PAPERS ON SEMANTIC WEB RESEARCH AT THE UNIVERSITY OF TEXAS AT DALLAS

Part I: Ontology Alignment and RDF Query Processing

Funded by

Intelligence Advanced Research Projects Activity
and
National Geospatial Intelligence Agency

2007-2009

Point of Contact:

Prof. Latifur Khan
lkhan@utdallas.edu

24 August 2009

TABLE OF CONTENTS

Published Papers

Accepted Papers

7. "R2D: A Bridge between the Semantic Web and Relational Visualization Tools," Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida and Bhavani Thuraisingham

   To appear in *Proc. of Third IEEE International Conference on Semantic Computing,* Berkeley, CA, USA - September 14-16, 2009.
8. "RDFKB: Efficient Support For RDF Inference Queries and Knowledge Management," James Mcglothlin, and Latifur Khan
   To appear in *International Database Engineering & Applications Symposium* (IDEAS) Cetraro (Calabria), Italy, 16-18 September 2009.
9. "R2D: Extracting relational structure from RDF stores"

   Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida, Bhavani M. Thuraisingham

   To appear in *Proc. of ACM/IEEE International Conference on Web Intelligence,* September, 2009, Milan, Italy.
10. "Semantic Schema Matching Without Shared Instances,"
    Jeff Partyka, Neda Alipanah, Latifur Khan, and Bhavani M. Thuraisingham

    To appear in Proc. of *Third IEEE International Conference on Semantic Computing, Berkeley, CA*, USA - September 14-16, 2009.

Submitted Papers

Invited Journal Paper

11. "R2D: A Framework for the Relational Transformation of RDF Data"
    Sunitha Ramanujam, Anubha Gupta, Latifur Khan, Steven Seida, Bhavani Thuraisingham
    Invited to the special issue on *International Journal of Semantic Computing, World Scientific Press.*

Conference Papers

12. "Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce,"
    Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, Bhavani Thuraisingham
    Submitted to *the 1st International Conference on Cloud Computing (CloudCom 2009),* December 1-4, 2009, Beijing, China

13. "Geographically-Typed Semantic Schema Matching,"
    Jeffrey Partyka, Latifur Khan, Bhavani Thuraisingham
    Submitted to *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2009),* Seattle, Washington, USA, November 2009.

14. “Smarter Searches using Location Driven Knowledge Discovery and Mining,”
Satyen Abrol, Tahseen Al-khateeb, Latifur Khan
Submitted to *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2009),* Seattle, Washington, USA, November 2009.

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.

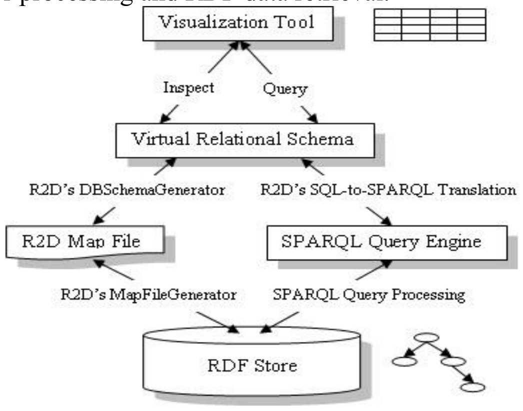


Figure 1: R2D System Architecture and Modules

At the heart of the RDF-to-Relational transformation process is the R2D mapping language – a declarative language that expresses the mappings between the 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 illustrated in Figure 2.

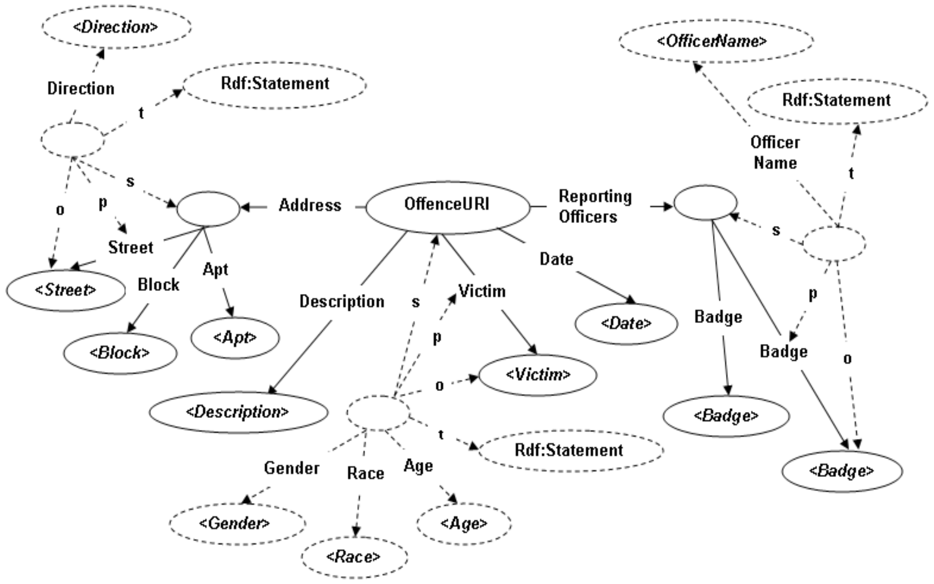


Figure 2: Sample Scenario involving Crime Data

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.

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 are briefly described below. 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 the *Badge* r2d:ColumnBridge and 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 and the details of the same listed in Table 3.

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.

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 2 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 2.

Table 2: 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.

```
01 Algorithm DBSchemaGenerator (for Reification)
02 Input: RDF-to-Relational Schema Mapping File
03 Output: A Normalized Relational Schema
04 Begin
05  For every entry of type r2d:ReificationNode or r2d:ComplexReificationNode
06   ParentTable = ReificationNode.BelongsToTableMap OR ReificationNode.BelongsToBlankNode
07   If ParentTable.Type = "Table"  OR ParentTable.Type = "SimpleLiteralBlankNode" then
08     If ReificationNode.ReifiedPredicate refers to MultiValuedColumnBridge then
09       If MVCB represents N:M relationship then
10         ReificationTable = Table corresponding to MVCB
11       Else              /* Line 09 if */
12         ParentTable = Table on N-side of the relationship
13         ReificationTable = ParentTable_Reification
14       End if            /* Line 09 if */
15     Else                /* Line 08 if */
16       ReificationTable = ParentTable_Reification
17       If reification table for ParentTable does not exist then
18         Tables += ReificationTable
19       End if
20     End if              /* Line 08 if */
21   Else                  /* Line 07 if */
22     ReificationTable = Table Corresponding to Blank Node
23   End if                /* Line 07 if */
24   For every entry of type r2d:ColumnBridge with r2d:BelongsToReificationNode
25     If column does not exist in ReificationTable then
26       ReificationTable.columns += column
27     End if
28   End For               /* Line 24 For  */
29   For every entry of type r2d:SimpleLiteralBlankNode(SLBN) with r2d:BelongsTo(Complex)ReificationNode
30     For every r2d:ColumnBridge with r2d:BelongsToBlankNode = above SLBN
31       Repeat Steps 24-28
32     End For             /* Line 30 For */
```

```
33    For every entry of type r2d:SimpleLiteralBlankNode that belongs to Line 29's SLBN
34      Repeat Steps 29-36
35    End For          /* Line 33 For */
36   End For           /* Line 29 For */
37  End For            /* Line 05 For  */
38 End Algorithm
```

Figure 3: DBSchemaGenerator Algorithm

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 the 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 3.

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 *“Name”* SLBN in Figure 3 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 3 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 3. 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 RDFMapFileGenerator and DBSchemaGenerator Algorithms detailed in Section 4. 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.
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.

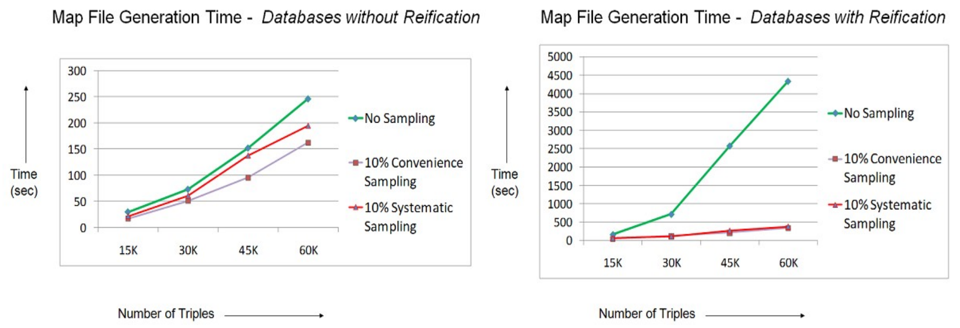
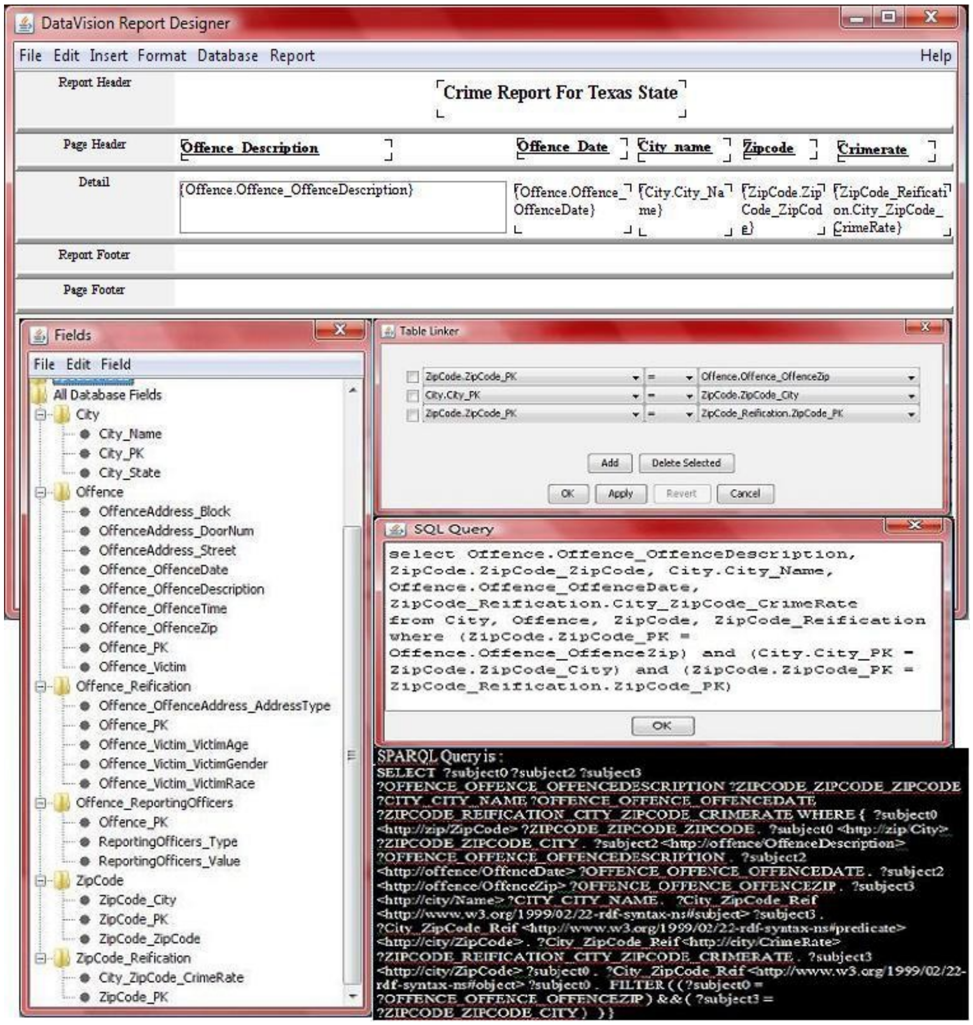


Figure 4: Map File Generation Times with/without Sampling for reified/un-reified data

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

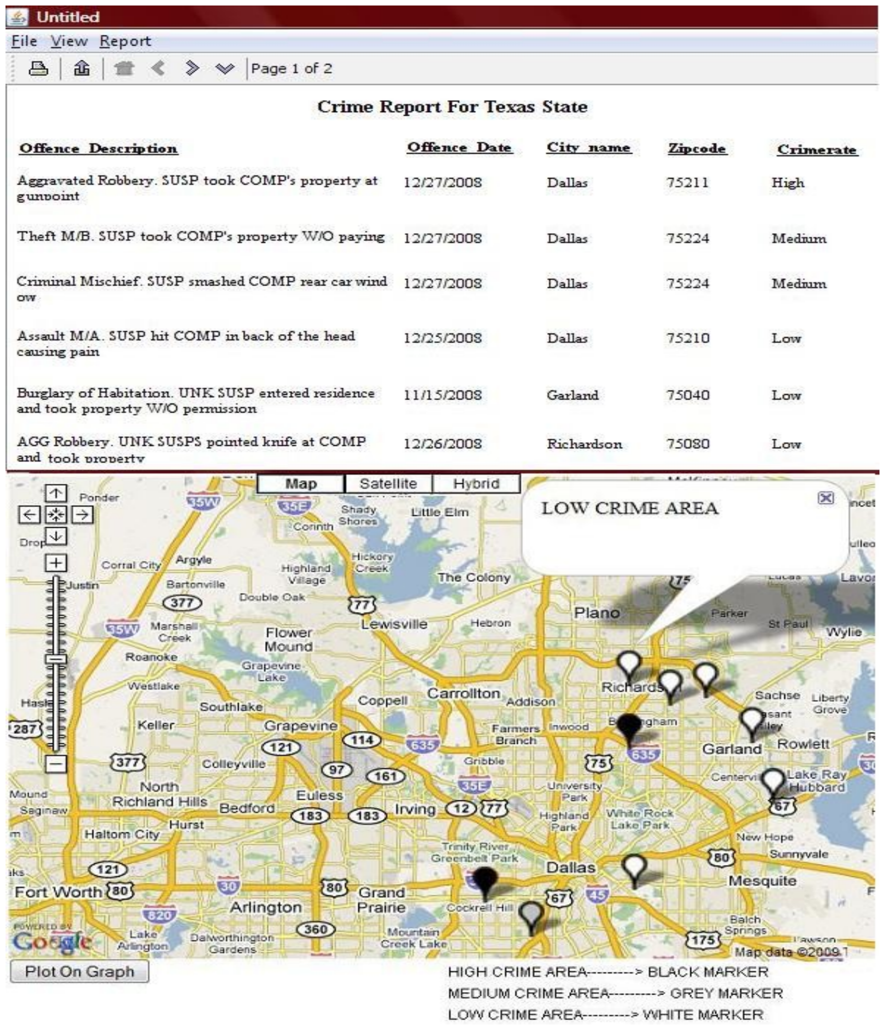
```
select Offence.Offence_OffenceDescription,
ZipCode.ZipCode_ZipCode, City.City_Name,
Offence.Offence_OffenceDate,
ZipCode_Reification.City_ZipCode_CrimeRate
from City, Offence, ZipCode, ZipCode_Reification
where (ZipCode.ZipCode_PK =
Offence.Offence_OffenceZip) and (City.City_PK =
ZipCode.ZipCode_City)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

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.

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

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.

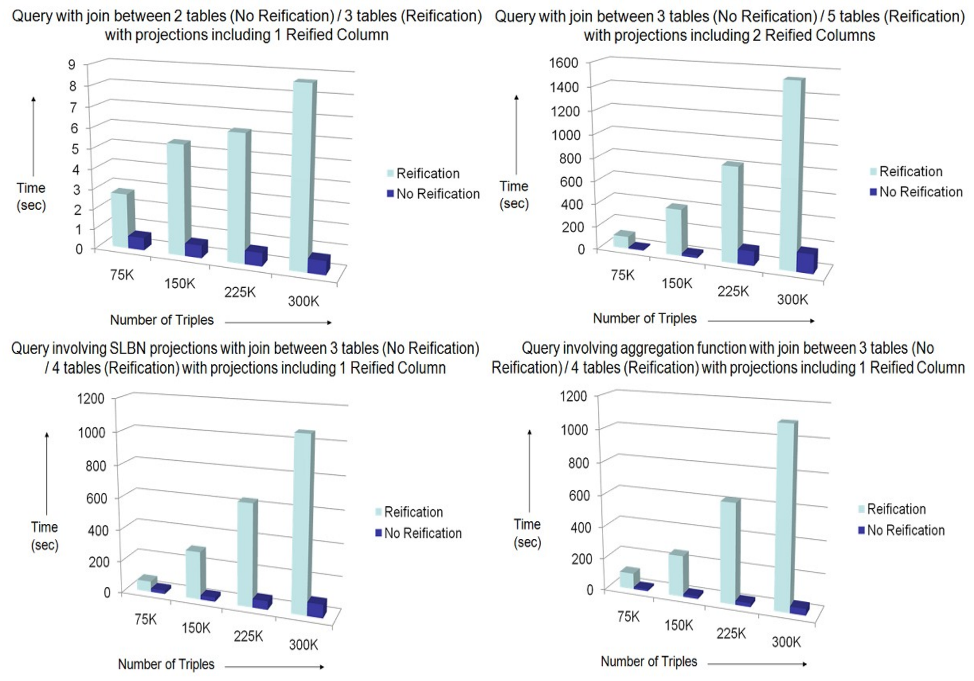


Figure 7: Response times for the chosen Queries

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/. Accessed 28 January 2009
2. Hendler J (2006) RDF Due Diligence. http://civicactions.com/blog/rdf_due_diligence. Accessed 15 January 2009
3. Ramanujam S, Gupta A, Khan L et al (2008) A Framework for the Relational Transformation of RDF Data. UTD Technical Report UTDCS-40-08. http://www.utdallas.edu/~sxr063200/Paper2.pdf. Accessed 29 January 2009
4. Da Silva Almendra V, Schwabe D (2006) Trust Policies for Semantic Web Repositories. In Second Semantic Web Policy Workshop: 17-31
5. Buneman P, Chapman A, Cheney J (2006) Provenance Management in Curated Databases. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data: 539-550
6. Powers S (2003) Practical RDF. O'Reilly Media.
7. Teswanich W, Chittayasothorn S (2007) A Transformation of RDF Documents and Schemas to Relational Databases. In IEEE PacificRim Conferences on Communications, Computers, and Signal Processing: 38-41
8. Bizer C, Cyganiak R, Garbers J et al (2007) The D2RQ Platform. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/. Accessed 29 January 2009
9. Han L, Finin T, Parr C et al (2008) RDF123: From Spreadsheets to RDF. International Semantic Web Conference, LNCS 5318: 451-466

10. Perez de Laborda C, Conrad S (2006) Bringing Relational Data into the Semantic Web using SPARQL and Relational OWL. In: 22nd International Conference on Data Engineering Workshops:55
11. Melnik, S (2001) Storing RDF in a Relational Database. http://infolab.stanford.edu/~melnik/rdf/db.html. Accessed 29 January 2009
12. Chebotko A, Lu S, Jamil HM et al (2006) Semantics Preserving SPARQL-to-SQL Query Translation for Optional Graph Patterns. Technical Report TR-DB-052006-CLJF. Wayne State University. Accessed 15 January 2009
13. Chen H, Wu Z, Wang H et al (2006) RDF/RDFS-based Relational Database Integration. 22nd International Conference on Data Engineering: 94-104
14. Chong EI, Das S, Eadon G et al (2005) An Efficient SQL –based RDF Querying Scheme. 31st International Conference on Very Large Databases: 1216-1227.

Relationalizing RDF Stores for Tools Reusability

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

[1] The University of Texas at Dallas
800, West Campbell Road
Richardson, TX 75080-3021
{sxr063200, axg089100, lkhan, bxt043000}@utdallas.edu

[2] Raytheon Corporation
1200 South Jupiter Road
Garland, TX 75042
steven_b_seida@raytheon.com

ABSTRACT
The emergence of Semantic Web technologies and standards such as Resource Description Framework (RDF) has introduced novel data storage models such as the RDF Graph Model. In this paper, we present a research effort called R2D, which attempts to bridge the gap between RDF and RDBMS concepts by presenting a relational view of RDF data stores. Thus, R2D is essentially a relational wrapper around RDF stores that aims to make the variety of stable relational tools that are currently in the market available to RDF stores without data duplication and synchronization issues.

Categories and Subject Descriptors
D.2.12 [Interoperability]: Data Mapping and Interoperability between applications

General Terms
Algorithms, Design, Management.

1. INTRODUCTION
Every new data storage paradigm comes with its own demands for data modeling and visualization tools to simplify data management. In order to salvage the time, effort, and resources exhausted in the development of such tools for existing, mature technologies such as relational database management systems, it would be prudent to focus on research efforts that attempt to recycle these tools for new data models such as the RDF Graph Model. R2D is one such effort that attempts to eliminate the learning curves associated with mastering new tools and to leverage the advantages offered by the relational tools while continuing to reap the benefits provided by the newer web technologies and standards such as RDF.

R2D, which could be considered as the complement of D2RQ [1] since it works in the reverse direction, uses a declarative mapping scheme for the translation of RDF Graph structures to equivalent relational schema constructs. Functionalities provided by R2D are:

- Ability to infer the entities comprising the RDF store, their attributes, and the relationships that exist between them.
- Ability to generate a meaningful, normalized, domain-specific relational schema corresponding to the RDF store.
- Ability to appropriately transform blank nodes, RDF containers, and RDF collections to relational entities/attributes.
- Ability to access information in a non-RDBMS data store using relational data visualization tools
- Ability to query a non-RDBMS data store using conventional SQL statements

R2D is implemented as a JDBC wrapper around RDF stores and the system architecture is illustrated in Figure 1.

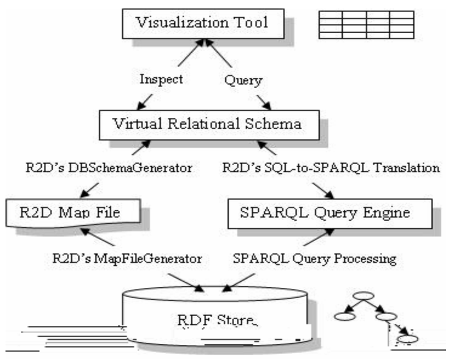


Figure 1. R2D System Architecture

At the heart of the transformation of RDF Graphs to virtual relational database schemas is the R2D mapping language and details of the same are presented below.

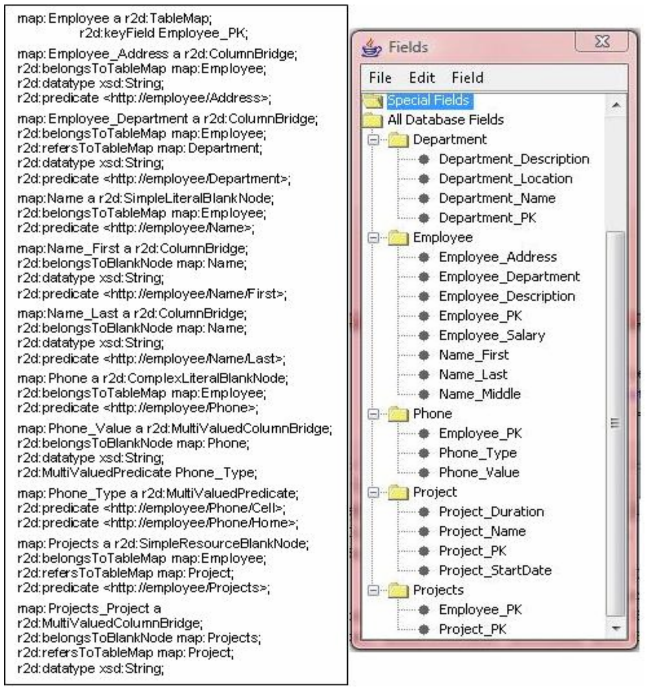
2. R2D Mapping Constructs
The chief construct of the R2D mapping language is the TableMap, which refers to a table in a relational database. Each rdfs:class object in the RDF store maps to a distinct r2d:TableMap, and, in the absence of rdfs:class objects, the r2d:TableMaps are inferred from the instance data in the RDF Store. Each TableMap entity has a set of columns which correspond to the predicates associated with the resource mapped by the TableMap. Simple predicates are mapped using the r2d:ColumnBridge construct while multi-valued predicates are mapped using the r2d:MultiValuedColumnBridge construct. Foreign key relationships are handled using the r2d:refersToTableMap construct. R2D also supports a variety of blank node scenarios such as single or multi-valued literal blank nodes, single or multi-valued resource blank nodes, and mixed (literal/resource) blank nodes using a variety of constructs that are described in [2]. RDF features such as RDF Containers and RDF Collections are also supported using the above-mentioned blank node mapping constructs. Lastly, R2D provides the r2d:MultiValuedPredicate construct to handle RDF triples that essentially map to multi-valued attributes in the relational domain. The "MultiValued" constructs are vital to ensure the generation of a normalized relational schema corresponding to the RDF store.

3. RDF Modules
There are three modules comprising the R2D framework. A) The first module is the RDFMapFileGenerator module which takes an RDF triples database as input and produces a mapping file as output. The map file generator includes detailed specifications to handle a variety of RDF blank nodes, containers, and collections. This module can be bypassed in the presence of a domain expert who can provide the mapping file manually. B) The second module is the DBSchemaGenerator, which parses the map file generated by the

first module and, for the RDF store, presents a list of relational tables, columns, and the relationships between them. C) The last R2D module is the SQL-to-SPARQL transformation process that transparently converts any SQL statement issued against the virtual relational schema generated by DBSchemaGenerator into its SPARQL equivalent, and transforms the results obtained from the SPARQL Query Engine into a relational/tabular format. This module includes the ability to translate SQL pattern matching and aggregation functionality into appropriate SPARQL clauses where applicable.

4. SAMPLE SCENARIO

The example RDF database stored using Jena 2.5.6 and considered for experimentation and elucidation purposes is one which has information pertaining to employees, departments, and projects. Employees have literal properties such as Name and Address. They also have blank nodes that contain phone numbers and projects information for the employee, and a resource predicate that associates a department with each employee. The map file excerpt corresponding to the Employee entity is listed in Figure 2 along with the relational schema corresponding to the RDF store, as seen through the relational visualization tool, DataVision [3]. A more detailed description of the various R2D Mapping constructs and schema generation processes can be found in [2].

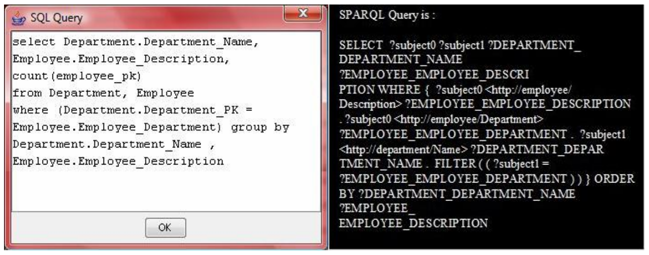


