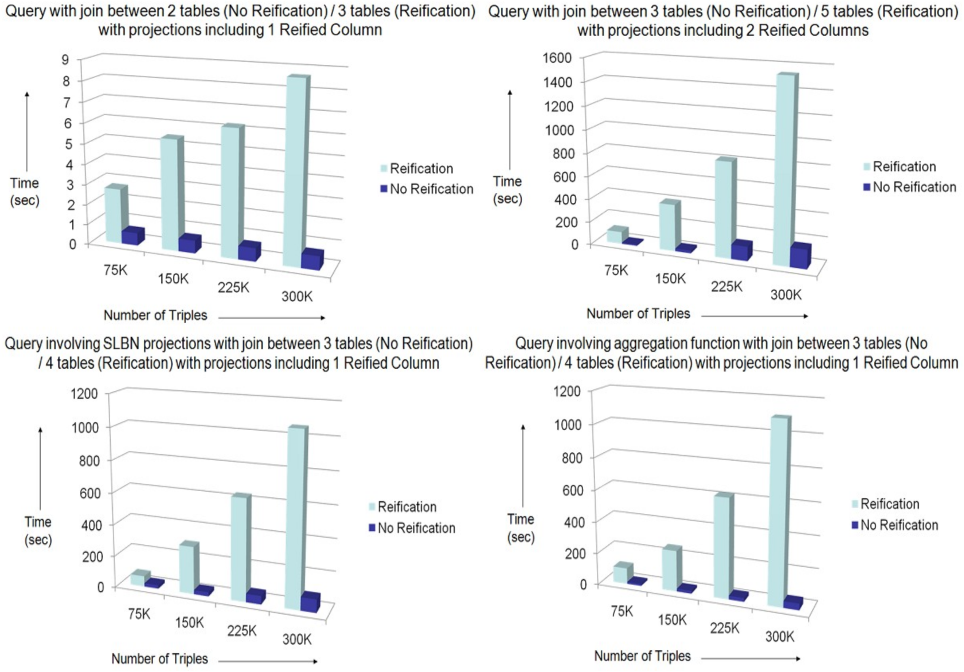
Offence Description	Offence Date	City name	Zipcode	Crimerate
Aggravated Robbery. SUSP took COMP's property at gunpoint	12/27/2008	Dallas	75211	High
Theft M/B. SUSP took COMP's property W/O paying	12/27/2008	Dallas	75224	Medium
Criminal Mischief. SUSP smashed COMP rear car wind ow	12/27/2008	Dallas	75224	Medium
Assault M/A. SUSP hit COMP in back of the head causing pain	12/25/2008	Dallas	75210	Low
Burglary of Habitation. UNK SUSP entered residence and took property W/O permission	11/15/2008	Garland	75040	Low
AGG Robbery. UNK SUSPS pointed knife at COMP and took property	12/26/2008	Richardson	75080	Low

Plot On Graph

HIGH CRIME AREA---------> BLACK MARKER
MEDIUM CRIME AREA---------> GREY MARKER
LOW CRIME AREA---------> WHITE MARKER

Figure 6: Excerpt from Datavision's output in report form and Google Maps plot form

In order to study the performance impact incurred by reification two versions of 4 queries were executed on simulated crime datasets of various sizes. The second version was created by including one or more reification fields to the first version. Figure 7 displays the response times of each of the queries as the sizes of the databases vary. While DataVision has options to specify aggregation and grouping functions, DataVision's support group has, for 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.


Query with join between 2 tables (No Reification) / 3 tables (Reification) with projections including 1 Reified Column

Query with join between 3 tables (No Reification) / 5 tables (Reification) with projections including 2 Reified Columns

Query involving SLBN projections with join between 3 tables (No Reification) / 4 tables (Reification) with projections including 1 Reified Column

Query involving aggregation function with join between 3 tables (No Reification) / 4 tables (Reification) with projections including 1 Reified Column

Figure 7: Response times for the chosen Queries

As was anticipated, reification adds overheads to query processing times as adding a reification quad for a triple results in the addition of a minimum of 4 to 5 extra triples to the data store. However, the time taken for SQL-to-SPARQL conversion is negligible and nearly constant. Thus, R2D does not add overheads to the SPARQL query performance.

SQL queries issued against relational databases created by physically duplicating RDF data may exhibit even better performance since refined performance optimization options have been at the disposal of relational databases for many decades. However, this improved performance comes at the expense of additional disk space due to duplication of data, and additional system resources and human effort required to synchronize the data. On the other hand, for possibly a small price in terms of response time, R2D offers an avenue for users to continue to take advantage of readily available visualization tools without having to “reinvent the wheel”.

6 Conclusion

Provenance Information plays a pivotal role in evaluating quality of data and determining trust in the source of data. This paper extends the R2D framework in [3] by including the ability to represent provenance information available in RDF stores, through the process of reification, in a relational format accessible through traditional relational tools. A JDBC interface aimed at accomplishing this goal through a mapping between RDF reification constructs and their equivalent relational counterparts was presented. The modus operandi of the proposed system was described along with in depth discussion on the algorithms comprising the R2D framework. Graphs highlighting response times for map file generation and query processing obtained using databases of various sizes, both with and without reification data, were also included. Future directions for R2D include providing support for the ability to relate an entity key field to multiple r2d:TableMaps corresponding to resources belonging to different classes, and improving the normalization process for mixed blank nodes and complex reification nodes.

7 References

1. W3C Recommendation (2004) RDF Primer, http://www.w3.org/TR/rdf-primer/
2. Hendler, J.: RDF Due Diligence. http://civicactions.com/blog/rdf_due_diligence (2006)
3. Ramanujam, S., Gupta, A., Khan, L., Seida, S., Thuraisingham, B.: A Framework for the Relational Transformation of RDF Data. UTD Technical Report UTDCS-40-08. http://www.utdallas.edu/~sxr063200/Paper2.pdf (2008)
4. Da Silva Almendra, V., Schwabe, D.: Trust Policies for Semantic Web Repositories. In: Second Semantic Web Policy Workshop, pp 17-31 (2006)
5. Buneman, P., Chapman, A., Cheney, J.: Provenance Management in Curated Databases. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp 539-550 (2006)
6. Powers, S.: Practical RDF. O’Reilly Media (2003)
7. Teswanich, W., Chittayasothorn, S.: A Transformation of RDF Documents and Schemas to Relational Databases. In: IEEE PacificRim Conferences on Communications, Computers, and Signal Processin, pp 38-41 (2007)
8. Bizer, C., Cyganiak, R., Garbers, J., Maresch, O., Becker, C.: The D2RQ Platform. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/

9. Han, L., Finin, T., Parr, C., Sachs, J., and Joshi, A.: RDF123: From Spreadsheets to RDF. In: International Semantic Web Conference, LNCS 5318, pp 451-466 (2008)
10. Perez de Laborda, C., Conrad, S.: Bringing Relational Data into the Semantic Web using SPARQL and Relational OWL. In: 22nd International Conference on Data Engineering Workshops, pp 55 (2006)
11. Melnik, S.: Storing RDF in a Relational Database. http://infolab.stanford.edu/~melnik/rdf/db.html
12. Chebotko, A., Lu, S., Jamil, H. M., and Fotouhi, F.: Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical Report TR-DB-052006-CLJF. Wayne State University (2006)
13. Chen, H., Wu, Z., Wang, H., and Mao, Y.: RDF/RDFS-based Relational Database Integration. In: 22nd International Conference on Data Engineering, pp 94-104, (2006)
14. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL –based RDF Querying Scheme. In: 31st International Conference on Very Large Databases, pp 1216-1227 (2005)