```
map:Employee a r2d:TableMap;
        r2d:keyField Employee_PK;
map:Employee_Address a r2d:ColumnBridge;
r2d:belongsToTableMap map:Employee;
r2d:datatype xsd:String;
r2d:predicate <http://employee/Address>;
map:Employee_Department a r2d:ColumnBridge;
r2d:belongsToTableMap map:Employee;
r2d:refersToTableMap map:Department;
r2d:datatype xsd:String;
r2d:predicate <http://employee/Department>;
map:Name a r2d:SimpleLiteralBlankNode;
r2d:belongsToTableMap map:Employee;
r2d:predicate <http://employee/Name>;
map:Name_First a r2d:ColumnBridge;
r2d:belongsToBlankNode map:Name;
r2d:datatype xsd:String;
r2d:predicate <http://employee/Name/First>;
map:Name_Last a r2d:ColumnBridge;
r2d:belongsToBlankNode map:Name;
r2d:datatype xsd:String;
r2d:predicate <http://employee/Name/Last>;
map:Phone a r2d:ComplexLiteralBlankNode;
r2d:belongsToTableMap map:Employee;
r2d:predicate <http://employee/Phone>;
map:Phone_Value a r2d:MultiValuedColumnBridge;
r2d:belongsToBlankNode map:Phone;
r2d:datatype xsd:String;
r2d:MultiValuedPredicate Phone_Type;
map:Phone_Type a r2d:MultiValuedPredicate;
r2d:predicate <http://employee/Phone/Cell>;
r2d:predicate <http://employee/Phone/Home>;
map:Projects a r2d:SimpleResourceBlankNode;
r2d:belongsToTableMap map:Employee;
r2d:refersToTableMap map:Project;
r2d:predicate <http://employee/Projects>;
map:Projects_Project a
r2d:MultiValuedColumnBridge;
r2d:belongsToBlankNode map:Projects;
r2d:refersToTableMap map:Project;
r2d:datatype xsd:String;
```

```
Fields
File  Edit  Field
Special Fields
All Database Fields
  Department
    Department_Description
    Department_Location
    Department_Name
    Department_PK
  Employee
    Employee_Address
    Employee_Department
    Employee_Description
    Employee_PK
    Employee_Salary
    Name_First
    Name_Last
    Name_Middle
  Phone
    Employee_PK
    Phone_Type
    Phone_Value
  Project
    Project_Duration
    Project_Name
    Project_PK
    Project_StartDate
  Projects
    Employee_PK
    Project_PK
```

Figure 2: Map File Excerpt for "Employee" and Relational Schema for RDF Store

The mapping information from the map file is used by the SQL-to-SPARQL translator to convert any SQL statements issued against the virtual relational schema into its SPARQL equivalent. One such query issued through DataVision and its converted SPARQL equivalent is illustrated in Figure 3.



```
SQL Query
select Department.Department_Name,
Employee.Employee_Description,
count(employee_pk)
from Department, Employee
where (Department.Department_PK =
Employee.Employee_Department) group by
Department.Department_Name ,
Employee.Employee_Description
OK
```

```
SPARQL Query is :

SELECT  ?subject0 ?subject1 ?DEPARTMENT_
DEPARTMENT_NAME
?EMPLOYEE_EMPLOYEE_DESCRI
PTION WHERE {  ?subject0 <http://employee/
Description> ?EMPLOYEE_EMPLOYEE_DESCRIPTION
. ?subject0 <http://employee/Department>
?EMPLOYEE_EMPLOYEE_DEPARTMENT .  ?subject1
<http://department/Name> ?DEPARTMENT_DEPAR
TMENT_NAME .  FILTER ( ( ?subject1 =
?EMPLOYEE_EMPLOYEE_DEPARTMENT ) ) } ORDER
BY ?DEPARTMENT_DEPARTMENT_NAME
?EMPLOYEE_
EMPLOYEE_DESCRIPTION
```

Figure 3: SQL-to-SPARQL Transformation

5. PERFORMANCE RESULTS

The performance of data retrieval using R2D was compared with that obtained using an RDF visualization tool called GRUFF [4] using a variety of queries of varying complexity. R2D queries were fired against Jena's in-memory data store. Table 1 lists the results obtained for an RDF triples database size of 0.5M.

Table 1: Query Performance Results

QUERY	GRUFF	R2D
3 Projections, 1 Where clause for LIKE, 2-table join	315secs	8secs
4 Projections involving properties and SimpleLiteralBlankNodes, 3 Where clauses involving LIKE and equality operators connected using conjunction and disjunction, 2-table join	315secs	6secs
3 Projections involving properties and SimpleResourceBlankNodes, 3-table join	600secs	6secs
4 Projections from both types of blank nodes above, aggregation function using Group By, 3-table join	550secs	8secs

The performance results highlight the fact that R2D's performance is far superior to that of the direct RDF visualization tool. Further, the time taken by the SQL-to-SPARQL translation process is negligible and, hence, R2D does not add any overheads to the SPARQL Query processing performance.

6. CONCLUSION

In today's highly web-enabled world, R2D offers users the ability to reuse existing knowledge and resources by enabling the integration of traditional mature and stable relational tools with the newer semantic web technologies such as RDF stores. The competitive query performance results obtained through R2D make it a viable contender in the RDF data visualization and management arena. Future work includes support for reification.

7. REFERENCES

[1] Bizer, C., and Cyganiak, R. D2R – Publishing Relational Databases on the Semantic Web. 5th International Semantic Web Conference, 2006

[2] Ramanujam, S., Gupta, A., Khan, L., Seida, S., Thuraisingham, B. A Framework for the Relational Transformation of RDF Data. UTD Technical Report UTDSC-40-08. http://www.utdallas.edu/~sxr063200/Paper2.pdf, 2008.

[3] DataVision. The Open Source Report Writer. http://datavision.sourceforge.net/

[4] GRUFF. A Grapher-Based Triple-Store Browser for Allegrograph. http://agraph.franz.com/gruff/

Content-based Ontology Matching for GIS Datasets

Jeffrey Partyka[1], Neda Alipanah[1], Latifur Khan[1], Bhavani Thuraisingham[1], Shashi Shekhar[2]

Department of Computer Science

University of Texas at Dallas[1]

University of Minnesota[2]

{jlp072000, na061000, lkhan, Bhavani.thuraisingham}@utdallas.edu[1]

shekhar@cs.umn.edu[2]

ABSTRACT

The alignment of separate ontologies by matching related concepts continues to attract great attention within the database and artificial intelligence communities, especially since semantic heterogeneity across data sources remains a widespread and relevant problem. In particular, the Geographic Information System (GIS) domain presents unique forms of semantic heterogeneity that require a variety of matching approaches.

Our approach considers content-based techniques for aligning GIS ontologies. We examine the associated instance data of the compared concepts and apply a content-matching strategy to measure similarity based on value types based on N-grams present in the data. We focus special attention on a method applying the concepts of mutual information and N-grams by developing 2 separate variations and testing them over GIS dataset including multi-jurisdictions. In order to align concepts, first we find the appropriate columns. For this, we will exploit mutual information between two columns based on the type distribution of their content. Intuitively, if two columns are semantically same, type distribution should be very similar. We justify the conceptual validity of our ontology alignment technique with a series of experimental results that demonstrate the efficacy and utility of our algorithms on a wide-variety of authentic GIS data.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *semantic networks, representations (procedural and rule-based)*

General Terms

Algorithms, Measurement, Design, Reliability, Experimentation, Human Factors

Keywords

Ontology, Ontology Alignment, Schema Matching, Geographic Information Systems, Dataset

1. INTRODUCTION

Ontology alignment is the most recent form of the information integration problem. The most popular definition of an ontology is that of a "formal, explicit specification of a shared conceptualization" proposed by Gruber[1]. In practice, ontologies for a given domain consist of a series of classes (or concepts) along with their properties, restrictions and instances, many of which are related by various types of relationships. The alignment of ontologies, therefore, entails deriving correspondences between concepts and their associated properties and instances.

Ontology matching continues to attract extensive interest, particularly with regards to the domain of GIS. Related work includes [2], which formally describes the various ways in which semantic heterogeneity may be encountered during the ontology alignment process in the GIS domain. Sunna and Cruz [3] describe matching ontologies using structural properties such as sibling similarity and descendant's similarity. Using these ideas, they introduce an ontology alignment tool for use in the GIS domain called AgreementMaker [4].

In developing a strategy for aligning GIS ontologies, we consider a novel approach based on the information theoretic concept of mutual information that utilizes content-matching techniques Specifically, we identify type distributions over distinct N-grams among the columns within the instance data of compared concepts and use these to obtain a similarity value (from now on the words column and attribute are used interchangeably). An *N-gram* is simply a substring of length N consisting of contiguous characters. In particular, using distinct types we strive to capture patterns from both the raw text of GIS datasets and encoded versions of this text which substitute the individual letters for their character types (i.e., letters are replaced by an 'a', numbers are replaced by a 'n', etc.)

A number of schema matching publications describing methods tailored more to the database community influenced our work. Dai, Koudas et al. [5] discussed content-based schema matching based on distributions of N-grams among compared columns. Despite the influence of this publication, some crucial differences exist between their approach and the methods explained here. First, their approach used data sources containing raw text from any given domain, whereas our methods specifically targeted the GIS domain. Second, their approach is designed for the area of schema matching, while our methods are made for the area of ontology matching, which means that in our work, additional complexities needed to be considered, such as concept matching over names as well as content. Third, they defined statistical types only over distributions of N-grams and used these to determine column similarity. In addition to this idea, our approach considers a number of variations. One of these treats N-grams themselves as distinct types extracted from the tuple values of compared columns. Also, these two approaches are applied over regular text and over encoded text, which allowed us to observe our algorithm's performance over vastly different kinds of data.

The rest of this paper is organized as follows. In section 2, we discuss the problem to be solved and our proposed solution for content similarity at both the conceptual and attribute levels. Next, in section 3, we present a series of experiments and their associated results.

2. PROBLEM STATEMENT AND PROPOSAL

2.1 Problem Statement

Given 2 data sources, S_1 and S_2, each of which is represented by ontologies O_1 and O_2, the goal is to find similar concepts between O_1 and O_2 by examining their names and their respective instances. Let us assume that O_1 and O_2 are derived from the GIS domain. Figures 1 and 2 display O_1 and O_2, the ontologies to be aligned. Also displayed for each ontology are their constituent concepts and two sample identifying attributes for each concept. Both ontologies are derived from the Roads and Ferries package of the Geographic Data Files (GDF) data model and the Ontology for Traffic Networks.

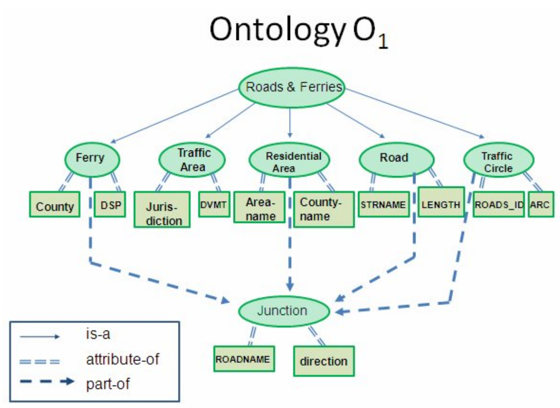


Figure 1. Concepts and attributes of ontology O_1

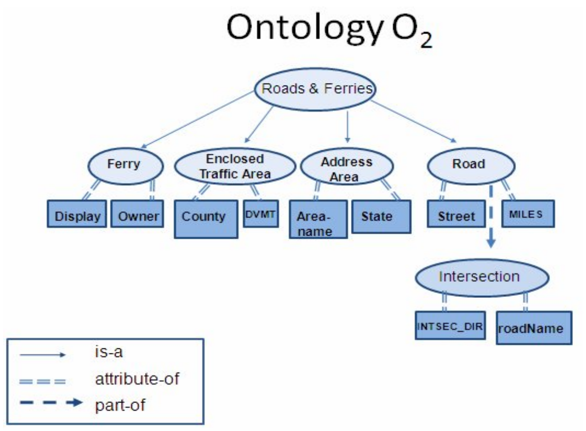


Figure 2. Concepts and attributes of ontology O_2.

With this in mind, an effective ontology alignment procedure would be expected to match up concepts which are semantically equivalent. In this case, O_1 and O_2 both feature Road and Ferry concepts, so a strong similarity value between each one would be expected. Furthermore, a close semantic equivalence would also seem to exist between the Residential Area concept of O_1 and the Address Area concept of O_2, and Traffic Area of O_1 and Enclosed Traffic Area of O_2. There may also be a fairly strong semantic similarity between Junction of O_1 and Intersection of O_2. Our goal is to determine this semantic similarity given instances for concepts.

2.2 Content Similarity

In order to do ontology alignment, we need to determine the similarity between concepts C_1 and C_2, which come from two different ontologies O_1 and O_2, respectively. For this, first we need to find similarity between the attributes of C_1 and C_2. Recall that each concept may have a set of attributes. For attribute similarity, without loss of generality, first we focus on 1:1 matching and later will apply our algorithm to 1:M matching. We will pick an attribute *a* from C_1 and compare it with all attributes in C_2 based on the EBD derived from attribute similarity. Attribute *a* will be assigned with the attribute in concept C_2 which gives the largest *Entropy based distribution (EBD)* (see Section 2.2.2).

2.2.1 Measuring type similarity

Content matching between two concepts involves measuring the similarity between the instance values for a pair of attributes. This is accomplished by extracting instance values from the compared attributes, subsequently extracting a characteristic set of N-grams from these instances, and finally comparing the respective N-grams for each attribute. During all of our experiments involving N-grams in this paper, the value of N was set equal to 2.

We experiment with a number of varying approaches using 2-grams that ultimately determines the instance similarity between the compared attributes. In our first approach, called *DNF (distinct N-gram features)*, we extract distinct N-gram features from the instances themselves and consider each unique 2-gram extracted as a value type. The similarity between the attributes is measured by determining the disparity between the 2-grams extracted and between the frequency of 2-grams they have in common. An alternative approach to the aforementioned method of content similarity via 2-gram feature extraction, called TPF (tuple features) is to collect all 2-grams and their corresponding frequencies for each tuple value within one of the compared attributes and use this information to construct a 2-gram set. In this case, the set of 2-grams itself would be considered a value type, rather than any of the individual 2-grams.

2.2.2 Measuring type similarity

Although different versions of the attribute similarity algorithm involving N-grams have been discussed, we have yet to discuss the specific measure used to quantify similarity between compared attributes. This measure is known as Entropy Based Distribution (EBD), and it takes the following form:

$$EBD = \frac{H(C \mid T)}{H(C)} \qquad (1)$$

In this equation, C and T are random variables where C indicates the union of the column types C_1 and C_2 involved in the comparison and T indicates the value type. EBD is a normalized value with a range from 0 to 1, where 0 indicates the lowest EBD, or no similarity whatsoever between compared attributes, and 1 indicates the highest EBD. Our experiments involve 1:1 comparisons between attributes of compared concepts, so the value of C would simply be $C_1 \cup C_2$. H(C) represents the entropy of a set of instance values for a particular attribute (or column) while H(C|T) indicates the conditional entropy of a set of instance values associated with a particular value type.

Intuitively, EBD is a comparison of the ratio of column types for each distinct value type (conditional entropy) with column types in C (entropy). A column C contains high entropy if it is impure; that is, the ratios of column types making up C are similar to one another. On the other hand, low entropy in C exists when one column type exists at a much higher ratio than any other type. Conditional entropy is similar to entropy in the sense that ratios of column types are being compared. However, the difference is that we are finding a ratio of column types for each distinct value type. Figure 4 provides examples to help visualize the concept.

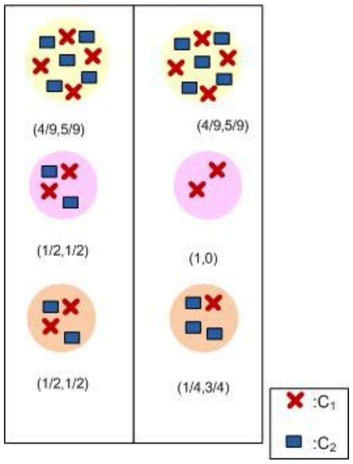


Figure 4. Distribution of column types and value types. EBD is high on the left figure since H(C) is similar to H(C|T) and it is low on right because H(C) and H(C|T) have dissimilar values

Figure 4 provides examples to help visualize the concept. In both examples, crosses indicate column types originating from C_1, while squares indicate column types originating from C_2. The value types are represented as clusters (larger circles), each of which is associated with a number of tuple values from C_1 and C_2. In the left figure, the total number of crosses is 8 and the total number of squares is 9, which implies that entropy is very high. The conditional entropy is also quite high, since the ratios of crosses to squares within 2 of the clusters are equal and nearly equal within the other. Thus, the ratio of conditional entropy to entropy will be very close to 1, since the ratio of crosses to squares is nearly the same from an overall perspective and from an individual cluster perspective. The right figure portrays a different situation: while the entropy is 1.0 (since the number of crosses is equal to the number of squares overall), the ratio of crosses to squares within each individual cluster varies considerably. One cluster features all crosses and no squares, while another cluster features a 3:1 ratio of squares to crosses. When computing the EBD value for this example, we will derive a value that is lower than the EBD for the first example because H(C | T) will be a much lower value. Intuitively, this makes sense because the ratios of value types between the compared attributes are dissimilar.

2.2.3. Algorithm for 1:M content matching

Algorithm 1 below describes our approach to 1:M matching. Let *a* be a column from concept A in O_1 that is compared with M columns $b_{1...M}$ (M <= N), where N is the total number of columns in C_2) from concept B in O_2. The algorithm for this matching is as follows:

```
Algorithm 1  Multiple Match
Input: Attribute a for concept A and Ontology O_1 and a set
attributes b_1, b_2,...b_N for concept B and Ontology O_2
Output: Determining concatenation of M attributes b from O_2
which are most similar to a from O_1
1: for each attribute b_i □ B, (0 <= i <= N )
2:     EBD ← Find_ content_ sim (a, b_i)
3:     add EBD to Similarity list SL
4: end for
5: Sort SL in descending order based on EBD
6: Pick Col_k of highest EBD from SL without replacement
7: EBD ← Find_content_ sim (a, Col_k)
8:  Repeat
9:  If SL is not empty then
10:      Col_highest ← Pick an attribute from SL with highest
                       EBD without replacement
11:      Col_k ← Concat (Col_highest , Col_k)
12:      EBD' ← Find_content_ sim (a, Col_k)
13:  else
14:      break;
15:   end if
16:  Until (EBD'-EBD) > δ
17: return Col_K
```

The algorithm takes as input one attribute *a* from concept A є O_1 and N attributes named b_1, b_2,...b_N from B є O_2. Lines 2 and 3 compute the EBD and add them to a similarity list. Line 5 sorts the list based on EBD values in decreasing order. In line 6, the algorithm picks the attribute with the largest EBD. Line 7 finds the new value of EBD for concatenated attributes of *b* and attribute *a*. In line 8, the algorithm use a loop and checks if SL is not empty so that we would be able to find another similar column with regard to EBD in greedy fashion (if exists). This loop will be finished when the difference between new EBD and previous EBD is less than a threshold or SL is empty. In other words, we could not find any new attributes that will help us to improve the EBD score.

3. EXPERIMENTS

We now present the experiments that we conducted regarding concept matching between 2 separate ontologies in the GIS domain.

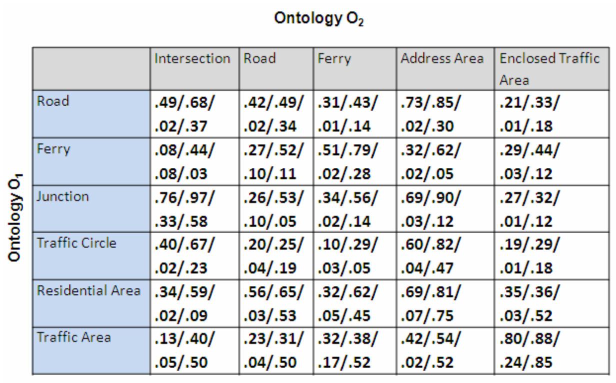
3.1 Dataset

Because data from several different areas of the United States were employed in our experiments, we effectively created a multi-jurisdictional GIS environment. The number of instances is as low as 24 (Ferry) and as high as 91059 (Junction and Intersection). Meanwhile, the number of attributes is as low as 3 (Ferry) and as high as 26 (Enclosed Traffic Area) and the geographic scope ranges from a particular city (ie. Dallas) to an entire state (Virginia).

3.2 Results

The results of the alignment of O_1 and O_2 using content similarity of the compared concepts are shown in Table 1. Each cell in the table represents a similarity calculation between one concept in O_1 and another concept in O_2, and is composed of four separate values. The first two values represent the content similarity over encoded text using TPF and DNF, respectively. The last two values represent content similarity over regular text using TPF and DNF, respectively. For example, between the concepts of Junction of O_1 and Intersection of O_2, the TPF was measured at .76, the DNF was measured at .97, the TPF was measured at .33, and the DNF was measured at .58. From the results, a number of conclusions can be drawn. First, for most of the concept comparisons, the calculated similarity values generated by using DNF, independent of the text type, are significantly higher than the values generated by TPF. These results can be explained due to the more stringent matching requirements of a value type in TPF as opposed to DNF. Keep in mind that for 2 tuples to have a matching value type in TPF, the sets of 2-grams contained within each must match exactly. If there is even one 2-gram contained in one tuple that the other tuple lacks, then the tuples will represent different value types in TPF. The end result of this situation will be that the tuples will not have any value type information in common. However, in DNF, these same tuples would be able to match on nearly all of their 2-grams, which in turn would raise the conditional entropy H(C|T) and result in a higher overall EBD value between the compared columns.

Table 1. EBD values between concepts of O_1 and O_2



Ontology O_2

Ontology O_1	Intersection	Road	Ferry	Address Area	Enclosed Traffic Area
Road	.49/.68/ .02/.37	.42/.49/ .02/.34	.31/.43/ .01/.14	.73/.85/ .02/.30	.21/.33/ .01/.18
Ferry	.08/.44/ .08/.03	.27/.52/ .10/.11	.51/.79/ .02/.28	.32/.62/ .02/.05	.29/.44/ .03/.12
Junction	.76/.97/ .33/.58	.26/.53/ .10/.05	.34/.56/ .02/.14	.69/.90/ .03/.12	.27/.32/ .01/.12
Traffic Circle	.40/.67/ .02/.23	.20/.25/ .04/.19	.10/.29/ .03/.05	.60/.82/ .04/.47	.19/.29/ .01/.18
Residential Area	.34/.59/ .02/.09	.56/.65/ .03/.53	.32/.62/ .05/.45	.69/.81/ .07/.75	.35/.36/ .03/.52
Traffic Area	.13/.40/ .05/.50	.23/.31/ .04/.50	.32/.38/ .17/.52	.42/.54/ .02/.52	.80/.88/ .24/.85

The second observation to be made from Table 1 is that the EBD values obtained over raw text were far lower than those obtained over encoded text. The reason for this is because for DNF in the case of raw text, the large increase in the number of possible 2-grams generated trivially leads to a larger number of value types between the compared columns. For TPF, all that is required to distinguish one 2-gram set from another is a single 2-gram. Consequently the number of unique sets of 2-grams generated via TPF will also rise sharply. Because of the expanded possibilities in 2-grams and 2-gram sets in raw text, there will also be far more value types present within the compared columns. This means that there is a greater possibility of unmatched types, and as a result, the conditional entropy values are more likely to be dissimilar. A final observation from Table 1 is that despite the discrepancies noted above, some sensible correlations emerge. For instance, the concepts Traffic Area and Enclosed Traffic Area share a high concept similarity based on TPF and DNF over both encoded and raw text. This is particularly evident when measuring the relative similarity values for either concept as compared to other matching concepts. The content similarity between Traffic Area and Enclosed Traffic Area using TPF over encoded text was .80, a minimum of .38 higher than other concepts for Traffic Area (with the second closest being .42 from Address Area) and .45 higher than other concepts for Enclosed Traffic Area (with the second closest being .35 from Residential Area). Notable correlations also existed between Residential Area-Address Area and Junction-Intersection.

REFERENCES

[1] T. R. Gruber (1993), "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, 5(2), 199-220.

[2] Guillermo Nudelman Hess, Cirano Iochpe, Alfio Ferrara, Silvana Castano, "Towards Effective Geographic Ontology Matching," *GeoS 2007*, pp. 51-65.

[3] William Sunna, "Multilayered Approach to Aligning Heterogeneous Ontologies",Ph.D. dissertation, University of Illinois at Chicago, 2007.

[4] William Sunna and Isabel Cruz, "Structure-based Methods to Enhance Geospatial Ontology Alignment", *Second International Conference on Geospatial Semantics*, Mexico City, Mexico, November 2007.

[5] Bing Tian Dai, Nick Koudas, Divesh Srivastava, Anthony K. H. Tung, and Suresh Venkatasubramanian, "Validating Multi-column Schema Matchings by Type," *24th International Conference on Data Engineering (ICDE)*, pp. 120-129, 2008.

Ontology Alignment Using Multiple Contexts

Jeffrey Partyka[1], Neda Alipanah[1], Latifur Khan[1], Bhavani Thuraisingham[1], Shashi Shekhar[2]

Department of Computer Science

University of Texas at Dallas[1]

University of Minnesota[2]

{jlp072000, na061000, lkhan, Bhavani.thuraisingham}@utdallas.edu[1]

shekhar@cs.umn.edu[2]

ABSTRACT

Ontology alignment involves determining the semantic heterogeneity between two or more domain specifications by considering their associated concepts. Our approach considers name, structural and content matching techniques for aligning ontologies. After comparing the ontologies using concept names, we examine the instance data of the compared concepts and perform content matching using value types based on N-grams and Entropy Based Distribution (EBD). Although these approaches are generally sufficient, additional methods may be required. Subsequently, we compare the structural characteristics between concepts using Expectation-Maximization (EM). To illustrate our approach, we conducted experiments using authentic geographic information systems (GIS) data and generate results which clearly demonstrate the utility of the algorithms while emphasizing the contribution of structural matching.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *semantic networks*, *representations (procedural and rule-based)*

General Terms

Algorithms, Measurement, Design, Reliability, Experimentation, Human Factors

Keywords

Ontology, Ontology Alignment, Schema Matching, Geographic Information Systems, Dataset

1. INTRODUCTION

Ontology alignment is the most recent incarnation of the information integration problem. A popular definition of an ontology is that of a "formal, explicit specification of a shared conceptualization", proposed by Gruber. In practice, ontologies for a given domain consist of a series of classes (or concepts) along with their properties, restrictions and instances, many of which are related by various types of relationships. The alignment of ontologies, therefore, entails deriving correspondences between concepts and their associated properties and instances.

2. PROBLEM STATEMENT AND PROPOSAL

Given 2 data sources, S_1 and S_2, each of which is represented by ontologies O_1 and O_2, the goal is to find similar concepts between O_1 and O_2 by examining their names, respective instances and structural properties. Let us assume that O_1 and O_2 are derived from the GIS domain.

The challenge involved in the alignment of these ontologies, assuming that they have already been constructed, is based on the derivation of procedures that will maximize the semantic similarity between any two concepts between the ontologies.

The ontology matching process consists of the matching of names, content and structure between compared concepts. The name match attempts to determine the degree of synonymy between the concept names. The content match determines similarity between the instances of each concept by measuring their mutual information, and it accomplishes this by the extraction of N-grams from the compared columns. The structural match determines similarity by leveraging the EM algorithm and the respective neighborhoods of all concepts to determine the most likely correspondences that occur between the ontologies. The overall similarity between two concepts is an equally weighted normalized sum of the name similarity, content similarity and structural similarity.

3. ONTOLOGY MATCHING ALGORITHM

3.1 Name Similarity

The first part of our approach attempts match concepts between two ontologies by measuring similarities between their names. The process consists of three steps. First, we check to see if an exact match exists between the compared concepts. If so, then a value of 1.0 is assigned to the name matching component of the overall similarity. If not, then we proceed with verifying whether the compared concept names are synonyms. To do this, an external dictionary such as WordNet is used to compute a semantic similarity score of the names between 0 and 1. If the words have any relation whatsoever, the semantic score returned by WordNet will represent the name matching component of the overall similarity. If there is no relation at all between the words, then the name similarity between the concepts is determined via the Jaro-Winkler string similarity metric.

3.2 Content Similarity

Content matching is accomplished by extracting instance values from the compared attributes, subsequently extracting a characteristic set of N-grams from these instances, and finally comparing the respective N-grams for each attribute. An N-gram is simply a substring of length N consisting of contiguous characters. For our experiments, the value of N was set equal to 2. The measure that was used to quantify similarity between compared attributes is known as Entropy Based Distribution (EBD), and it takes the following form:

$$EBD = \frac{H(C|T)}{H(C)}$$

In this equation, C and T are random variables where C indicates the union of the column types C_1 and C_2 involved in the comparison and T indicates the value type (2-gram for an instance value). EBD is a normalized value from 0 to 1, where 0 indicates no similarity between compared attributes, and 1 indicates that the attributes are identical. In our experiments, $C = C_1 \cup C_2$. H(C) represents the entropy of a set of instance values for a particular attribute (or column) while H(C|T) indicates the conditional entropy of a set of instance values for a particular value type.

3.3 Structural Similarity

In many situations, name and content matching are insufficient for reducing semantic heterogeneity during ontology alignment. As a result, our approach also attempts to match concepts by considering their surrounding structural characteristics. Specifically, we leverage the Expectation-Maximization algorithm to generate a mathematical model which indicates the most likely set of correspondences between concepts of O_1 and concepts of O_2. We compare all neighbors of a concept C_1 from O_1 and compare against all neighbors of a concept C_2 from O_2 to yield the structural similarity between C_1 and C_2. In adopting this algorithm, we decided to treat the concepts of each ontology as observable values while designating the set of correspondences between concepts in O_1 and O_2 as hidden values. Next, we decided that our mathematical model should be a mixture model represented by a similarity matrix SM consisting of $|O_1|$ rows and $|O_2|$ columns, where each individual entry represents an individual component of the mixture. Each entry indicates with a particular confidence value between 0 and 1 (for practical purposes, a probability value) whether or not a correspondence exists between a concept from O_1 and a concept from O_2. If a correspondence is indicated, then the entry has a value of 1, otherwise, the value is 0.

4. EXPERIMENTS

4.1 Datasets

Because data from several different areas of the United States were employed in our experiments, we effectively created a multi-jurisdictional GIS environment. GIS data assigned to concepts for O_1 is disjoint with the data assigned to the concepts for O_2. The number of instances is as low as 24 (Ferry) and as high as 91059 (Junction and Intersection). Meanwhile, the number of attributes is as low as 3 (Ferry) and as high as 26 (Enclosed Traffic Area), and the geographic scope ranges from a particular city (ie. Dallas) to an entire state (Virginia).

4.2 Results

Table 1 below shows the results of concept matching between O_1 and O_2 using name similarity, content similarity, and structural similarity via EM.

Table 1. Name + Content + Structure Similarity between concepts of O_1 and O_2

		Ontology O_2				
		Intersection	Road	Ferry	Address Area	Enclosed Traffic Area
Ontology O_1	Road	.12	.78	.13	0.00	.06
	Ferry	.10	.19	.76	.06	.11
	Junction	.38	.01	.04	.04	.04
	Traffic Circle	.40	.48	0.00	.11	.06
	Resident-ial Area	.13	.21	.06	.60	.27
	Traffic Area	.14	.25	.09	.12	.44

All of the correct correspondences between concepts of O_1 and O_2 are identified by a wide margin. Name similarity makes its strongest contribution to the accuracy of the algorithm regarding obvious correspondences such as Road-Road and Ferry-Ferry while failing to match correspondences such as Residential Area-Address Area and Junction-Intersection whose names are not similar. On the other hand, content similarity solves many of these problems by matching common N-grams existing among the instances of these concepts. While many of the correspondences are identified by name and content similarity, some, such as Traffic Circle-Intersection, remain unidentified, and others, such as Residential Area-Address Area are identified only weakly. To alleviate these problems, structure level matching via EM was applied. After doing this, correspondences that should be strong between concepts such as Residential Area-Address Area are associated with proportionally higher scores. Even in the situation where there does not exist a single correspondence that is significantly stronger than another, the composite algorithm captures the semantics appropriately. This occurs for the correspondences between Traffic Circle-Intersection and Traffic Circle-Road. Since a Traffic Circle is both a Road and an Intersection, the fact that the correspondence values are similar verifies the accuracy of our approach.

5. CONCLUSION

In this paper, we have outlined an algorithm that aligns two separate ontologies from the GIS domain using name similarity, content similarity and structural similarity. We focused on the structural similarity algorithm, which exploits EM to help determine the set of correspondences between concepts of two different ontologies. In regards to future efforts, we will expand our structure-level matching techniques to more accurately and thoroughly examine concept similarity. We will also analyze some of the more traditional techniques, such as sibling relationship similarity, and analyze its effects.

Geospatial Data Qualities as Web Services Performance Metrics

Ganesh Subbiah Ashraful Alam Latifur Khan Bhavani Thuraisingham
Department of Computer Science, Erik Johnson School of Engineering and Computer Science
University of Texas at Dallas
ganesh.subbiah@student.utdallas.edu, malam@utdallas.edu,
lkhan@utdallas.edu, bhavani.thuraisingham@utdallas.edu

ABSTRACT

Service discovery is the crucial phase in the emerging Geospatial Semantic Web to select functionally similar services for the user query. Quality of Service (QoS) based service discovery, popularly studied in traditional Web Services, applies also to Geospatial Web Services. QoS allows service clients to fine-tune their search according to their specific needs and criteria. In high-performance service-based geospatial applications, it becomes an interesting research challenge to identify geospatial parameters to further improve the search process. In this paper we have proposed a set of geospatial criteria that can be used alongside the regular QoS parameters in service discovery and invocation. We show that using this novel approach of incorporating domain-specific drill-down information in addition to the commonly used QoS parameters yield more accurate and trustable Web services platform. We use the proposed geospatial parameters as performance metrics in the experimental evaluation of our application. The parameters reflect geospatial data quality attributes already standardized and well-studied in geospatial literature.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *complexity measures, performance measures, process metrics.*

General Terms

Algorithms, Performance, Design, Reliability, Experimentation, Security.

Keywords

Geospatial Semantic Web, Semantic Web Services, Security, Trust, SOA, OWL-S, RDF

1. INTRODUCTION

Web services are increasingly seen as an invaluable part of any large-scale data query and dissemination strategy. The rise of Service Oriented Architecture (SOA) to provide intra- and inter-domain business services has ensured the rapid growth of Web services as the primary delivery platform. Business can query, find, and invoke specific services to perform their tasks instead of relying on bulky applications with superfluous features. Web services are a perfect suit for the geospatial domains since geospatial features are easy to modularize and serve to clients. As a result, clients can retrieve only the pertinent data according to their need.

Geospatial web services have been an active area of research in the context of geospatial non-interoperability problems. The collaborative effort by the industry and federal geospatial clearinghouses has focused on the standardization process to mitigate the non-interoperability problems. Although the importance of geospatial web services is well established, their efficiency is often questionable. Geospatial data tends to be voluminous even for few features; consequently on-the-fly data fetching becomes infeasible. Moreover, the data comes in various modalities even though they represent the same base facts. For instance, aerial imagery can be viewed at different resolution and vector data can be represented in different granularities. Then there is the issue of data quality that further exacerbates the efficiency of geospatial web services. To eliminate the above impediments, web services are incorporated with the Quality of Service (QoS) parameters that provide a baseline contract of what a client wants and what to expect from a service provider.

The issue of QoS has provided a major area of Web services research ([3],[4]). In [3][4] ,QoS based service selection is used to find trustworthiness of web services. The common theme in the geospatial QoS literature is to use the regular QoS parameters to efficiently exchange geospatial data [5]. The addition of domain information in the QoS values has been overlooked by researchers so far. Also there is not much work done on using geospatial specific QoS for estimating the trustworthiness of the geospatial web service

Our experience in building end-to-end geospatial web services frameworks [1,2], we have found that the client requirements revolve around four major threshold types: completeness, resolution, accuracy, and data type [6]. While there are other requirements as well, these four appear on a consistent basis. The completeness, resolution, and accuracy criteria pertain to qualitative side of geospatial data, whereas data type refers to the format of the data. Our approach is to combine these four criteria alongside the generic QoS parameters to yield a more customizable and client-centric geospatial web services platform. We refer to these four criteria as GQoS- Geospatial Quality of Service metrics.

In this paper, we propose a framework which provides a mechanism to select trustworthy geospatial web services based on geospatial quality parameters. The application is based on our work on semantically annotated geospatial web services discovery We develop an application called DAGIS (Discovery of Annotated Geospatial Information Services)[2], which we augment with GQoS and perform experimental evaluations to show its usefulness in identifying trust measures dynamically and to eliminate untrustworthy services for the query. This DAGIS framework provides a methodology to realize the semantic interoperability both at the geospatial data encoding level and also for the service framework. DAGIS is an integrated platform that provides the mechanism and architecture for building geospatial data exchange interfaces using the OWL-S Service ontology. Coupled with the geospatial domain specific ontology for automatic discovery, dynamic composition and invocation of services, DAGIS is a one-stop platform to fetch and integrate geospatial data.

The rest of the paper is organized as follows. Section 2 presents the DAGIS architecture. The proposed GQoS metrics are described in section 3. Section 4 describes service selection algorithm using GQoS parameters. The experiments are reported in section 5.

2. DAGIS PLATFORM for GEOSPATIAL SERVICES

In DAGIS ([1]), we focus on devising an improved query mechanism through semantic annotations. The application allows clients to query on a visual interface for geospatial data. The returned results can be intermingled with other types of data if requested. The results retrieved by a client can be displayed on the interface or stored on disk files. This section describes the DAGIS architecture.

2.1 Motivating Scenario

"Find movie theaters within 30 miles of 75080" is a query posed by users on current geospatial information systems and search engines. This query is an example of the type of requests carried out by service providers on the web. Service providers would often embed or layer the geospatial data in other kinds of data (e.g., medical, temporal, transactions etc.). The following sections describe how DAGIS platform handles queries of this nature.

2.2 Service Selection and Discovery

First, a query profile is generated based on the client request. The profile contains the functional and QoS metrics of the specified parameters in the client request. These requirements are used by the Matchmaker agent for selecting the appropriate service providers. In this phase, a DAGIS application module, henceforth referred to as DAGIS agent or simply agent, communicates with the Matchmaker agent for geospatial service selection (Figure 1). Prior to the service discovery, the agents of each service provider advertise the respective OWL-S service profiles to the Matchmaker. The Matchmaker in our framework does capability based reasoning using the Pellet OWL-DL reasoner. The implemented Matchmaker for this framework is based on the OWL-S MX Matchmaker, a hybrid Matchmaker that complements logic based reasoning with approximate matching based on syntactic IR based computations.

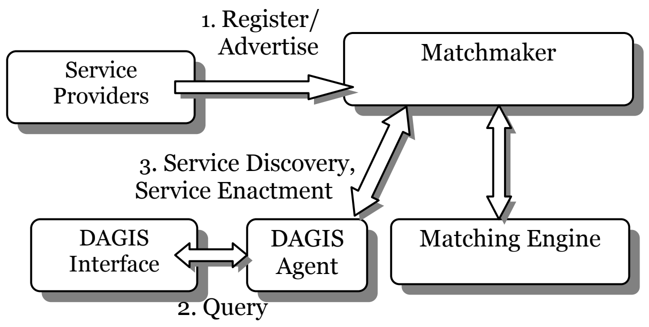


Figure 1. DAGIS System Architecture

2.3 Service Invocation

In the this phase, the DAGIS agent has the selected Service Provider's Uniform Resource Identifier (URI) from the discovery process and invokes the provider by calling one or more business methods on the URI. The service provider agent uses the same domain ontology as the DAGIS agent for semantic annotations of its services. The DAGIS agent does the invocation of the service through OWL-S grounding. The OWL-S grounding in turn uses WSDL grounding to invoke the Web Service using AXIS in our framework.

3. GEOSPATIAL QUALITY of SERVICE (GQoS)

We have proposed a set of four geospatial attributes, commonly used to specify data quality for various standards, to incorporate into our base framework (i.e., DAGIS). They augment the generic QoS parameters to allow geospatial users more precise control over their query. There are many advantages in using this approach. Traditional Web services provide the modularity but take away the ability to precisely control the use of the data. To get around this problem, one can retrieve a large amount of data from a service provider and perform offline filtering or various types of modifications themselves. However, this is a very inefficient and time-consuming procedure since a lot of processing is done post hoc. The GQoS parameters allow clients to restrict the types of service providers it is interested in before any processing on the data is done. If there is no provider available that matches the client criteria, then the client can alter the query and resubmit. These GQoS Parameters are added as OWL-DL classes to our QoS Ontology described in our previous paper [1].

In this section, we describe the following GQoS parameters: Accuracy, Resolution, Completeness, and Types.

3.1 Accuracy

Accuracy of geospatial data is defined in terms of (Attribute, Value) tuple, where attribute refers to a geographic concept/object and the Value is its measurement. We assume geospatial service providers provide data that conform to such tuples. We also assume that there is an objective assessment of all concept values. Governmental agencies, for example, would be assumed to have the most accurate object values in the event that there are multiple values for a geographic object.

3.2 Resolution

Resolution refers to the amount of detail that can be determined in space, time or theme. Both image and vector data have resolution properties. Image resolution generally refers to pixel details where more pixels per unit of an image mean better clarity. Vector data can be represented in either fine or coarse granularity. The coarser the data is, the less information is available about vector points of an object's shape. Resolution is also related to accuracy because the level of resolution affects the database specification against which accuracy is assessed.

3.3 Completeness

Completeness refers to the absence of omissions in a provider database. Completeness is distinct from accuracy in that the errors that result in lack of completeness are not incorrect encoding of object values. Instead, when a service provider fails to keep its database updated with latest data is considered to have incomplete data. For instance, the road Atlas of 2006 contains data about roads and highways built since the previous Atlas editions were published. As a result, the 2006 version would contain more complete information than the one from year 2000.

3.4 Data Types

Data types refer to the format of desired data. Even though the area of geospatial data interoperability has made a lot of progress, various reasons still exist that lead clients to request specific type of data format. For instance, although Open Geospatial Consortium (OGC) has been pushing Geography Markup Language (GML) as a standardized data exchange platform, not all geospatial applications support it. As a result, it would be rather inconvenient for a user of such an application to request data from a provider only to end up with GML data. If the user could specify that along with other requirements in the query, he can avoid spending time on retrieving useless data.

4. PROPOSED ALGORITHM to IMPLEMENT GQoS PARAMETERS

During the Semantic Service Discovery Phase[2.2], the query profile of the user is submitted to the matchmaker for determining the functional matches from the set of published services. The Matchmaker returns a set of functionally similar services if the query to be solved involves single service provider; otherwise returns a dynamically composed service if the query requires service orchestration.

4.1 New Service Discovery Algorithm

To incorporate the GQoS values, we add a step to the DAGIS service discovery algorithm. The new algorithm operates as follows.

1. Service providers publish profiles to Matchmaker
2. Generate query profile
3. Find semantically similar services for the query using the functional parameters :input and output parameters
4. If there is no such service from step 3, dynamically compose complex service using the services registered using DAGIS Composer Algorithm [2]
5. Sort the Functionally Similar Semantic Services using the GQoS Algorithm (see Figure 3)
6. Return the URI of the best Service from step 5 to user

We will describe the approach developed by us for performing the Step 5 of the service discovery algorithm. The QoS selection differs when we have a dynamic composition that involves computing the aggregate QoS values of the services dynamically, which is also one of our contribution in this paper.

4.2 GQoS Algorithm

Interaction Model: The Environment is comprised of registered service providers S_1, S_2 … Sj, , Users U_1, U_2 … U_i, matchmakers M_1, M_2 … M_k. In our interaction model we assume only one matchmaker. We employ special monitoring services which get the user reports on QoS relevance feedback which are called Trust Monitors TM_1, TM_2 … TM_l. Matchmaker can also additionally act as Trust Monitor,

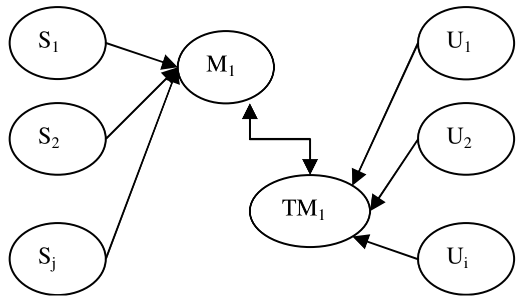


Figure 2. Interaction Model

```
User Query List UQ = {(uq_1,r_1), (uq_2,r_2) ….(uq_n,r_n)}
TargetMatch // Number of concept matches required
G_val = 0 for all services
1. ∀S_j in Functional Match Set F
2. dist = 0.0
3.   ∀q_i:q_i=quality concept in uq
4.     If q_i matches with a concept in sq_j
5.       conceptmatch = conceptmatch +1
6.         dist += |r_i – p_i|
7.     If concept match >= TargetMatch then
8.       G_val =  diff/conceptmatch
9. Return F sorted by ascending order of
   G_val scores.
```

Figure 3: GQoS Similarity Match Algorithm

Service providers publish their QoS values (sq_1,p_1) , (sq_2,p_2) , … where (sq_i, p_i) are vector pairs of concepts and their values. Users provide the QoS requirements for every query as (uq_1,r_1), (uq_2,r_2) , … where (uq_i ,r_i) are vector pairs of concepts and user required values. GQoS vector values pis , r_i are fuzzy values which are in the range [1,5] . 1 is the worst GQoS support available and 5 is the best support available for that GQoS parameter.

In the First Phase, for each registered Service Provider j in the functional match set F of the Query Q, a G_{val} is evaluated using the advertised QoS parameters. G_{val} is the Manhattan distance averaged over the number of quality concept matches between the user requirement and the service provider advertised GQoS values.

$$Gval = \sum (r - p) / conceptmatch$$

The GQoS Similarity Match Algorithm is illustrated in figure 3 to select a set of services. All the Service providers are set with Gval = 0 and the target concept matches between query and service provider concept is set to a constant (is 3 in our experiments). In Step 1 for every service Sj returned from Functional Set F returned from Matchmaker. The similarity between r and p vectors is meausred using Manhattan Distance. For every quality concept qi in Vector uq , if there is a ConceptMatch (exact, subsumes) with a concept in sqj, conceptmatch is incremented. The diff is updated for this match, In step 7 we check if there are at least target number of matches for meeting the user requirement, we compute the Gval as average distance over the concept matches in step 8. Step 9 returns the F in ascending order of Gval.

In the second phase of the GQoS measurements, we use the user feedback to update the advertised GQoS parameters of the selected service S_i as follows. All the user reports pertaining to the similar query Q posed is aggregated here in this phase. The user feedback list UF of every user is evaluated as shown in Figure 4.

In our model, user reports are considered to be credible as only authenticated users of the system can log on to the system for service discovery. The evaluation of the credibility of user values reported is not in scope of our work. We assume that the Service Providers who publish their service descriptions to the matchmaker do not cancel their registration during the interaction for at least a certain number of iterations (say 10) to facilitate the catching of untrusted providers. In future, we would maintain logs of the interactions to capture these cancellation scenarios also.

```
Aggregate Feedback Vector FV;
For every Service Provider $S_j$
1. Read every User Feedback List $U_iF$ received = {($uq_1$,$f_1$), ($uq_2$,$f_2$) ….($uq_n$,$f_n$)} where i=1:n
2.  FV = FV + {(uq1,f1), (uq2,f2) ….(uqn,fn)}
3. End For
4. FVavg = FV / n
5. Update each QoS parameter sqj  of Sj as
        pj =  pj  (1 – Fv) + Pj
```

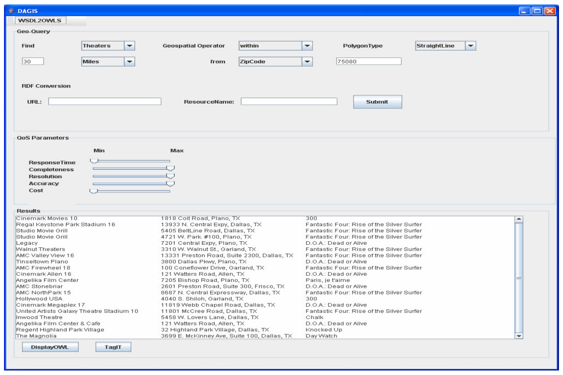
Figure 4: GQoS Propagation Algorithm

5. EXPERIMENTAL EVALUATION

The experiment and evaluation results are to be shown during the demo at the poster session.

6. CONCLUSION

In this paper, we have successfully proposed geospatial data parameters which are used in the automatic service discovery for emerging semantic enabled geospatial web. The framework proposed and implemented helps to distinguish the untrustworthy service providers by penalizing them using the performance metrics evaluated by keeping the user in the loop. We are working on further experiments which show the increase in the precision and relevance measures due to these proposed geospatial quality metrics. This work provides an intuitive way to select trustworthy semantic web services using the geospatial data quality parameters along with QoS measures which is novel step towards the building geospatial web of trust.



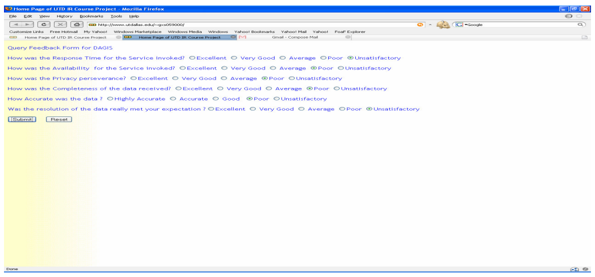
Find: Theaters; Geospatial Operator: within; PolygonType: StraightLine
30 Miles from ZipCode 75080

URL: ResourceName:

ResponseTime
Completeness
Resolution
Accuracy
Cost

Cinemark Movies 10	1818 Coit Road, Plano, TX	300
Regal Keystone Park Stadium 16	13933 N. Central Expy, Dallas, TX	Fantastic Four: Rise of the Silver Surfer
Studio Movie Grill	5405 BeltLine Road, Dallas, TX	Fantastic Four: Rise of the Silver Surfer
Studio Movie Grill	4721 W. Park. #100, Plano, TX	Fantastic Four: Rise of the Silver Surfer
Legacy	7201 Central Expy, Plano, TX	D.O.A.: Dead or Alive
Walnut Theaters	3310 W. Walnut St., Garland, TX	Fantastic Four: Rise of the Silver Surfer
AMC Valley View 16	13331 Preston Road, Suite 2300, Dallas, TX	Fantastic Four: Rise of the Silver Surfer
Tinseltown Plano	3800 Dallas Pkwy, Plano, TX	D.O.A.: Dead or Alive
AMC Firewheel 18	100 Coneflower Drive, Garland, TX	Fantastic Four: Rise of the Silver Surfer
Cinemark Allen 16	121 Watters Road, Allen, TX	D.O.A.: Dead or Alive
Angelika Film Center	7205 Bishop Road, Plano, TX	Paris, je t'aime
AMC Stonebriar	2601 Preston Road, Suite 300, Frisco, TX	D.O.A.: Dead or Alive
AMC NorthPark 15	8687 N. Central Expressway, Dallas, TX	Fantastic Four: Rise of the Silver Surfer
Hollywood USA	4040 S. Shiloh, Garland, TX	300
Cinemark Megaplex 17	11819 Webb Chapel Road, Dallas, TX	D.O.A.: Dead or Alive
United Artists Galaxy Theatre Stadium 10	11801 McCree Road, Dallas, TX	Fantastic Four: Rise of the Silver Surfer
Inwood Theatre	5458 W. Lovers Lane, Dallas, TX	Chalk
Angelika Film Center & Cafe	121 Watters Road, Allen, TX	D.O.A.: Dead or Alive
Regent Highland Park Village	32 Highland Park Village, Dallas, TX	Knocked Up
The Magnolia	3699 E. McKinney Ave, Suite 100, Dallas, TX	Day Watch

Appendix A Dagis Semantic Query Interface


Home Page of UTD IR Course Project - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Customize Links Free Hotmail My Yahoo! Windows Marketplace Windows Media Windows Yahoo! Bookmarks Yahoo! Mail Yahoo! FoaF Explorer
Home Page of UTD IR Course Project
Home Page of UTD IR Course Project
Gmail - Compose Mail
Google
Submit
Reset

http://www.utdallas.edu/~gxs059000/

Query Feedback Form for DAGIS

How was the Response Time for the Service invoked? Excellent Very Good Average Poor [✓ Unsatisfactory]

How was the Availability for the Service invoked? Excellent Very Good Average [✓ Poor] Unsatisfactory

How was the Privacy perseverance? Excellent Very Good Average [✓ Poor] Unsatisfactory

How was the Completeness of the data received? Excellent Very Good Average [✓ Poor] Unsatisfactory

How Accurate was the data ? Highly Accurate Accurate Good [✓ Poor] Unsatisfactory

Was the resolution of the data really met your expectation ? Excellent Very Good Average Poor [✓ Unsatisfactory]

Appendix B User Feed back Form for a Query

7. REFERENCES

[1] Ashraful Alam, Ganesh Subbiah and Bhavani Thuraisingham: Reasoning with Semantics-aware Access Control Policies for Geospatial Web Services. 2006 ACM Workshop on Secure Web Services (SWS),

[2] Ashraful Alam, Ganesh Subbiah, Latifur Khan, and Bhavani Thuraisingham: DAGIS: A Geospatial Semantic Web Services Discovery and Selection Framework, Second International Conference on geospatial semantics GeoS2007

[3] Maximilien, E. M. and Singh, M. P. 2004. Toward autonomic web services trust and selection. In Proceedings of the 2nd international Conference on Service Oriented Computing ,ICSOC '04. ACM Press.

[4] Le-Hung Vu, Manfred Hauswirth, and Karl Aberer: QoS-based Service Selection and Ranking with Trust and Reputation Management,2005 International Conference on Cooperative Information Systems (CoopIS), Nov 2005

[5] Wen-jun Li, Shu-neng Zhao, Heng Sun, Xiao-bin Zhang, "Ontology-Based QoS Driven GIS Grid Service Discovery," *skg*, p. 49, Second International Conference on Semantics, Knowledge, and Grid (SKG'06), 2006

[6] Veregin H and Hargitai P 1995 An evaluation matrix for geographical data quality. In Guptill S C and Morrison J L (eds) Elements of spatial data quality. Oxford: Elsevier 167-188.

DAGIS: A Geospatial Semantic Web Services Discovery and Selection Framework

Ashraful Alam, Ganesh Subbiah, Latifur Khan, and Bhavani Thuraisingham

Department of Computer Science,
University of Texas at Dallas, Dallas, TX 75083
{malam,ganesh.subbiah,lkhan,bhavani.thuraisingham}@utdallas.edu

Abstract. The traditional Web services architecture uses a keyword based search to match a query to one or more service providers. However, a world-to-word matching to discover a service provider is too simplistic for geospatial data and fails to capture matches that advertise their functionality using domain-dependent terminology. In this paper, we present DAGIS (Discovering Annotated Geospatial Information Services) – a semantic Web services based framework for geospatial domain that has graphical interface to query and discover services. It handles the semantic heterogeneities involved in the discovery phase and we propose algorithms for selecting the best service through QoS (Quality of Service) based semantic matching. The framework is capable of performing dynamic compositions on the fly through a back chaining algorithm. The framework is evaluated by solving queries posed by users in various geospatial decision making scenarios.

1 Introduction

Geospatial data plays a pivotal role in value-added content exchange between software agents or amongst people. The ability to provide additional dimensions to otherwise monotonic information has led to an enormous increase in the use of geospatial services. A rather underrated aspect behind such an escalation is the fact that spatially-aware data is more amenable to human cognition than strictly textual information. A far more appreciated aspect is that the integration of diverse data types with geospatial sources has yielded practical business and research benefits. Medical data overlapped with digital maps provides wealth of information in forecasting epidemics; population research centers can trace genealogical data over a region to discover social trends and so forth. This growing interest and activity level in the geospatial domain is further edified by more than 232 million hits on Google ™ for the keyword 'geospatial.' Geospatial data is characterized by multitude data formats and data models and integration of this valuable data is crucial for the businesses and applications on the World Wide Web. But lack of a common unified framework for discovery, collection, and dissemination of geospatial data is characterized by the coherent heterogeneities present at both the syntactic and the semantic level.

Web Services driven Service Oriented Architecture model provides a mechanism to handle the syntactic heterogeneities to an extent for geospatial data sources. The current geospatial standards recommended by OGC- a flagship consortium that specifies standards for describing the geospatial data and services are founded on these principles of providing geospatial data interoperability. On the other side, emergence of semantic web and its associated technologies which aims to transform the web data sources into intelligent knowledge repositories that will use web agents to reason and infer information in more sophisticated manner. Semantic Web technologies provide strikingly similar standards for better interoperability of data and services with less human intervention for the World Wide Web. This prompted the researches from both the communities towards the vision of geospatial semantic web for realizing semantic interoperability of geospatial data. The recent OGC geospatial semantic web interoperability experiments are a major step towards this vision.

It's argued by researches Semantic interoperability is an important goal but hard to pin down due to lack of common accepted formal specifications. Kuhn [13] establishes that Service Signatures needs to be semantically annotated to achieve semantic interoperability but the challenge of annotating proper semantics for web services description and automatic discovery is imminent.

In this paper, we propose DAGIS – Discovery of Annotated Geospatial Information Services framework for building geospatial semantic web services using the OWL-S Service ontology coupled with the geospatial domain specific ontology for automatic discovery, dynamic composition and invocation. The algorithms developed for this framework enables semantic matching of functional and non-functional services during each phase of Service Orientation. In addition, our approach makes use of [2] since its hybrid mechanism seems to produce better results.

There has been major work done on geospatial data interoperability. Vckovski et al. [7] and Goodchild et al. [8] address various interoperability issues related to spatial data processing of vectors and graphics, semantics, heterogeneous databases and representation. OGC identified that the key to solve interoperability issues are through the interface of software components where data and its operations are inseparable. This resulted in syntactic specification for geospatial data exchange through Geography Markup Language [9]. Operations on features in GML are implemented through web services [1]. Web Feature Service (WFS), Web Map Service (WMS), Web Coverage Service (WCS) are the core standards for Web services being developed by OGC to allow distributed geo-processing systems to provide complex services.

The rest of the paper is organized as follows. Section 2 presents the DAGIS architecture, its automatic discovery mechanism, dynamic composition algorithms and the invocation mechanism. Section 3 presents QoS based service selection. Finally, section 4 presents complex queries.

2 DAGIS Framework

Integration of geospatial and non-geospatial information tasks involves separate data sources and service providers. Executing the tasks with minimal human intervention is the motivation behind our proposed architecture. The implementation of the architecture -- called DAGIS -- focuses on devising improved query mechanisms through automated reasoning using a domain specific ontology. We have built DAGIS as a prototype application that is useful for finding information for local businesses over a geographical region. We have identified the major phases in developing this framework. These phases are discussed in the following sections.

DAGIS provides an immediate advantage over other web 2.0 and GIS based map solutions. The latter products have limitations when the following types of queries are encountered: “Find Movie Theaters between Richardson, TX and Irving, TX”. This geospatial query is commonly posed by users looking for local information around the geographical regions of interest. Current solutions do not recognize the semantics of the geospatial operator “between” in this query. We posed this query on Google Maps and observed that it is oblivious to the presence of such operators.

2.1 DAGIS System Architecture

DAGIS system architecture is described in this section. Functionality of each of the components is addressed through a running example. We distinguish the layers that constitute an end-to-end query execution and result display. The major layers are the presentation layer, semantic middleware layer and the ontology data layer. DAGIS Framework has major components at each of this layer.

DAGIS Query Browser Portlet: In the presentation side, the DAGIS query browser portal gets the user query. We have developed a Java™ portlet that provides the required interface for the query.

DAGIS Agent: DAGIS agent, placed at the semantic middleware layer, fetches the query parameters from the user. We can deploy multiple DAGIS agents in this layer. In our current application we describe the behavior of a single DAGIS agent. This agent communicates with the DAGIS Matchmaker using OWL-S (formerly known as DAML-S) [5] service ontology language. It automatically constructs an OWL-S query for the given user query.

DAGIS Matchmaker: DAGIS Matchmaker is the component that performs semantic matching between the submitted queries and the semantic web service providers present in the registry. It performs both functional and non functional based selection and service discovery.

DAGIS Composer: DAGIS Composer dynamically builds service chain to solve the user query when there is no single service provider available to match user query requirements. This dynamic composition is done automatically and the composed service URI is returned back to the Matchmaker.

OWL-S Registry: The semantic web services are stored in this registry, which acts like a catalogue of useful services.

WSDL Registry: The WSDL registry is any standard UDDI or public web services registry such as www.x-methods.net and www.salcentral.com.

WSDL2OWLS Converter: This converter converts the WSDL service description file to OWL-S file. The XSLT conversions are currently done manually, but in the future there would be full fledged automatic conversion package.

Figure 1 shows how the aforementioned components fit into the DAGIS framework. Initially a user requests for service through a query browser (i.e., portlet). DAGIS agent receives the query and forwards it to a matchmaker. The matchmaker inquires the OWL-S registry to determine a match. The matchmaker is responsible for talking to the domain ontologies through a common OWL-S API and performing the semantic interpretation of the terms. Figure 1 also shows the separation of layers based on their functional requirements. The presentation layer allows the client to actually input the query. Then we have the middleware layer that allows interchangeable components to provide meta-service related functionality such as service search and reasoning. It is important that the middleware layer is not tied to a special platform or architecture. It should be abstracted in a way so that other layers do not have dependency on the underlying details of the middleware components. This abstract also encourages extensibility by swapping in and out modules to fit one's needs. The third layer consists of the ontologies including the service and domain ontologies. We describe the workflow of the DAGIS architecture in more details in the following sections.

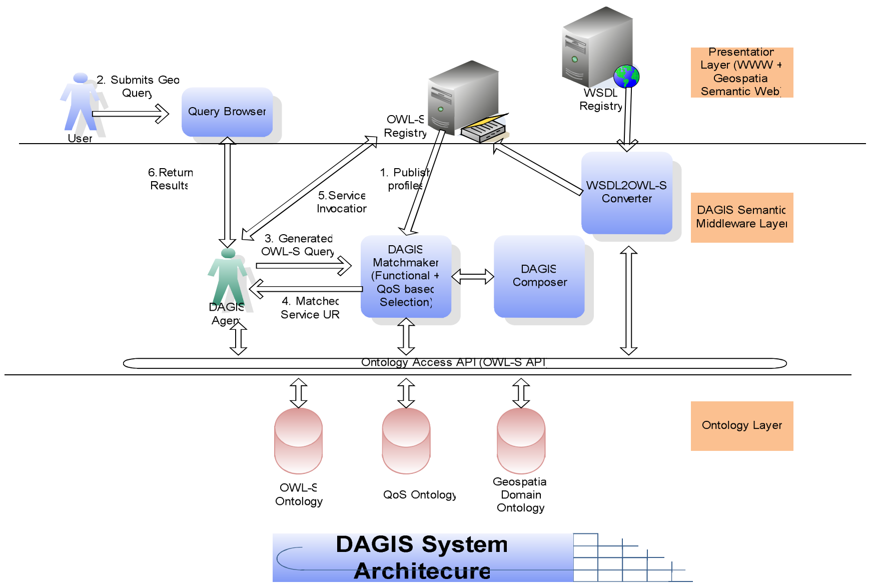


Fig. 1. DAGIS system architecture

2.2 Geospatial Ontology Development Phase

In our work, we have developed geospatial service ontology to describe concepts used by geospatial web services. The concepts defined in our ontology were developed in accordance with OGC Web Services Specification Architecture. The QoS ontology developed is described along with QoS selection process. Figure 2 shows the snapshot of our geospatial ontology developed for DAGIS. The businesses are categorized under the geocoder results class. City, Latitude, Map, State, Zip code are also subclasses of this class. The different kinds of geospatial web services are categorized under the main class OGCSemanticWebServices. These subclasses are Feature Handling Services and Mapping Services. Web Feature Service like Gazetteer Service is part of Feature Handling Service. Coverage Portrayal Service, Feature Portrayal Service, Web Map Services are subclasses of Mapping Services.

2.3 Automatic Semantic Query Profile Generation

After the user submits the query, it is disambiguated using our developed ontology; subsequently, an OWL-S service profile is automatically generated. In the next step the query profile is used by the DAGIS Agent for service discovery and selection of the service providers that will solve this query.

The DAGIS Agent uses this semantic profile for selecting the appropriate service provider from the matchmaker agent. The following figure shows a snapshot of the profile for a simple query: 'Find Movie Theaters within 30 miles of zip code 75080'. The profile of this OWL-S file has input ZipCode, distance 30 miles and output required is movie theaters. Figure 3 shows the query profile generated by DAGIS agent in response to the user query.

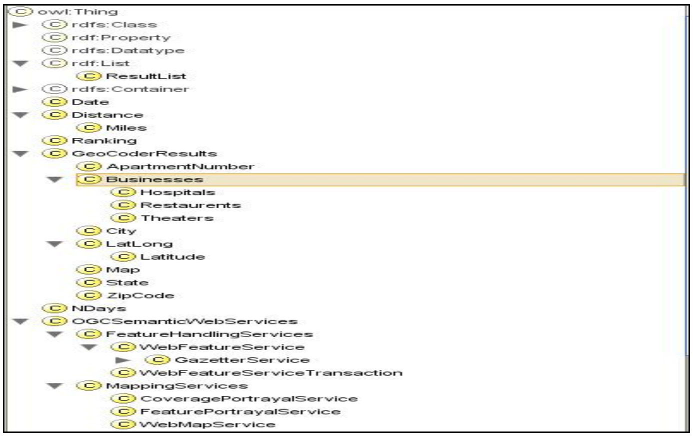


Fig. 2. Snapshot of geospatial service ontology

2.4 Geospatial Service Selection and Discovery

The service selection based on the functional and non-functional requirements of the generated query profile is used by the DAGIS Matchmaker agent for selecting the appropriate service providers. The Matchmaker in our framework does capability based reasoning using the Pellet OWL-DL reasoner. Our implementation of the Matchmaker for this framework is developed by extending the OWL-S MX Matchmaker [2]. It is Java™-based and uses Pellet for logic based filtering. It also uses loss-of-information, extended Jacquard, and Jensen-Shannon information divergence based similarity metrics for complementary approximate matching. We extend this hybrid matchmaker to handle service selection based on QoS. There are different degrees of matches based on the similarity. The similarity criteria form a

```
    <profile:Profile rdf:about="#QueryProfile">
    <profile:hasInput>
    <process:Input rdf:ID="ZipCode">
    <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">http://127.
0.0.1/Ontology/OGCServiceontology.owl#ZipCode</process:parameterTyp
e>
    </process:Input>
    </profile:hasInput>
    <profile:hasOutput>
    <process:Output rdf:ID="Movie Theaters">
    <process:parameterType
rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI"
    >http://127.0.0.1/Ontology/OGCServiceontology.owl#MovieTheaters</
process:parameterType>
    </process:Output>
    </profile:hasOutput>
    </profile>
```

Fig. 3. Generated query profile

lattice based on how relaxed the similarity is. EXACT match is least relaxed and FAIL is most relaxed.

3 QoS Based Service Selection

The QoS based automatic service selection plays a crucial role in the matchmaking process when there is more than one registered service provider providing similar functionalities. In our proposed system, trust calculations are established through capability based matching of the QoS parameters. QoS parameters are the nonfunctional attributes that aid in the dynamic service discovery and selection. This facilitates the dynamic computation of the trust for the service provider and selection can be made for a suitably trusted service by the client.

Our architecture is based on agent-based trust framework where the different QoS parameters characterized under various dimensions for describing the quality are captured in the client profile and the providers' profiles. The proposed geospatial services ordering metric (GSOM) for QoS evaluation and for establishing trust is described in the following section.

3.1 QoS Ontology

Our QoS ontology is developed in line with the upper and middle ontologies as described in [11]. This facilitates modular development and can easily be extended for our geospatial domain concepts defined in the geospatial ontology. The main concepts in the QoS ontology are:

- Quality: Representing the measurable nonfunctional concept of a service.
- QAttribute: The value of a quality concept is determined by the type of QAttributes that constitute that concept.

- QMeasurement: This described the measurement of quality which can be subjective or objective
- QRelationship: For describing relationship between two or more quality concepts.

During the service discovery phase, the query profile of the user is submitted to the matchmaker for determining the functional matches from the set of published services. The Matchmaker returns a set of functionally similar services if the query to be solved involves single service provider; otherwise, it returns a dynamically composed service. To incorporate the QoS based selection, we add a step to this service discovery process. The new algorithm operates as follows.

1. Service providers publish profiles to Matchmaker
2. User submits query and corresponding semantic query profile is generated
3. Find semantically similar services for the queBry using the functional parameters – that is, the input and output parameters
4. If there is no such service from step 3, dynamically compose complex service using the services registered using DAGIS composer algorithm
5. Sort the functionally similar semantic services using the GQoS Algorithm
6. Return the URI of the best service from step 5 to user

We will describe the approach developed by us for performing the step 5 of the above service discovery algorithm. The QoS selection differs when we have a dynamic composition. In that case, it involves computing the aggregate QoS values of the services dynamically, which is one of our contributions in this paper.

3.2 QoS Selection Algorithm

Interaction Model: The environment is comprised of registered service providers S_1, S_2 … Sj, users U_1, U_2 … U_i, matchmakers M_1, M_2 … M_k. In our interaction model we assume only one matchmaker. We employ special monitoring services that get user reports on QoS relevance feedback called trust monitors TM_1, $TM_{2\,...}$ TM_l.

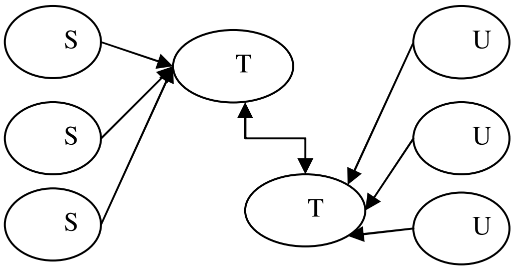


Fig. 4. Interaction model

Service providers publish their QoS values (sq_1, p1), (sq_2, p2) …where (sq_i, p_i) are vector pairs of concepts and their values. Users provide QoS requirements for every query as (uq_1,r_1), (uq_2,r_2) , … where (uq_i ,r_i) are vector pairs of concepts and user required values (see Figure 4). During feedback loop, users submit their feedbacks as

U_{1j}, U_{2j}, U_{3j} where j is index for the service provider j selected during each query iteration process.

In the first phase, for each registered service provider j in the functional match set F of the query Q, a G_{val} is evaluated using the advertised QoS parameters. The QoS similarity matching algorithm is illustrated in Figure 5. All the service providers are initially set with $G_{val} = 0$ and the target concept matches between query and service provider concept are set to 3. In step 1 for every service Sj a functional set F is returned from the Matchmaker. The aggregated difference in the user expected and provided values is stored in diff, which was initially set to 0. For every quality concept q_i in Vector uq, if there is a concept match (exact, subsumes etc.) with a concept in sq_j, ConceptMatch is incremented. The diff is updated for this match. In step 7 we check if there are at least target number of matches for meeting the user requirement; then we compute the G_{val} as average diff in step 8. Step 9 ensures that as G_{val} is updated through propagation algorithm (discussed next), when it goes above the threshold T, service s_j is considered to be untrustworthy and removed from set F. Step 11 returns the F in ascending order of G_{val}.

In the second phase we use the user feedback to update the advertised GQoS parameters of the selected service S_i as follows. For every query Q posed by U_i , C_{ij} is the conformance value vector submitted by U_i for S_j to TM_l. The satisfaction of the user on each QoS parameter he had specified is measured qualitatively through Cij on a fuzzy scale. This is used to get the weighted expectation vector (Uij * Cij) of a user. The feedback vector is used to update the P_i of Service S_i in step 4 in QoS propagation algorithm (Not reported here). In our model, user reports are considered to be credible only for authenticated users of the system, who log on to the system for service discovery. We assume that the service providers that publish their service descriptions to the matchmaker do not cancel their registration during the interaction for at least a certain number of iterations. The current model sets a hard number on the lower bound of the provider availability period to determine untrustworthy providers. The period is defined in terms of the number of iterations a provider was available for the Matchmaker. Right now this number is 10, but in the future we will maintain logs of the interactions to capture these cancellation scenarios also.

4 Complex Queries Using DAGIS

The scenario described in section 2.1 is a relatively simple one that involves selection of a single service provider. Real world scenarios often involve complex queries that necessitate dynamic composition of different service providers. To explain the complexities further, we restate the example from section 2.1. Consider the following query *"Find movie theaters within 30 miles of Richardson?"*.

We use the DAGIS visual interface to drive the user query, thereby bypassing the need to parse natural language based queries. Based on the client query profile a search is performed in service registry to discover matching OWL-S profiles. Since there is no service that takes city as input and returns movie theaters within a certain radius, the matchmaker resorts to decomposing the query into multiple atomic processes using DAGIS decomposer algorithm. Decomposing the query into two atomic parts results in a successful Web service execution since there is a profile that

```
User Query List UQ = {(uq_1, r1), (uq_2, r2) …. (uq_n, rn)}
TargetMatch = 3
G_val = 0 for all services
findSimilarityMatch()
1. ∀S_j in Functional Match Set F
2. diff = 0.0
3. ∀q_i:q_i=quality concept in uq
4. If q_i matches with a concept in sq_j
5.       conceptmatch = conceptmatch +1
6. diff += |p_j-r_i|
7. If concept match >= TargetMatch
8.       G_val =  diff/conceptmatch
9. If G_val > T
10.    remove S_j from F.
11. Return F sorted by ascending order  of G_val scores.
```

Fig. 5. QoS similarity match algorithm

outputs zip-codes given a city and there is a second profile that outputs movie theaters given a zip-code. The Compose Sequencer component constructs the composite service.

4.1 DAGIS Composition and Sequencing Algorithm

The composer and sequencer algorithms in this section are based on the Recursive Back Chaining algorithm proposed in [12]. To construct the service chain, our algorithm is recursively called for each likely service available in the service registry. A service is selected only if its output is equivalent to desired output of the requesting client. We also have a sequencer algorithm that provides composite process chaining for non-atomic processes. This algorithm uses a trivial bind function to create a mapping between input and output parameters of two processes (a hash map can be used to represent the mapping data structure in the actual implementation).

4.2 Service Invocation

In this phase, the DAGIS Agent has the selected service provider's OWL-S URI from the discovery process and invokes the service provider. In this scenario, the selected service has an Atomic Process – GetTheaterProcess. As the service provider agent also uses the same domain ontology as the DAGIS Agent for semantic annotations of its services. This is the major benefit of sharing the semantic concepts using a unified ontology framework. The DAGIS agent does the invocation of the service through OWL-S grounding. The OWL-S grounding then uses WSDL grounding to invoke the Web Service using AXIS in our framework. The OWL-S API used in this system provides the execution engine and monitoring environment to monitor the process execution and for exception handling.

References

1. Sondheim, M., Gardels, K., Buehler, K.: GIS Interoperability. In: Longley, P., Goodchild, M., Maguire, D., Rhind, R. (eds.) Geographical Information Systems 1 Principles and Technical Issues, John Wiley & Sons, New York (1999)
2. Klusch, M., Fries, B., Sycara, K.: Automated Semantic Web Service Discovery with OWLS-MX. In: AAMAS. Proceedings of 5th International Conference on Autonomous Agents and Multi-Agent Systems, Hakodate, Japan, ACM Press, New York (2006)
3. Srinivasan, N., Paolucci, M., Sycara, K.: An Efficient Algorithm for OWL-S Based Semantic Search in UDDI. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 96–110. Springer, Heidelberg (2005)
4. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing Semantics to Web Services: The OWL-S Approach. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 26–42. Springer, Berlin (2005)
5. OWL-S: Semantic Markup for Web Services. W3C member submission (2004), available: http://www.w3.org/Submission /OWL-S/
6. Egenhofer, M.: Toward the Semantic Geospatial Web. In: Voisard, A., Chen, S.-C. (eds.) ACM-GIS, McLean (2002)
7. Včkovski, A., Brassel, K.E., Schek, H.-J. (eds.): Interoperating Geographic Information Systems. In: Včkovski, A., Brassel, K.E., Schek, H.-J. (eds.) INTEROP 1999. LNCS, vol. 1580, Springer, Berlin (1999)
8. Goodchild, M.F., Egenhofer, M., Feeas, R., Kottman, C.: Interoperating Geographic Information Systems. Proceedings of Interop 1997, Santa Barbara, CA, Norwell, MA. Kluwer, Dordrecht (1998)
9. Geography Markup Language (GML) version 3.1.1 Specification, available: http://opengis.net/gml/
10. Sirin, E., Bijan Parsia, B.: The OWL-S Java API. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, Springer, Heidelberg (2004)
11. Li, L., Horrock, I.: A Software Framework for Matchmaking Based on the Semantic Web Technology. In: Proceedings of 12th Int Conference on the World Wide Web, Workshop on E-Services and the Semantic Web (2003)
12. Renier Gibotti, F., Gilberto, R.: GeoDiscover – a Specialized Search Engine to Discover Geospatial Data in the Web. In: 7th Brazilian Symposium on GeoInformatics (2005)
13. Kuhn, W.: Geospatial Semantics: Why, of What, and How? In: Spaccapietra, S., Zimányi, E. (eds.) Journal on Data Semantics III. LNCS, vol. 3534, pp. 1–24. Springer, Heidelberg (2005)

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

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

[1] The University of Texas at Dallas
Richardson, Texas, U.S.A.
{sxr063200, axg089100, lkhan, bxt043000}@utdallas.edu

[2] Raytheon 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, mapping constructs, and algorithms, 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 fledgling 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 mapping preliminaries and relationships handled. R2D's system architecture and algorithms 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.

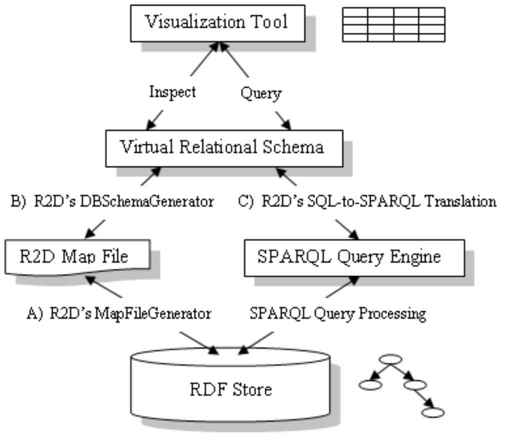

Visualization Tool
Inspect
Query
Virtual Relational Schema
B) R2D's DBSchemaGenerator
C) R2D's SQL-to-SPARQL Translation
R2D Map File
SPARQL Query Engine
A) R2D's MapFileGenerator
SPARQL Query Processing
RDF Store

Figure 1. R2D System Architecture

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.

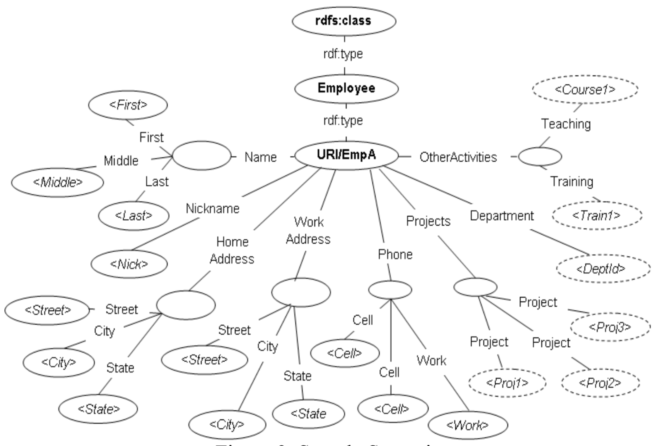


Figure 2. Sample Scenario

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:keyField, 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 (MVSLBN): 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 a MVSLBN.*

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. RDF 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:MultiValued{Simple/Complex}ResourceBlankNode (MVSRBN and MVCRBN): Duplicate simple/complex resource blank nodes are represented using the MVSRBN and MVCRBN 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 MVSRBN.*

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).

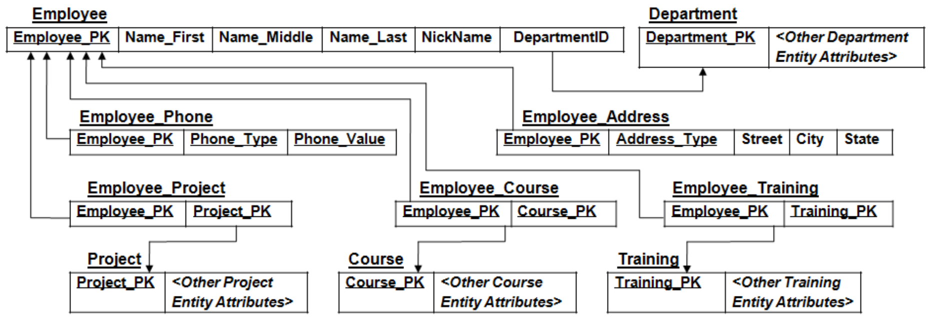


Figure 3: Equivalent Relational Schema for the Sample Scenario in Figure 2

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

R2D CONSTRUCTS	**RDBMS RELATIONSHIP**
r2d:ColumnBridge	Column *(Example: <Nickname>)*
r2d:SimpleLiteral BlankNode *(Ex: <Name>)*	Column *(Ex: <First>, <Middle>, <Last>)*
r2d:Complex LiteralBlankNode *(Ex: <Phone>)*	Multi Valued Attribute (resulting in a new table (that includes a TYPE column) for the 1:N relationship)
r2d:[Simple/ Complex] ResourceBlank Node *(Ex:<Projects>, <OtherActivities>)*	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)
r2d:MultiValued {Simple/Complex} (Literal/ Resource) BlankNode *(Ex: Home/WorkAddress)*	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)
r2d:MixedBlank Node	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:MixedBlankNode)
r2d:refersToTableMap	Foreign Key *(Ex: <DeptID>)*
r2d:MultiValued Predicate	"Type" column in parent table *(Ex: Phone_Type* for *<Phone>)*

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

OWL/RDFS CONSTRUCTS	**RELATIONAL CONCEPT**
rdfs:class	Table
rdf:property	Column
rdf:domain	Table that the rdf:property is a column of
rdf:range	Datatype of the column
rdf:type predicate	Values of Primary Key column of the 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) then, for each predicate off of the blank Node, the ProcessLiteralPredicate procedure is called which works as described above. 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) 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, department WHERE employee.employee_PK = employee_Phone.employee_PK and employee_Phone.Phone_Type = <http://Phone/Cell> and employee.department_id = department.department_id AND (name_First LIKE 'ABC%' OR employee_pk = <http://empl/123>);

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<*Index*>" variable where *Index* 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 = <http://Phone/Cell> && employee_department_id = subject1 && (regex(?name_First, "^ABC") || ?subject0 = http://empl/123) }

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 ?subject*i* 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-valued 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 <http://empl/Name> ?employee_name .
?employee_Name <http://Name/First> ?name_First .
?employee_Name <http://Name/Last> ? name_Last .
?subject0 <http://empl/deptId> ?employee_department_id .
?subject1 <http://dept/dept_name> ?department_name .

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 *Lecture* 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.

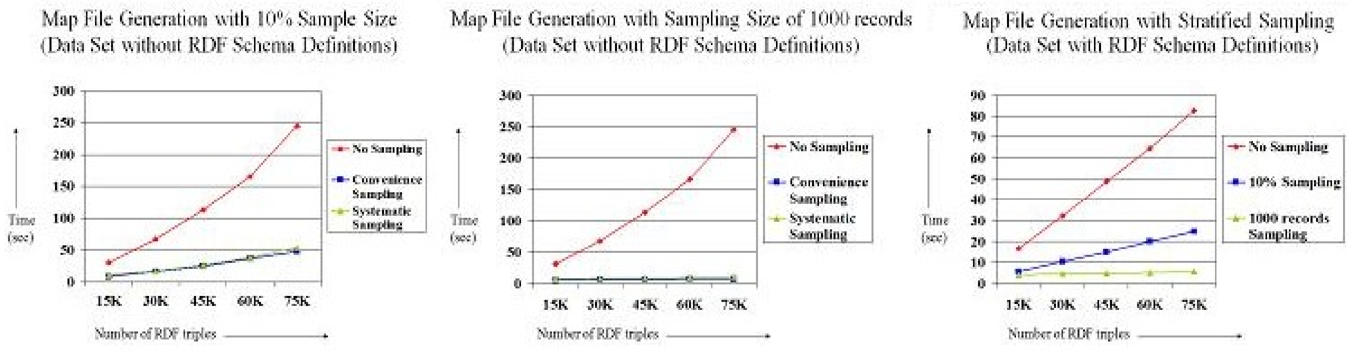
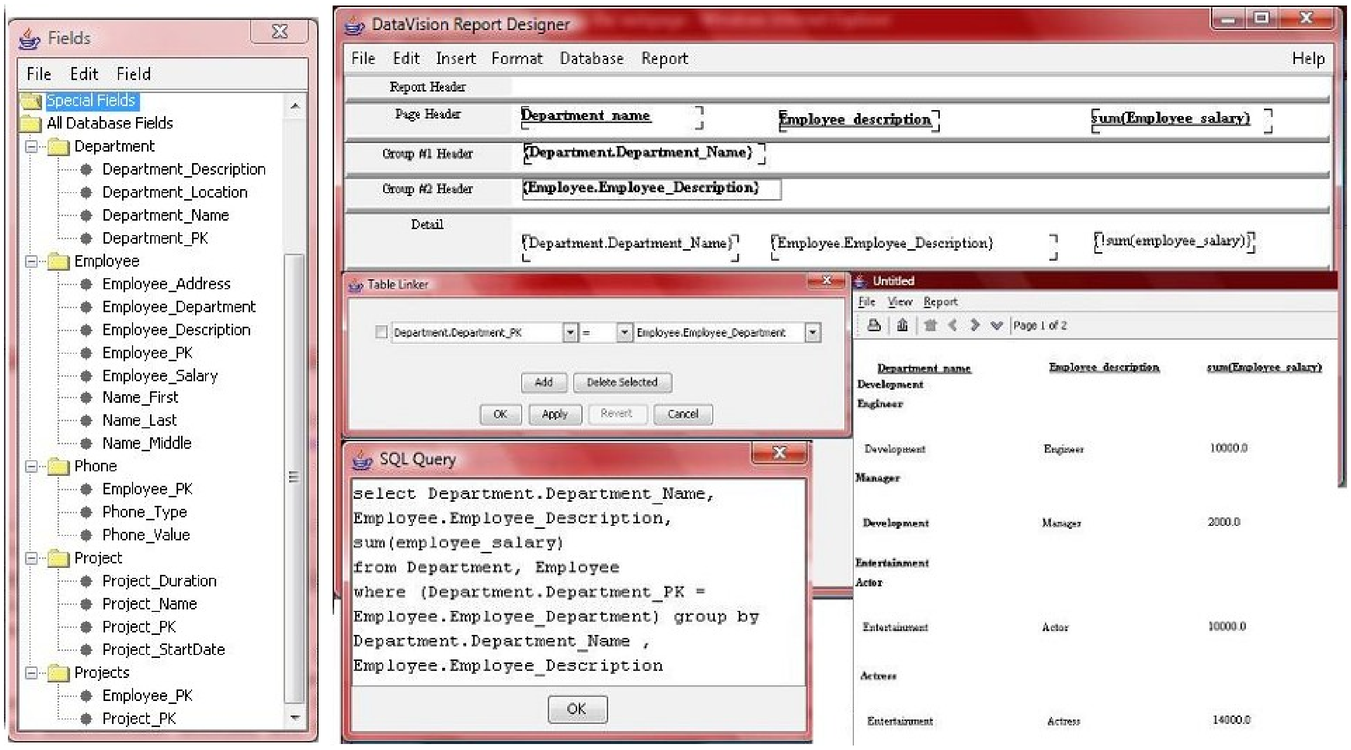


Figure 4. Map File Generation Times



```
Special Fields
All Database Fields
Department
    Department_Description
    Department_Location
    Department_Name
    Department_PK
Employee
    Employee_Address
    Employee_Department
    Employee_Description
    Employee_PK
    Employee_Salary
    Name_First
    Name_Last
    Name_Middle
Phone
    Employee_PK
    Phone_Type
    Phone_Value
Project
    Project_Duration
    Project_Name
    Project_PK
    Project_StartDate
Projects
    Employee_PK
    Project_PK
```

```
Page Header:     Department name    Employee description    sum(Employee salary)
Group #1 Header: {Department.Department_Name}
Group #2 Header: {Employee.Employee_Description}
Detail:          {Department.Department_Name}    {Employee.Employee_Description}    {!sum(employee_salary)}
```

```
Department.Department_PK = Employee.Employee_Department
```

```
select Department.Department_Name,
Employee.Employee_Description,
sum(employee_salary)
from Department, Employee
where (Department.Department_PK =
Employee.Employee_Department) group by
Department.Department_Name ,
Employee.Employee_Description
```

```
  Department name        Employee description     sum(Employee salary)
Development
Engineer

  Development            Engineer                 10000.0
Manager

  Development            Manager                  2000.0

Entertainment
Actor

  Entertainment          Actor                    10000.0

 Actress

  Entertainment          Actress                  14000.0
```

Figure 5. Equivalent Relational Schema as seen by DataVision and DataVision's Report Designer

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.

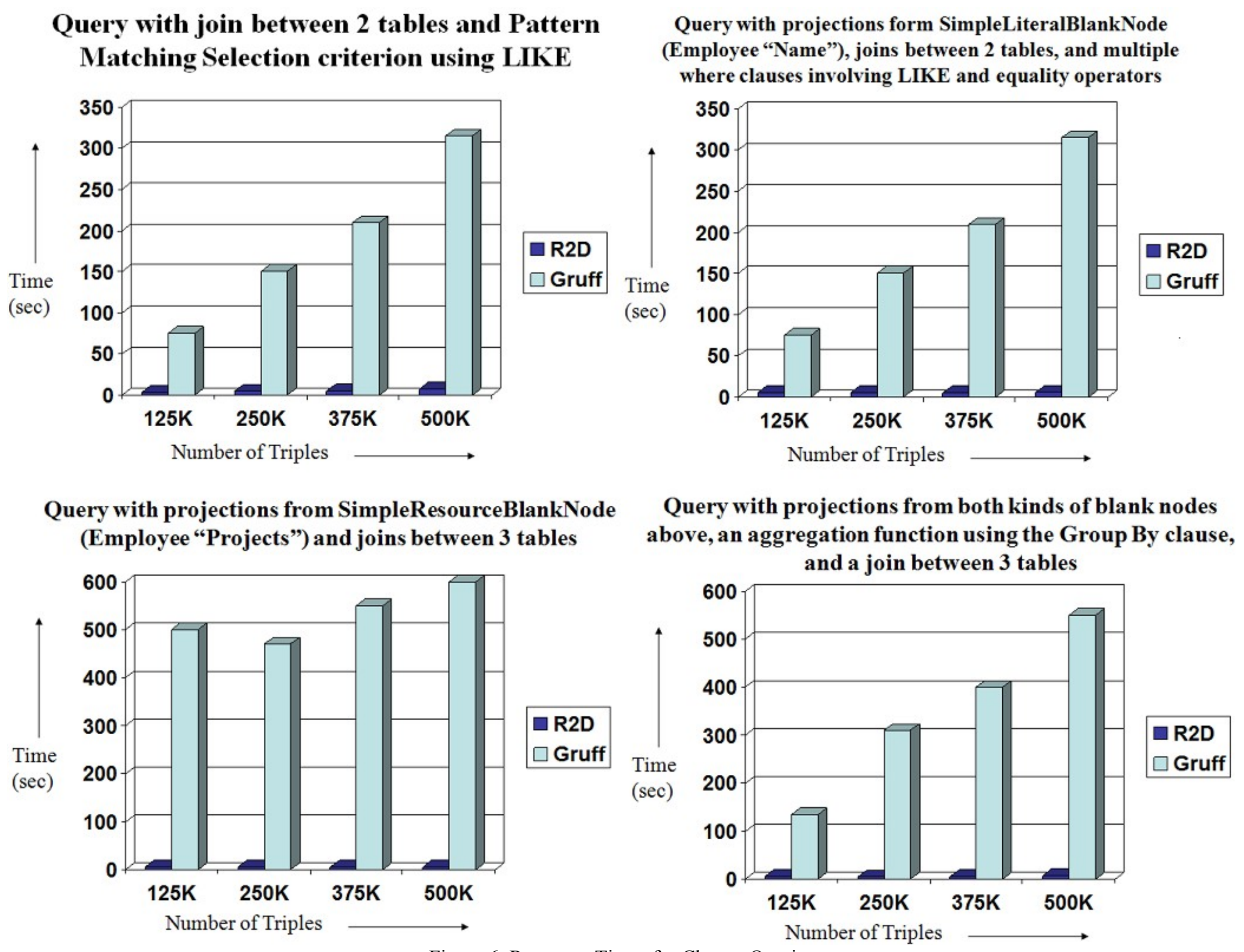


Figure 6. Response Times for Chosen Queries

REFERENCES

[1] W3C Recommendation, "RDF Primer", Feb, 2004. http://www.w3.org/TR/rdf-primer/.

[2] A.Muys, "Building an Enterprise-Scale Database for RDF Data", 2006, http://www.netymon.com/papers/muysa06buildforrdf.pdf

[3] W.Teswanich, S,Chittayasothorn, "A Transformation of RDF Documents and Schemas to Relational Databases", IEEE PacificRim Conferences on Communications, Computers, and Signal Processing, 2007, pp. 38-41.

[4] J.Hendler, "RDF Due Diligence", 2006. http://civicactions.com/blog/rdf_due_diligence.

[5] S.Ramanujam, A.Gupta, L.Khan, S.Seida, and B.Thuraisingham, "Relationalizing RDF Stores for Tools Reusability", 18th International World Wide Web Conference, 2009, pp. 1059-1060.

[6] S.Ramanujam, A.Gupta, L.Khan, and S.Seida, "R2D: Extracting relational structure from RDF stores", International Conference on Web Intelligence, 2009, in press.

[7] C.Bizer, and A.Seaborne, "D2RQ – Treating Non-RDF Databases as Virtual RDF Graphs", 3rd International Semantic Web Conference, 2004.

[8] O.Erling, and I.Mikhailov, "RDF Support in the Virtuoso DBMS", 1st Conference on Social Semantic Web, 2007, pp. 1617-5468.

[9] L.Han, T.Finin, C.Parr, J.Sachs, and A.Joshi, "RDF123: From Spreadsheets to RDF", International Semantic Web Conference, LNCS 5318, 2008, pp.451-466.

[10] S.Auer, S.Dietzold, J.Lehmann, S.Hellmann, and D.Aumueller, "Triplify – Light-Weight Linked Data Publication from Relational Databases", 18th International World Wide Web Conference, 2009, pp. 621-630.

[11] Z.Pan, and J.Heflin, "DLDB: Extending Relational Databases to Support Semantic Web Queries", Practical and Scalable Semantic Systems, 2003, pp.109-113.

[12] H.Chen, Z.Wu, H.Wang, and Y.Mao, "RDF/RDFS-based Relational Database Integration", 22nd International Conference on Data Engineering, 2006, pp.94-104.

[13] E.I.Chong, S.Das, G.Eadon, and J.Srinivasan, "An Efficient SQL-based RDF Querying Scheme", *31st International VLDB Conference,* 2005, pp. 1216-1227.

[14] Y.Yan, C.Wang, A.Zhou, W.Qian, L.Ma, and Y.Pan, "Efficiently Querying RDF Data in Triple Stores", 17th International Conference on World Wide Web, 2008, pp. 1053-1054.

RDFKB: Efficient Support For RDF Inference Queries and Knowledge Management

James P. McGlothlin
The University of Texas at Dallas
800 West Campbell Road
Richardson, TX 75080
jpm083000@utdallas.edu

Latifur R. Khan
The University of Texas at Dallas
800 West Campbell Road
Richardson, TX 75080
lkhan@utdallas.edu

ABSTRACT
RDFKB (Resource Description Framework Knowledge Base) is a relational database system for RDF datasets which supports inference and knowledge management. Significant research has addressed improving the performance of queries against RDF datasets. Generally, this research has not addressed queries against inferred knowledge. Solutions which do support inference queries have done so as part of query processing. Ontologies define the rules that govern inference for RDF datasets. These inference rules can be applied to RDF datasets to derive additional facts through methods such as subsumption, symmetry and transitive closure. We propose a framework that supports inference at data storage time rather than as part of query processing. The dataset is increased to include all knowledge whether explicitly specified or derived through inference with a negligible overhead. Queries against inferred data are simplified, and performance is increased.

Categories and Subject Descriptors
H.2.1[**Database Management**]: Logical Design- *data models, schema and subschema.*

General Terms.
Design, Performance.

Keywords
Semantic Web, Resource Description Framework, Data Models, Information Integration and Retrieval, Logic and Databases, Knowledge Base Management Systems, Ontology.

1. INTRODUCTION
A knowledge base can be defined as "a special kind of database for knowledge management, providing the means for the computerized collection, organization and retrieval of knowledge." [9] The goal of RDFKB is to provide solutions to convert RDF datasets into knowledge bases, while preserving efficient performance. In order to support the retrieval of knowledge, and not simply the recording of facts, the knowledge base must enable answering queries that require inference and deductive reasoning.

A simple example of inference is the well-known logical syllogism from the ancients Greeks:

All men are mortal.
Socrates is a man.
Therefore, Socrates is mortal.

The core strategy of RDFKB is to apply all known inference rules to the dataset to determine all possible knowledge, and then store all this knowledge. Inference is performed at the time the data is stored rather than at query time. When new data is added to the database, we execute the inference engine and attempt to determine and store all additional facts that can be inferred. In the syllogism example above, we would actually add "Socrates is mortal" to the database. Thus, the dataset contains all known knowledge, and it can be directly queried; there is no need for further inference at query execution time.

The World Wide Web Consortium [1] defines the RDF data format as the standard mechanism for describing and sharing data across the web. All RDF datasets can be viewed as a collection of triples, where each triple consists of a subject, a property and an object. OWL (The Web Ontology Language)[10] defines rules for inference. In OWL, the following constructs are some examples of ontology rules that will allow the inference of new RDF triples from existing knowledge: rdfs:subClassOf, owl:equivalentClass, owl:equivalentProperty, owl:sameAs, owl:inverseOf, owl:TransitiveProperty, owl:SymmetricProperty, rdfs:subPropertyOf, and owl:intersectionOf. RDF and OWL have "provable inference" and "rigorously defined notions of entailment"[11]. Therefore, the semantics of the RDF documents and ontology concretely define what information can be known.

The key elements of RDFKB's solution are

a) Inference is performed at storage time and inferred triples are persisted
b) Data is stored in relational databases using efficient schema

Performing inference at storage time simplifies queries and improves query performance[15]. Using relational databases for storage provides efficiency and robustness[3][7][15]. The simplest way to store data in a relational database is to simply create a three column table of RDF triples. However, research shows that this is not the most efficient solution[4][5]. Many alternative solutions have been proposed including vertical partitioning[4], sextuple indexing[6], RDF-3X[15], and

RDFJoin[8]. None of these solutions store inferred data or address the task of querying inferred knowledge. Inference queries against these databases require detailed knowledge and encoding of the ontology logic, and require unions and joins to consolidate the inferred triples. There are existing solutions that perform inference in memory[14], providing simpler queries. There are even solutions which support inference and relational database storage[14][15], but they have fixed schema and do not support customized tables for efficiency. Such solutions pay a large query performance penalty due to increasing the dataset to include the inferred triples. RDFKB supports inference at storage time combined with efficient database schema in a manner that not only simplifies queries but also improves performance.

Traditionally, databases have sought to avoid storing redundant information. Instead, we propose to *increase* redundancy. There are two reasons why traditional databases and knowledge bases have not attempted to store such inferred knowledge, and have, instead, supported inference as part of query processing. The first is that transactional databases seek to reduce redundancy in order to preserve and enforce referential integrity. However, semantic web databases should not be viewed as traditional transactional databases. RDFKB supports adding new knowledge to the database, but we are not concerned with supporting transactions that delete or change triples in the dataset. Such updates can still be performed, but in these instances the inference is recalculated. The second reason is that it has been considered too expensive and broad a search to attempt to infer all possible knowledge. At the time of query execution, there is information about which knowledge relates to the query, and this can be used to limit the scope of the inference search. However, we are concerned with query performance rather than the performance of adding knowledge to the database. With our solution, all inferred knowledge can be stored without a query performance penalty.

The remainder of this paper is organized as follows: In Section 2, we provide background information on technology used by our project. In Section 3, we specify our architecture. In Section 4, we present implementation solutions for several types of inference that are defined in OWL. In Section 5, we evaluate queries defined by the Lehigh University Benchmark (LUBM)[2] that involve inference, and we specify implementation solutions and performance results for these queries. In Section 6, we examine related research. In Section 7, we present areas for future work, and we make some conclusions.

2. BACKGROUND

RDFKB relies on adding inferred RDF triples to the dataset. This would not be a viable solution unless these triples can be added and stored without increasing the performance cost of queries. The database schema documented in *RDFJoin: A Scalable Data Model for Persistence and Efficient Querying of RDF Datasets* [8] allows us to add new inferred triples to the dataset without a performance penalty. Therefore, while RDFKB's design is not contingent upon using this particular database storage scheme, our solution is implemented and tested using the RDFJoin technology.

RDFJoin stores the RDF triples in five base tables: the SOIDTable, the PropertyIDTable, the POTable, the SOTable and the PSTable. The SOIDTable and the PropertyIDTable simply map each unique property, subject or object to a sequential id number. Each URI or literal appears in the table only once regardless of how many times it appears in the dataset. The PSTable includes three columns: the PropertyID, the SubjectID and the ObjectBitVector. The POTable contains three columns: the PropertyID, the ObjectID and the SubjectBitVector. The SOTable contains three columns: the SubjectID, the ObjectID and the PropertyBitVector.

The bit vectors are the key to the RDFJoin approach, and the reason why RDFJoin is able to store inferred triples at little or no cost. The length of the ObjectBitVector in the PSTable and the SubjectBitVector in the POTable is equal to the max(SOID) in the SOIDTable. Each and every unique subject or object URI or literal in the dataset has a corresponding bit in these bit vectors. In the PSTable, there is a bit in the bit vectors for each unique URI, indicating whether that URI appears as an object in a triple with the corresponding property and subject. Similarly, in the POTable, there is a bit in each SubjectBitVector to indicate if each subject URI appears with that property and object in a triple in the dataset.

Figure 1 shows an example dataset and some of the corresponding RDFJoin tables. The subject of the first triple in the dataset is "UTD". According to the SOIDTable, the SubjectID for UTD=1. The property of the first triple is "fullName", which has PropertyID=1 according to the PropertyIDTable. The object, "The University of Texas at Dallas", has SOID=2 from the SOIDTable. Therefore, the PSTable has a tuple with PropertyID=1, SubjectID=1 and the 2nd bit in the ObjectBitVector set on. SOID=3 would indicate "Richardson". There is no triple <UTD fullName Richardson> therefore the 3rd bit in the bit vector is 0. However, if we were to add such a triple, it would not increase the number of bits stored, it would only change this third bit to 1.

Our inference solution relies upon adding additional triples for all inferred knowledge. Typical RDF databases incur a performance penalty for increasing the number of triples in the data set. However, except for a potential reduction in the achievable compression rate, RDFJoin does not incur this penalty. Instead, RDFJoin incurs a performance penalty only when the dataset's vocabulary is increased. RDFJoin queries experience a performance reduction in a linear fashion corresponding to the number of unique URIs and literals in the dataset. This is because there is already a bit in the tables for each and every possible combination of the known subjects, properties and objects. Many times, inferred triples will not introduce unique URIs at all, and if unique URIs are introduced, they are only unique for that ontology rule. While an RDF dataset may include millions of triples, the number of unique terms in the ontology is generally not large. For our experiments, in Section 5, we utilize the LUBM (Lehigh University Benchmark)[2] dataset with more than 44 million RDF triples. For this dataset, 20,407,385 additional triples are inferred, yet only 22 unique URIs are added by this inference. Thus, there is no performance penalty for the addition of the millions of inferred triples, only for the addition of 22 new URI terms. For these reasons, the RDFJoin solution provides a database scheme that allows inferred triples to be added to the dataset at almost no performance cost.

EXAMPLE DATASET:

<UTD, fullName, The University of Texas at Dallas>
<UTD, locationCity, Richardson>
<UTD, locationState, TX>
<ComputerScience, subOrganizationOf, UTD>
<James McGlothlin, worksFor, ComputerScience>
<James McGlothlin, position, GraduateStudent>
<Latifur Khan, worksFor, ComputerScience>
<Latifur Khan, position, Professor>
<James McGlothlin, position, ResearchAssistant>
<James McGlothlin, advisedBy, Latifur Khan>
<James McGlothlin, takesCourse, CS7301>
<Latifur Khan, teacherOf, CSC7301>
<CSC7301, fullName, Data Mining>
<James McGlothlin, authorOf, RDFJoin>
<Latifur Khan, authorOf, RDFJoin>

PropertyIDTable

fullName	1
locationCity	2
locationState	3
subOrganizationOf	4
worksFor	5
position	6
advisedBy	7
takesCourse	8
teacherOf	9
authorOf	10

SOIDTable

UTD	1
University of Texas At Dallas	2
Richardson	3
TX	4
ComputerScience	5
James McGlothlin	6
Graduate Student	7
Latifur Khan	8
Professor	9
Research Assistant	10
CSC7301	11
Data Mining	12
RDFJoin	13

PSTable

PropertyID	SubjectID	Objects (bit vector)
1	1	0100000000000
1	11	0000000000010
2	1	0010000000000
3	1	0001000000000
4	5	1000000000000
5	6	0000100000000
5	8	0000100000000
6	6	0000001001000
6	8	0000000010000
7	6	0000000100000
8	6	0000000000100
9	8	0000000000100
10	6	0000000000001
10	8	0000000000000

POTable

PropertyID	ObjectID	Subjects (bit vector)
1	2	1000000000000
1	12	0000000000010
2	3	1000000000000
3	4	1000000000000
4	1	0000100000000
5	5	0000010100000
6	7	0000010000000
6	9	0000000100000
6	10	0000010000000
7	8	0000010000000
8	11	0000010000000
9	11	0000000100000
10	13	0000010100000

Figure 1: RDFJoin Tables and Example Dataset

3. ARCHITECTURE

The core of the RDFKB design is that for each RDF triple, we infer all possible additional RDF triples, store this data, and make it accessible to queries.It is not the intention of this project to attempt to identify and implement each and every possible rule of inference. There are many such rules defined by the RDF ontologies, and these can change and be added to over time. Moreover, there may be inference rules that are not specified in the ontology at all. Specific domains may have business logic that includes domain-specific inference. For these reasons, we allow each inference rule to execute independently and to store its own inference data.

RDFKB defines a global function *add(subject, property, object).* This function encapsulates the details of our schema from the user. The user does not have to have any knowledge of the database tables and schema to add triples to the dataset. Furthermore, the database user does not have to have any knowledge concerning inference. The simple act of adding triples to the dataset invokes the inference process without any intervention by the user.

RDFKB's architecture is that each inference rule registers a function *add(subject, property, object).* When a triple is added, these functions are each executed, and every inference rule has the opportunity to derive and add more triples. The order that the inference rules are executed is not relevant. The only change the inference rule is allowed to make to the database is to add more triples by calling the global *add* function.

This is a recursive solution that only requires a one level inference search. As an example, assume there is an inference rule to support subClassOf. Assume an ontology that defines Steak as a class that is a subClassOf Meat, and Meat as a class that is a subClassOf Food. Now, if we add a triple <FiletMignon type Steak>, the inference rule will add another triple <FiletMignon type Meat>. Then, the same inference rule will be executed on this new triple, and add another triple <FiletMignon type Food>.

One issue that should be addressed is whether there are any unwanted effects of adding extra triples to the dataset. Usually, enforcing distinction in the dataset will negate the effect of additional triples; RDF triples only appear once in the dataset. While it is possible that two different inference rules will add the same RDF triple (for example if James is a student and an employee, both of these facts can be used to infer James is a person), this duplication has no effect. The query does not have to address this issue, because it is handled when the inference is executed, at addition time. There are also times when a query will not want to perform any inference or query inferred data. For example, a cardinality query such as “How many jobs does John have?” should not take in to account the fact that his employer is part of a transitive suborganization hierarchy. Also, sometimes we may simply want to know the exact type an instance is declared as, and not include all the further information that can be inferred. Generally, the responsibility to know that inference is not sought rests with the query. Therefore, our architecture creates a second copy of all tables, including all triples from the dataset but ignoring all triples added by the inference rules. These tables are simply labeled with the extension _Without_Inference. A query can retrieve information from these tables when inference should be specifically avoided.

4. ONTOLOGY

This section presents details and examples for OWL ontology inference rules.. Many of the OWL examples contained in this section are taken directly from the *OWL Web Ontology Language Reference*[10] and from the OWL ontology file for the Lehigh University Benchmark (LUBM)[2]. For each of these rules, we have implemented and registered an inference rule with RDFKB. The inference rule simply checks if the condition of the rule is met, and if so determines and adds the triple on the right side of the equation. For example, with subClassOf (Section 4.1), the condition is that property='type' and object=ClassA, and the action is *add(subject, type, ClassB)* where ClassA is a subclass and ClassB is its superclass in the ontology.

4.1 subClassOf

Rule: if ClassA subClassOf ClassB then <Subject type ClassA> → <Subject type Class B>	
Ontology: <owl:Class rdf:ID="Corvette" <rdfs:subClassOf rdf:resource="SportsCar" /> </owl:Class>	
Base triple: <Vehicle1 type Corvette>	*Inferred triple:* <Vehicle1 type SportCar>

4.2 equivalentClass

Rule: if ClassA equivalentClass ClassB then <Subject type ClassA> ➔ <Subject type ClassB> && <Subject type ClassB> ➔ <Subject type ClassA>
Ontology: <owl:Class rdf:about="US_President"> <equivalentClass rdf:resource="Commander_in_Chief"/> </owl:Class>
Base triple: <BarackObama type US_President>
Inferred triple: <BarackObama type Commander_in_Chief>

4.3 subPropertyOf

Rule: if property1 subPropertyOf property2 then <Subject property1 Object> ➔ <Subject property2 Object>
Ontology: <owl:ObjectProperty rdf:ID="hasMother"> <rdfs:subPropertyOf rdf:resource="#hasParent"/> </owl:ObjectProperty>
Base triple: <GeorgeWBush hasMother BarbaraBush>
Inferred triple: <GeorgeWBush hasParent BarbaraBush>

4.4 TransitiveProperty

Rule: if property1 is TransitiveProperty then <Subject property2 Object1> && <Subject property1 Object2> ➔ <Subject property2 Object2>
Ontology: <owl:TransitiveProperty rdf:ID="subRegionOf"> <rdfs:domain rdf:resource="#Region"/> <rdfs:range rdf:resource="#Region"/> </owl:TransitiveProperty>
Base triples: <Texas subRegionOf UnitedStates> <Austin locatedIn Texas> <UnitedStated subRegionOf NorthAmerica>
Inferred triples: <Austin locatedIn UnitedStates> <Texas subRegionof NorthAmerica> <Austin locatedIn NorthAmerica>

4.5 SymmetricProperty

Rule: if property1 is SymmetricProperty then <Subject property1 Object> ➔ <Object property1 Subject>	
Ontology: <owl:SymmetricProperty rdf:ID="friendOf"> <rdfs:domain rdf:resource="#Human"/> <rdfs:range rdf:resource="#Human"/> </owl:SymmetricProperty>	
Base triple: <Jane friendOf John>	*Inferred triple:* <John friendOf Jane>

4.6 inverseOf

Rule: if property1 is inverseOf property2 then <Subject property1 Object> ➔ <Object property2 Subject>
Ontology: <owl:ObjectProperty rdf:ID="hasChild"> <owl:inverseOf rdf:resource="#hasParent"/> </owl:ObjectProperty>
Base triple: <GeorgeWBush hasParent BarbaraBush>
Inferred triple: <BarbaraBush hasChild GeorgeWBush>

4.7 intersectionOf

Rule: determine the subsumption resulting from intersection,

and add this subsumption as though an actual subclass was specified.

Ontology:

```
<owl:Class rdf:ID="Employee">
  <rdfs:label>Employee</rdfs:label>
  <owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#Person" />
  <owl:Restriction>
        <owl:onProperty rdf:resource="#worksFor" />
        <owl:someValuesFrom>
              <owl:Class rdf:about="#Organization" />
        </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Base triple:	*Inferred triple:*
<James type Employee>	<James type Person>

5. QUERY IMPLEMENTATION AND EXPERIMENTAL RESULTS

Lehigh University Benchmark (LUBM) [2] dataset has been used to evaluate the performance of many RDF data storage solutions including Hexastore[6] and RDFJoin[8]. LUBM defines queries that require inference. In prior research, the approach has been to either avoid the queries that include inference or to support these queries by translating them into unions of subqueries without inference. RDFKB supports these queries as defined, without requiring the query to have any knowledge of the inference. This section includes performance results and analysis for each of these queries. The LUBM queries that require inference are queries 3, 5, 6, 11, and 13; therefore, these are the queries implemented and tested here.

All of our experiments here were performed on a system with an Intel Core 2 Duo CPU @ 2.80 GHz, with 8GB RAM, running 64 bit Windows. Our code was developed in Java, C++, and SQL, and we tested with the MonetDB column store database. We created a database using the LUBM dataset with 400 universities and 44,172,502 tuples. After all inference rules are applied, this dataset grows to 64,579,887 triples. In our graphs, we show the number of original triples in the dataset; inferred triples are not included. This makes the comparison to other technologies more accurate, as they do not store inferred triples.

There is no high-performance solution with inference that uses relational databases for RDF datasets. It seems uninformative to compare our results with solutions that access and parse RDF documents directly, and the performance of such tools is slower by many orders of magnitude. Therefore we chose two relational database RDF solutions to compare with: vertical partitioning[4] and RDFJoin[8]. Neither of these solutions provides automated support for inference. Instead, the query implementation requires specific knowledge and encoding of the ontology logic. These queries require a minimum of 4 subqueries and 3 unions, and a maximum of 29 subqueries.

We additionally tested with a single triples table sorted by property, subject, object and subject, property, object. The triple store implementation consistently performed slower than vertical partitioning or RDFJoin. Therefore, we chose not to document these results in order to avoid saturating the graphs.

5.1 LUBM Query 3

(type Publication ?X)
(publicationAuthor ?X http://www.Department0.University0.edu/AssistantProfessor0)

Publication has a wide hierarchy in the LUBM ontology. Therefore, to perform this query would normally involve querying many different subclasses of Publication and unioning the results together. Without RDFKB, this query in SQL requires 10 unions to support all the different classes in the Publication hierarchy. With our inference solution, this query involves selecting two subject bit vectors and executing a single *or* operation.

In the actual dataset, LUBM defines all publications as type Publication, and does not in fact ever define an instance of any of its subclasses. If we were to assume this knowledge, RDFJoin would actually be as fast or faster than RDFKB. Since we cannot assume this knowledge, RDFJoin is slightly slower due to the time needed to attempt to query these subclasses. This query is a subject-subject join. In the vertical partitioning solution, subject-subject are linear merge joins, so vertical partitioning is fairly efficient. Figure 2 shows the performance results for RDFKB, RDFJoin and VP (Vertical Partitioning) for Query 3.

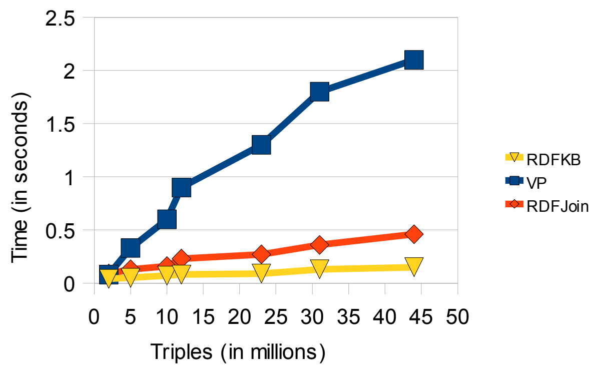


Figure 2: Performance Results for LUBM Query 3

5.2 LUBM Query 5

(type Person ?X)
(memberOf ?X http://www.Department0.University0.edu)

This query invokes five kinds of inference rules: subClassOf, subPropertyof, inverseOf, TransitiveProperty and intersectionOf. Person is the superclass of a wide variety of subclass types. memberOf has many subproperties, and has an inverse: member.

There are 21 classes in the Person hierarchy in the LUBM ontology. In the LUBM dataset, there are instantiations for 8 of these classes. Therefore, to query ?x type Person using vertical partitioning or RDFJoin requires 21 subqueries; 8 of which will return results that must be unioned together. There are three different ways to express memberOf in the dataset. It can be expressed directly with the memberOf property, or through the subProperty worksFor, or through the inverse property member. To perform this query with vertical partitioning or RDFJoin requires that triples with all three properties be queried. To make matters more difficult, the range of the memberOf property is type Organization. Organization is affected by the transitive property subOrganizationOf. So it is necessary to query if the subject is a memberOf or worksFor any entity that in turn is a subOrganizationOf http://www.Department0.University0.edu. Transitive closure is not limited to a single level, therefore it is necessary to repeat this loop until the organization does not appear as a suborganization. In point of fact, there is no subOrganizationOf department, so this transitive closure will not introduce any new results, but it still must be attempted.

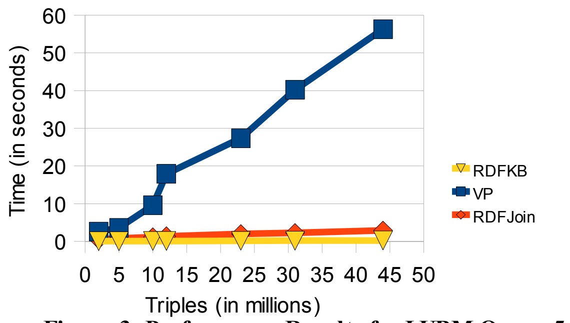


Figure 3: Performance Results for LUBM Query 5

5.3 LUBM Query 6

(type Student ?X)

This query requires the subClassOf inference rule as well as the intersectionOf rule. There are 4 different classes that should be included in this query. So, without inference, 4 subqueries and three unions are required. While this is the simplest query evaluated here, it requires a selection based on object and has low selectivity. Vertical partitioning is sorted on subject rather than object, and therefore pays a performance penalty. Figure 4 shows the performance results for Query 6.

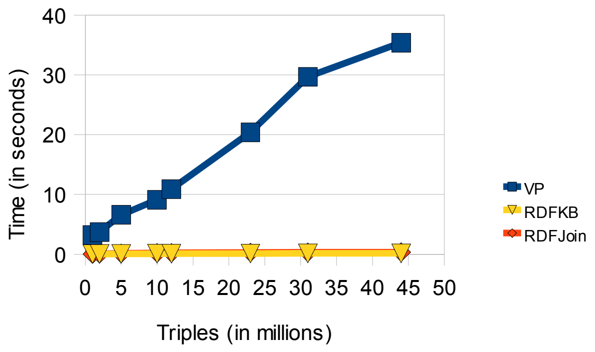


Figure 4: Performance Results for LUBM Query 6

5.4 LUBM Query 11

(type ResearchGroup ?x)
(subOrganizationOf ?x http://www.University0.edu)

This query requires the TransitiveProperty infeorence rule. Research groups are not defined as a subOrganizationOf universities in the dataset; this requires transitive closure. This query is highly selective, and does not involve a class hierarchy, which improves the efficiency of the non-inference solutions. However, unlike in Query 5, the transitive closure does add data to the result set as the research groups are defined as subOrganizations of departments, rather than the university. Transitive closure requires a subject-object join, which in the vertical partitioning solution requires a nested loop. Figure 5 shows the performance results for Query 11.

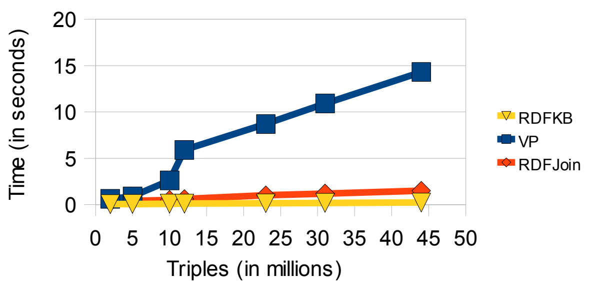


Figure 5: Performance Results for LUBM Query 11

5.5 LUBM Query 13

(type Person ?X)
(hasAlumnus http://www.University0.edu ?X)

This query requires four types of inference: subClassOf, intersectionOf, inverseOf and subPropertyOf. hasAlumnus is not defined in the dataset, but the ontology defines it as the inverseOf degreeFrom. Even degreeFrom is not defined in the dataset, but the ontology defines subproperties of degreeFrom that are defined in the dataset. As already stated, Person includes 21 different class specifications. To query the property hasAlumnus requires querying 5 separate properties. All of these subqueries then have to be unioned together to recreate this inference during query processing. Only hasAlumnus, which actually never appears in the dataset, involves a subject-object join, and this join will never actually be executed as hasAlumnus returns no results. So this is a subject-subject join, and all of the unions are subject-subject merge joins. Figure 6 shows the performance results for Query 13.

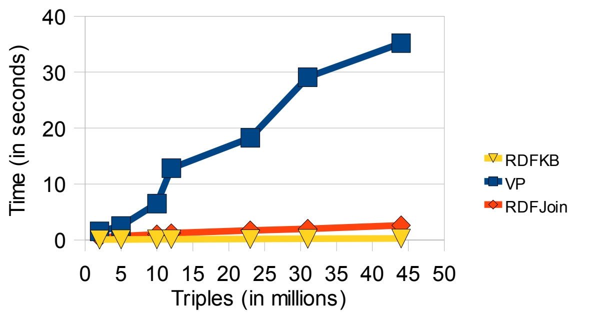


Figure 6: Performance Results for LUBM Query 13

5.6 TradeOffs

RDFKB stores all inferred data rather than performing inference at query time. Thus, inference is calculated at storage time, more triples are persisted, and more triples are loaded into memory. The trade-offs of this approach are added storage time, increased storage space requirements, and increased memory consumption. The number of stored triples is increased by 46.2% for our LUBM dataset. We have asserted that the architecture of RDFJoin minimizes the costs of this trade-off. To validate this, we designed and executed tests to quantify the costs.

To perform all of the queries and tests outlined in Section 5, involving 44 million triples, our maximum memory usage was 3.713 GB. This memory consumption is only 3.7% higher than RDFJoin without inference.

Obviously, the size of the database is increased 100% by storing a second copy of the tables without the inferred triples. There is no other significant increase in the database size (<0.2%) resulting from adding the inferred triples. There is a reduction in the achievable compression ratio. In our experiments with the LUBM dataset, this reduction varied from 6.3% to 9.8%.

RDFKB does support adding triples to the dataset, and inference is calculated at the time the triples are added. This increases the amount of time required to add and store triples. Figure 7 shows the time to add triples for RDFJoin, for RDFKB and for just inference. Adding a triple to RDFKB includes adding it to the RDFJoin tables. Therefore, the time added as a result of inference is the time difference between between RDFJoin and RDFKB. This is plotted on the graph and labeled "Inference Only", i.e. the actual cost of inferring the triples and storing them. This cost is 12% of the overall time cost of the additions for 10 million triples. We assert that a 12% increase in the time to add triples to the dataset is well worth it to achieve the added functionality of inference and a significant improvement in query performance.

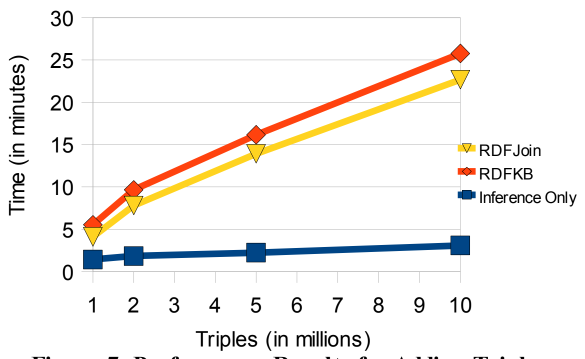


Figure 7: Performance Results for Adding Triples

5.7 Performance Summary

Table 1 provides a summary performance comparison of RDFKB, RDFJoin and vertical partitioning. In every query, RDFKB consistently achieves the best performance, and the performance gain increases as the dataset grows, demonstrating scalability.

Furthermore, RDFKB requires no special coding or understanding of the ontology to develop these queries. No unions are required to implement any of the queries within the RDFKB system.. Even if the inference logic for the other solutions is discovered from the ontology and performed automatically, the subqueries and unions would still be required as the inferred knowledge is not part of the dataset. Thus, hard-coding the query specifics yields the fastest solution vertical partitioning or RDFJoin can provide.

RDFKB consistently outperforms both vertical partitioning and RDFJoin and eliminates the need for complex queries or understanding of the ontology by the developer.

Table 1: Performance Improvements

Query #	% query time reduction vs vertical partitioning	% query time reduction vs RDFJoin
3	92.9%	67.4%
5	99.6%	92.0%
6	99.6%	60.0%
11	98.4%	84.7%
13	99.3%	91.2%

6. RELATED WORK

Jena [14], a semantic web framework for Java, supports a framework for registering and executing inference rules that is similar to RDFKB. Jena and RDFKB both define a Java abstract class (Jena calls this Reasoner) that specifies the interface to the inference rule implementation, factory methods to instantiate instances of this class, and a registration system. Jena's getRawModel() API provides similar support to the _Without_Inference tables in RDFKB, and Jena allows Reasoners to add additional "virtual" triples to the dataset. Jena even provides an OWL reasoner that implements most of the same inference rules implemented in our experiments. However, there are several important differences between Jena's inference support and that of RDFKB. Inferred data is not persisted with Jena; inference is performed at query time, or precomputed using the prepare() method. Furthermore, while Jena does support persistence of triples to relational databases, it does so only as an alternative storage method and requires a fixed schema. Thus, adding millions of inferred triples will incur a performance penalty because a schema such as RDFJoin can not be used to store the data. Finally, RDFKB supports growing the dataset. If triples are added in Jena, this will cause all the deduced triples to be discarded, and inference will be reprocessed across the entire dataset.

In *An Approach to RDF(S) Query, Manipulation and Inference on Databases*[15], Lu et al. propose a solution to RDF inference that is based on top of relational databases. In this research, they propose to infer knowledge at storage time and store all inferred triples. In this way, the strategy is very similar to RDFKB. They also reach similar conclusions concerning the superiority of this approach to inference during query processing. However, this database's schema incurs a performance penalty for the storage of inferred triples. Because RDFKB utilizes column stores and RDFJoin, it is able to achieve greater performance. Our experiments show that our solution is highly scalable, and performs efficiently with very large datasets. In [15], the query time steadily increases as the dataset grows. The reported experiments involve much smaller datasets than our testcases, and the reported query performance is several orders of magnitude slower the RDFKB's performance results.

There is significant research involving efficient database schemata for RDF datasets. In *Column Stores for Wide and Sparse Data*[3], Abadi proposes using column store databases, and, in *Scalable Semantic Web Data Management Using Vertical Partitioning*[4], Abadi et al. propose partitioning the dataset into two column tables for each predicate. In *Hexastore: Sextuple Indexing for Semantic Web Data Management*[6], Weiss et al. propose a main memory schema using sextuple indexing. In

RDF-3X: a RISC-style engine for RDF[17], Neumann et al. propose a RDF indexing system with an efficient query processor. In all of these solutions, specialized schemata and indexes are used to improve query performance. However, none of these solutions provide a solution for inference queries. RDFJoin and RDFKB leverage concepts from these designs, including subject-subject merge joins and sextuple indexing. However, unlike these solutions, RDFJoin can add triples without a performance penalty except when the vocabulary is increased. RDFKB utilizes this technology to provide a high performance inference solution.

7. FUTURE WORK AND CONCLUSIONS

One area for future work is to develop inference rules based on logic outside of OWL ontology files. There are specialized ontology tools and description logic systems that provide rules and axioms. There are also tools such as WordNet[12] which can be used for textual inference. The total of all unique noun, verb, adjective and adverb strings in WordNet 3.0 is 147,278[12]. This is much smaller than the number of unique URIs in our LUBM test dataset, less than 0.3% of the size actually. Therefore, even taking into account that many of these strings have multiple meanings that must be accounted for, it is very reasonable to claim that we can store every combination of triples of words in RDFKB and execute and store textual inference using WordNet.

Inference involving probabilities is also left as future work. OWL defines inference rules that are absolute, so reasoning with uncertainty has not been addressed. Ontology mapping and handling multiple RDF schemata is also a problem for future work. Generally, ontology mapping involves probabilistic reasoning as well since there is uncertainty involved in mapping automation[18][19][20]. One potential solution is to store the probability of the triple being in the dataset, rather than simply a 1 or 0 in the bit vector. In this case, the bit vectors would become vectors of fractions, where each fraction represents a probability. The future work would be to utilize RDFKB to store, retrieve and manage this information in conjunction with the RDF dataset.

We have proposed a solution for adding inference to RDF datasets stored in relational databases. This solution is efficient and scalable. RDFKB outperforms existing solutions, and the performance improvement increases as the dataset increases, which demonstrates scalability. Query processing does not require any knowledge of inference rules to access inferred data. Queries become simpler and more efficient.

The definition of knowledge base requires a knowledge base to provide for "computerized collection, organization and retrieval of knowledge."[9] Our solution, RDFKB, uses inference rules to acquire and complete knowledge in the database. RDFKB organizes and persistently stores the inferred data, allowing simple and efficient query retrieval of the knowledge. Therefore, we assert that RDFKB is a highly functional and efficient knowledge base for managing RDF information.

8. REFERENCES

[1] World Wide Web Consortium (W3C). http://www.w3c.org.

[2] Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm.

[3] Abadi, D.J. Column Stores for Wide and Sparse Data. In Proceedings of CIDR. 2007, 292-297.

[4] Abadi, D.J., Marcus, A., Madden, S., and Hollenbach, K.J. Scalable Semantic Web Data Management Using Vertical Partitioning. In Proceedings of VLDB. 2007, 411-422.

[5] Chong, E.I., Das, S., Eadon, G., and Srinivasan, J. An Efficient SQL-based RDF Querying Scheme. In Proceedings of VLDB. 2005, 1216-1227.

[6] Weiss, C, Karras, P, and Bernstein, A. Hexastore: Sextuple Indexing for Semantic Web Data Management. In Proceeding of VLDB. 2008.

[7] Chen, H., Wu, Z., Wang, H., and Mao, Y. RDF/RDFS-based Relational Database Integration. In Proceedings of ICDE. 2006, 94-94.

[8] McGlothlin, J., Khan, L. RDFJoin: A Scalable Data Model for Persistence and Efficient Querying of RDF Datasets. Technical Report UTDCS-08-09**.** http://www.utdallas.edu/~jpm083000/rdfjoin.pdf.

[9] http://en.wikipedia.org/wiki/Knowledge_base

[10] Web Ontology Language. http://www.w3.org/2004/OWL.

[11] Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts.

[12] WordNet: A Lexical Database for the English Language. http://wordnet.princeton.edu.

[13] Broekstra, J., Klein, M.C.A., Decker, S., Fensel, D., Harmelen, F.V., and Horrocks, I. Enabling knowledge representation on the Web by extending RDF schema. In Proceedings of WWW. 2001, 467-478.

[14] Jena- A Semantic Web Framework for Java. http://jena.sourceforge.net/.

[15] Lu, J., Yu, Y., Tu, K., Lin, C., and Zhang, L. An Approach to RDF(S) Query, Manipulation and Inference on Databases. In Proceedings of WAIM. 2005, 172-183.

[16] Neumann, T. and Weikum, G. RDF-3X: a RISC-style engine for RDF. In Proceedings of PVLDB. 2008, 647-659.

[17] Sidirourgos, L., Goncalves, R., Kerstten, M., Nes, N. and Manegold. S. Column-Store Support for RDF Management: not all swans are white. In Proceedings of VLDB. 2008.

[18] Cali, A., Lukasiewicz, T., Predoiu, L., and Stuckenschmidt, H. Tightly Integrated Probabilistic Description Logic Programs for Representing Ontology Mappings. In Proceedings of FoIKS. 2008, 178-198.

[19] Cali, A. and Lukasiewicz, T. Tightly Integrated Probabilistic Description Logic Programs for the Semantic Web. In Proceedings of ICLP. 2007, 428-429.

[20] Curino, C., Quintarelli, E., and Tanca, L. Ontology-Based Information Tailoring. In Proceedings of ICDE Workshops. 2006, 5-5.

R2D: Extracting Relational Structure from RDF Stores

Sunitha Ramanujam[1], Anubha Gupta[1], Latifur Khan[1], Steven Seida[2], Bhavani Thuraisingham[1]
[1] *The University of Texas at Dallas*
{sxr063200, axg089100, lkhan, bxt043000}@utdallas.edu
[2] *Raytheon Corporation*
steven_b_seida@raytheon.com

Abstract **– The enthusiastic acceptance of Resource Description Framework (RDF) as a data model has given birth to a new data storage paradigm, namely, the RDF Graph model. The pool of modeling and visualization tools available for RDF stores is limited due to the technology being in its fledgling stage. The work presented in this paper, called R2D (RDF-to-Database) is an effort to make available, to RDF data stores, the abundance of relational tools that are currently in the market. This is done in the form of a JDBC wrapper around RDF Stores that presents a relational view of the stores and their data to the modeling and visualization tools. This paper presents key R2D functionalities and mapping constructs, procedures for every stage of R2D deployment, and sample results in the form of screenshots and performance graphs.**

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

I. INTRODUCTION

In recent years, the explosion of the Internet has resulted in the emergence of an evolutionary stage of the World Wide Web, namely, the Semantic Web. To realize the Semantic Web vision various standards, such as Resource Description Framework [1], are being developed to enable users to access information more efficiently and accurately. The simplicity and flexibility offered by RDF data models have resulted in an increase in the number of data stores that use the RDF Graph model.

Such a plethora of RDF information stores have, consequently, given rise to the need for tools to manage and visualize this data, However, most of the currently available data modeling, visualization, and management tools are still based on the more mature models such as relational and tabular models [2]. In order to continue to leverage the advantages offered by relational tools without losing out on the benefits offered by newer web technologies, the gap between the two needs to be bridged.

One method to bridge this gap is to create an equivalent relational schema in an existing Relational Database Management System and copy the RDF triples data into the corresponding tables in the relational schema. This approach leads to space wastage due to duplication of the data in the RDF store. Further, synchronization of the data in the two stores is another issue to be considered, and some sort of resource and time intensive mechanism would have to be in place to ensure that the relational store is a true and current version of the RDF store.

We propose a solution to the bridging problem without the need to create an actual physical relational schema and duplicate data. The work presented in this paper, called R2D (RDF-to-Database), 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. Our research provides a relational interface to data stored in the form of RDF triples and, to the best of our knowledge, has no comparable counterparts. Our contributions are:

- We propose a mapping scheme for the translation of RDF Graph structures to an equivalent relational schema
- The proposed mapping process includes the ability to map, through extensive examination of instance data, even "sloppy" RDF Graphs that either do not have any, or have incomplete structural/schema information included along with the data.
- Based on the RDF-to-RDBMS map file created, we propose a transformation process that presents a normalized, non-generic, domain-specific, virtual relational schema view of the given RDF store
- We propose a mechanism to transform any relational SQL queries issued against the virtual relational schema into its SPARQL equivalent, and return the triples data to the end-user in a relational/tabular format
- We provide all of the above in the form of a JDBC interface that can be plugged into existing visualization tools seamlessly.

Section II provides an overview of current research in the relational-to-rdf arena. Section III discusses R2D's modus operandi and mapping constructs. The algorithms involved in the mapping process are described in Section IV followed by experimental results in Section V. The paper concludes with a discussion on the advantages of this research in Section VI.

II. RELATED WORK

While the overall concept of R2D is unique and has no comparable counterparts, several research efforts exist that attempt to bring relational database concepts and semantic web concepts together, albeit from a perspective that is opposite to that considered in our work. Some of these efforts include D2RQ [3] and Virtuoso RDF Views [4] which are essentially mapping efforts between relational schema and OWL/RDFS concepts where a relational database schema is taken as input and an RDF interface of the same is presented as output. Triplify [5] is another effort at publishing linked data from relational databases and it achieves this by extending SQL and using the extended version as a mapping language. RDF123 [6], an open source translation tool, also uses a mapping concept, however its domain is spreadsheet data and it attempts to achieve spreadsheet-to-RDF translation by allowing users to define mappings between spreadsheet semantics and RDF graphs.

The Hybrid model [7] is the nearest match to the mapping methodology in our work, however, since the model generates a table for every property in the ontology, it results in unnecessary tables in the case of 1:N relationships between subject and object resources. R2D avoids this by adding a foreign key column to the appropriate table when processing 1:N relationships. The hybrid model also fails on RDF graphs which do not include schema information while R2D is able to glean structural information even in the absence of ontological constructs.

As can be seen from the above discussions, none of the existing research efforts address the issue of enabling relational applications to access RDF data without data replication. Thus, to the best of our knowledge, R2D is the first endeavor to address this issue.

III. R2D PRELIMINARIES

The architecture of the proposed system and the deployment sequence of the algorithms comprising R2D are illustrated in Figure 1. R2D's functionality is made available as a JDBC Interface that can be plugged into any visualization tool that is based on a relational data model.

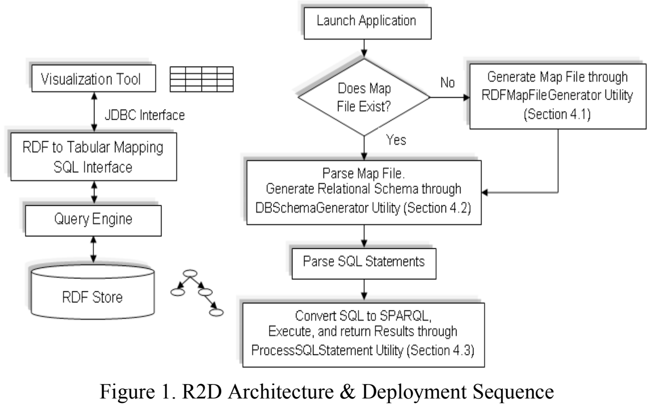

Visualization Tool
JDBC Interface
RDF to Tabular Mapping SQL Interface
Query Engine
RDF Store
Launch Application
Does Map File Exist?
No
Yes
Generate Map File through RDFMapFileGenerator Utility (Section 4.1)
Parse Map File. Generate Relational Schema through DBSchemaGenerator Utility (Section 4.2)
Parse SQL Statements
Convert SQL to SPARQL, Execute, and return Results through ProcessSQLStatement Utility (Section 4.3)

Figure 1. R2D Architecture & Deployment Sequence

Table 1 tabulates the notional mapping between OWL/RDFS Ontology terminologies and relational concepts that is adopted by R2D.

Table 1: Notional Mapping between RDFS/OWL and R2D

OWL/RDFS	RELATIONAL CONCEPT
TERMS	
rdfs:class	Table
rdf:property	Column
rdfs:domain	Table that the rdf:property is a column of
rdfs:range	Datatype of the column
rdf:type	Values of the Primary Key column of the table

At the heart of the relational transformation of RDF Graphs is the R2D mapping language –a declarative language that expresses the mappings between RDF Graph constructs and relational database schema constructs. In order to better explain the R2D mapping language constructs, examples from the sample scenario in Figure 2 are included where applicable.

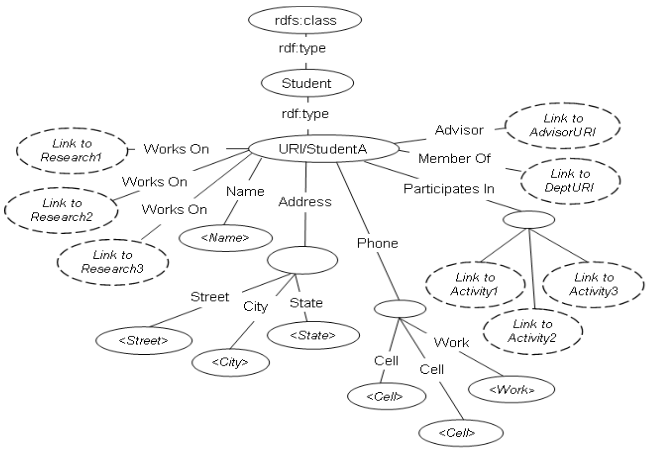


Figure 2. Sample Scenario

The constructs of the current version of the mapping language are presented below.

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

The mapping constructs specific to an r2d:TableMap are as follows.

r2d:keyField: The r2d:keyField construct specifies the primary key attribute for the r2d:TableMap to which the field is attached. The data value associated with the field specified by r2d:keyField is the object of the "rdf:type" predicate belonging to the rdfs:class subject corresponding to its r2d:TableMap. ***Example:*** *An r2d:keyField (primary key) called "Student_PK" field is attached to the "Student" TableMap and one of its values, corresponding to the sample scenario in Figure 2, is "URI/StudentA".*

r2d:ColumnBridge: r2d:ColumnBridges relate single-valued RDF Graph predicates/properties 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 "Name" and "Member Of" predicates in Figure 2 become r2d:ColumnBridges belonging to the "Student" r2d:TableMap*

r2d:MultiValuedColumnBridge(MVCB): Those RDF Graph predicates that have multiple object values for the same subject are mapped using the MVCB construct. MVCBs typically correspond to RDF constructs such as RDF containers and collections and are used to indicate N:1 and N:M relationships between the virtual relational schema tables.

Example: *The "Works On" predicate in Figure 2 is an example of an MVCB mapping.*

r2d:SingleValuedBlankNode (SVBN): This construct helps relate blank nodes with distinct predicates to relational database columns. In the virtual relational schema, the blank node is ignored and the predicates of blank nodes are treated as having simple 1:1 relationships to the subject of the blank node.

Example: *The object of the "Address" predicate in Figure 2 is an example of an SVBN that has the distinct predicates of "Street", "City", and "State".*

r2d:MultiValuedBlankNode (MVBN): This construct refers to blank nodes in the RDF Graph that contain repeating predicates. These blank nodes have multiple object values for the same subject and predicate concept associated with the blank node. An MVBN typically results in the generation of a separate r2d:TableMap with a foreign key relationship.

Example: *The object of the "Phone" predicate in Figure 2 is an example of an MVBN that has multiple object (Cell) values for the subject (URI/StudentA) and predicate (Cell) concept associated with the MVBN.*

The mapping constructs specific to column bridges and blank nodes are described below.

r2d:belongsToTableMap(BTTM): This construct connects a r2d:ColumnBridge or MVCB to an r2d:TableMap. Every r2d:ColumnBridge must specify a value for either this construct or the r2d:belongsToBlankNode construct. ***Example:*** *The "Name" predicate in Figure 2 is associated with the resource "URI/StudentA", an instance of the "Student" r2d:TableMap. Hence, the BTTM construct corresponding to "Name" r2d:ColumnBridge is set to a value of "Student", thereby connecting the ColumnBridge to a table.*

r2d:belongsToBlankNode (BTBN): This construct ties a r2d:ColumnBridge or MVCB to an SVBN or an MVBN. ***Example:*** *The "Street" r2d:ColumnBridge corresponding to the "Street" predicate in Figure 2 is associated with the "Address" SVBN. Hence, for the "Street" r2d:ColumnBridge the BTBN construct is used to associate it to the "Address" blank node.*

r2d:refersToTableMap (RTTM): This construct is optional for column bridges and is only used for those triples that contain a resource object for a predicate. This construct is used to generate primary key-foreign key relationships within the virtual relational schema.

Example: *The object of the "Member Of" predicate in Figure 2 is a resource that translates to another r2d:TableMap called "Department". Hence the "MemberOf" r2d:ColumnBridge includes the RTTM construct with a value of "Department".*

r2d:predicate: The r2d:predicate construct is used to store the fully qualified property name of the predicate which corresponds to the column bridge. 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

r2d:MultiValuedPredicate (MVP): This construct is used when there are multiple predicate names that refer to the same overall object type despite each individual object having a different value. r2d:MultiValuedPredicates are also used to keep track of the predicates associated with RDF containers and RDF collections.

r2d:datatype: This construct specifies the datatype of its column bridge and is derived from the rdfs:range predicate or, in its absence, by examination of the object values of the predicate.

The virtual relational schema generated by R2D for the scenario in Figure 2 is as illustrated in Figure 3. Section IV (B) explains how this schema is arrived at.

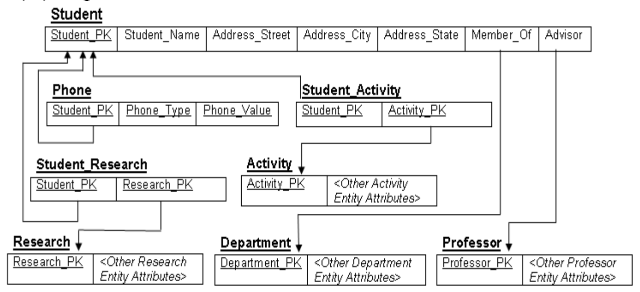


Figure 3. Equivalent Relational Schema for Scenario in Figure 2

IV. R2D: A PROTOTYPE DESIGN

In addition to the design of the RDF-to-Relational mapping language discussed in the previous section, the objectives of this research are to develop algorithms that enable the relationalization of RDF stores. These algorithms comprise the R2D framework and are discussed in the following subsections.

A. *RDFMapFileGenerator*

The RDFMapFileGenerator algorithm automatically generates an RDF-to-Relational mapping file. It takes as input the RDF Store that is to be transformed and produces the transformation mapping file as output. The RDFMapFileGenerator algorithm works on RDF Stores with or without structural/schema information.

When structural information about the triples database is present the RDFMapFileGenerator algorithm discovers the schema definitions and creates appropriate Table and Column mappings based on the schema information. Predicates belonging to instances with structural information are processed and added to the r2d:tableMap corresponding to the "rdfs:class" of the instance using the constructs defined in Section III.

Instances without structural information are handled by creating a potential TableMap for each such instance. For every simple predicate of such resources, a new column is added to the resource's TableMap if a column corresponding to a predicate does not already exist in the TableMap. Additionally, the nature of the relationship that exists between all predicates (both pre-defined and undefined) and the subject is also determined. If the subject contains multiple object values for the same predicate then column type of the corresponding column is set to MVCB. Otherwise, the column type is set to r2d:ColumnBridge. This determination is mandatory in order to arrive at a normalized and logically sound relational schema.

Furthermore, cardinality estimation is performed during the processing of predicates for those predicates that link subjects to objects that are resources and not literals. Whenever 1:N or N:M relationships are identified the corresponding predicate is mapped using the MVCB or MVBN construct, whichever is applicable. Once all predicates are processed, the potential TableMap is compared with other existing TableMaps; if an identical TableMap exists, the potential TableMap is discarded, otherwise it is added to the list of TableMaps.

B. DBSchemaGenerator

The DBSchemaGenerator module is the next stage in the R2D process. This algorithm takes the RDF-to-Relational Schema mapping file generated in Section III (A) and returns a virtual, appropriately normalized relational database schema consisting of entities/tables and the relationships between them. A high-level description of the algorithmic details follows.

For every entry of type r2d:TableMap in the map file one relational table is added to the virtual relational schema. For the sample scenario in Figure 2, a virtual table called *Student* is created corresponding to the *Student* r2d:TableMap. The more complex structures such as r2d:SingleValuedBlankNodes (SVBNs), r2d:MultiValuedBlankNodes (MVBNs), and r2d:MultiValuedColumnBridges (MVCBs) are handled as follows. For SVBNs, the predicates belonging to the blank node are associated with the table corresponding to the subject of the blank node object. Thus, the *Street*, *City*, and *State* predicates of the *Address* SVBN in Figure 2 are added as columns to the *Student* table.

When an MVBN or MVCB with literal objects is encountered (this is equivalent to a multi-valued attribute in relational database terminology) a new table is added to the virtual relational schema and the primary key fields of the table associated with the r2d:belongsToTableMap construct specified for the MVBN or MVCB are added as fields to this new table.

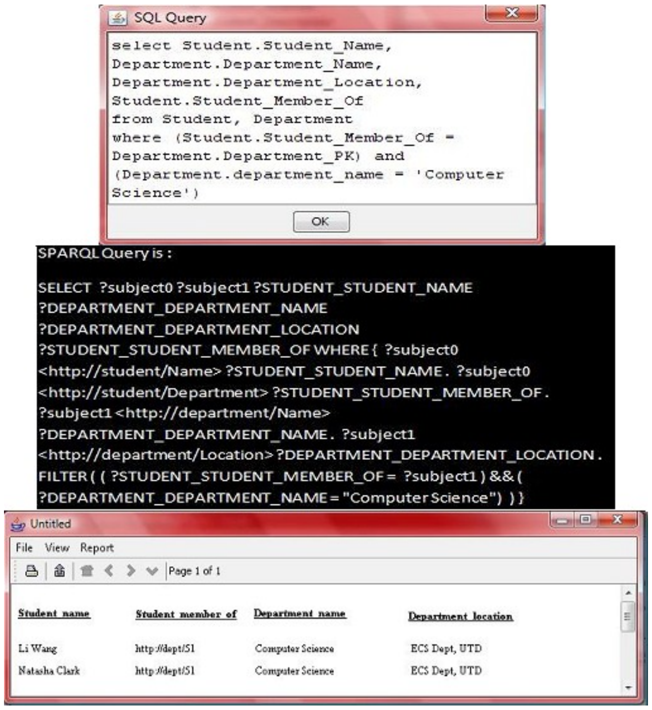
The object of the "Phone" predicate in Figure 2 is an example of an MVBN. The relational transformation for *Phone* involves the generation of an r2d:TableMap of the same name. This *Phone* r2d:TableMap includes as columns a *Type* field that holds the values of the multi-valued predicates off of the MVBN (in our sample scenario, the *Type* field will hold the values "*Cell"* and "*Work"*), and a *Value* field that holds the object values of the predicates off of the MVBN. Additionally, the r2d:TableMap also includes, as foreign key, the *Student_PK* column which references the primary key of the *Student* r2d:TableMap.

In RDF graphs where the MVBN or MVCB has objects that are resources themselves (as indicated by the r2d:refersToTableMap construct specified for the MVBN or MVCB), the type of relationship that exists between the subject and the object of the MVBN/MVCB is assessed. If an N:M relationship exists between the {subject, object} pair, a join table is added to the virtual relational table list and the primary key fields of both the tables (corresponding to the subject and the object) are added to this join table. The *Works On* predicate in Figure 2 is an example of one such MVCB whose relational transformation results in the generation of a new r2d:TableMap of the same name. This new TableMap represents the N:M relationship between *Student* and *Research* and has the primary keys of both the tables included as fields. If the {subject, object} pair shares a 1:N or N:1 relationship, the primary key of the referred table is added to the attribute list of the referring table.

Finally, entries of type r2d:ColumnBridge in the map file are processed by adding the column bridge as an attribute to the table or blank node referred to in the r2d:belongsToTableMap or r2d:belongsToBlankNode construct specified for the column bridge.

C. ProcessSQLStatement

The final stage in the R2D process is the SQL-to-SPARQL translation where SQL statements issued against the virtual relational schema are parsed, translated into equivalent SPARQL queries that are executed against the RDF Store, and the results are returned in relational format. The algorithm for this stage is called ProcessSQLStatement. Due to space constraints only a brief description of this algorithm is provided here. Very broadly, for every field in the original SQL select list, a variable is added to the SPARQL SELECT list. Next, the predicates of every non-primary-key field in the SPARQL SELECT list are retrieved and added to the SPARQL WHERE clause to bind the SELECT list variables. SQL WHERE clauses of the types (field <operator> <value/field2>) are typically included in the FILTER clause which is then added to the SPARQL Query. The transformed SPARQL Query is executed, and the retrieved data is returned to the visualization tool in relational format seamlessly. Figure 4 shows a sample SQL query and its SPARQL equivalent and tabular results as generated by ProcessSQLStatement.



```
select Student.Student_Name,
Department.Department_Name,
Department.Department_Location,
Student.Student_Member_Of
from Student, Department
where (Student.Student_Member_Of =
Department.Department_PK) and
(Department.department_name = 'Computer
Science')
```

```
SPARQL Query is :

SELECT ?subject0 ?subject1 ?STUDENT_STUDENT_NAME
?DEPARTMENT_DEPARTMENT_NAME
?DEPARTMENT_DEPARTMENT_LOCATION
?STUDENT_STUDENT_MEMBER_OF WHERE { ?subject0
<http://student/Name> ?STUDENT_STUDENT_NAME . ?subject0
<http://student/Department> ?STUDENT_STUDENT_MEMBER_OF .
?subject1 <http://department/Name>
?DEPARTMENT_DEPARTMENT_NAME . ?subject1
<http://department/Location> ?DEPARTMENT_DEPARTMENT_LOCATION .
FILTER ( ( ?STUDENT_STUDENT_MEMBER_OF = ?subject1 ) && (
?DEPARTMENT_DEPARTMENT_NAME = "Computer Science") ) }
```

Student name	Student member of	Department name	Department location
Li Wang	http://dept/S1	Computer Science	ECS Dept, UTD
Natasha Clark	http://dept/S1	Computer Science	ECS Dept, UTD

Figure 4: SQL-to-SPARQL Transformation

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 tools used include Jena 2.5.6[1] to manipulate the RDF triples, MySQL 5.0 to house the relational equivalent of the given RDF store, Java 1.5 for development of the algorithms detailed in Section IV, and DataVision 1.2.0[2] to visualize/generate reports based on RDF data. The performance experiments conducted and the reporting tool outputs presented below are based on the IngentaConnect's publication domain[3] that includes information about journals, issues, and articles. Synthetic RDF triples data stores of various sizes were created based on IngentaConnect's schema in Jena for performance evaluation exercises.

The relational equivalent of the RDF data set was generated using the RDFMapFileGenerator and DBSchemaGenerator Algorithms detailed in Section IV. An 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. Figure 5 displays the time taken by the map file generation process for RDF stores of various sizes and the database schema as seen by DataVision.

The map file generation process is especially time-intensive for large databases without structural information (which is the case with our experimental data set) since RDFMapFileGenerator has to explore every resource to ensure that no property is left unprocessed. Sampling methods can be used to improve performance, but at the risk of reduction in accuracy. Also, if a domain expert is available, this step can be bypassed completely by providing a map file manually.

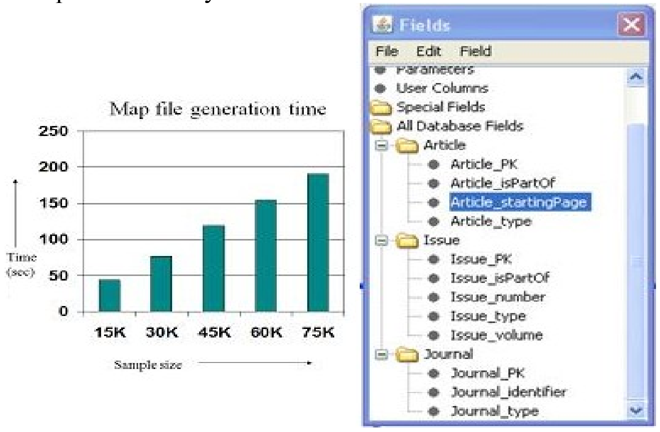
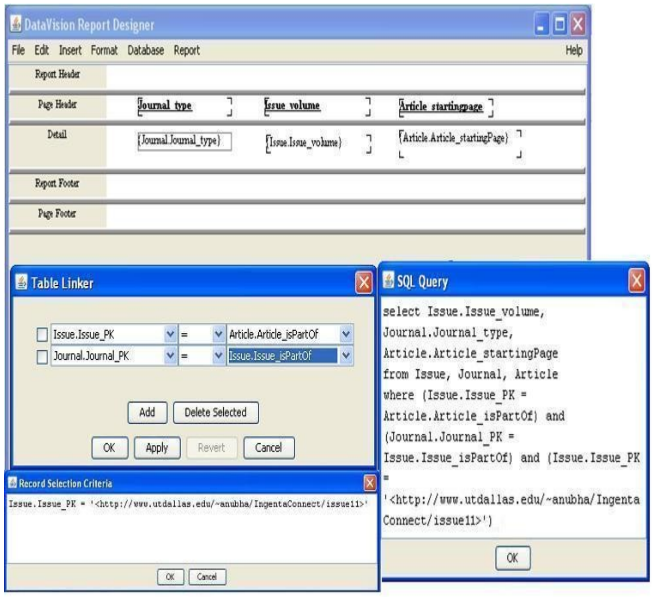


Figure 5: Response Time for RDFMapFileGenerator

Figure 6 illustrates DataVision's query building process. Based on the fields chosen (in the "Report Designer" window), the table linkages (i.e., joins, illustrated in the "Table Linker" inset) specified, and additional record selection criteria specified (illustrated in the "Record Selection Criteria" inset), DataVision generates an appropriate SQL query, as shown in the "SQL Query" inset, to extract the required data. At this juncture, the Statement, PreparedStatement, and ResultSet JDBC Interfaces are invoked which trigger ProcessSQLStatement and return the results to DataVision in the expected tabular format.



Issue.Issue_PK	=	Article.Article_isPartOf
Journal.Journal_PK	=	Issue.Issue_isPartOf

```
Issue.Issue_PK = '<http://www.utdallas.edu/~anubha/IngentaConnect/issue11>'
```

```
select Issue.Issue_volume,
Journal.Journal_type,
Article.Article_startingPage
from Issue, Journal, Article
where (Issue.Issue_PK =
Article.Article_isPartOf) and
(Journal.Journal_PK =
Issue.Issue_isPartOf) and (Issue.Issue_PK
=
'<http://www.utdallas.edu/~anubha/Ingenta
Connect/issue11>')
```

Figure 6. DataVision Query Processing

[1] http://jena.sourceforge.net/index.html
[2] http://datavision.sourceforge.net
[3] www.ingentaconnect.com

To compare the performance of queries executed through the virtual relational schema offered by R2D against the query performance from an equivalent RDBMS, a physical relational schema corresponding to the publications data was created in MySQL and populated with data similar to the triples data in the Journal-Issue-Article RDF data store in Jena. Four queries were run against Jena RDF stores and MySQL relational databases of various sizes and the response times are displayed in Figure 7.

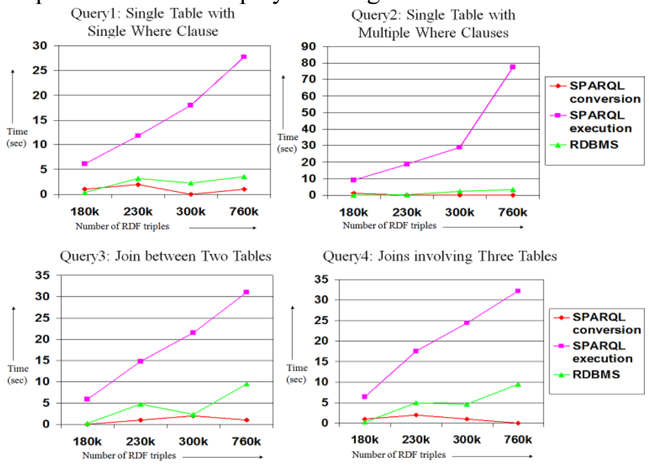


Figure 7. Response times for Queries

The time taken for the SQL-to-SPARQL conversion (ProcessSQLStatement Algorithm) is negligible and nearly constant. Hence, R2D does not add any overheads to the SPARQL query performance. The fact that the relational (SQL) queries exhibit superior performance than their SPARQL equivalents is not surprising since refined performance optimization options have been at the disposal of relational databases for many decades now. Further, for each row of the RDBMS with 'n' columns, there are 'n' triple tuples in the corresponding RDF Store. Thus, for the datasets considered, the RDBMS equivalent of the RDF Stores had approximately two-thirds less data than the RDF Stores which was another contributor to better RDBMS response times than the RDF data store.

However, this improved performance comes at the expense of additional disk space due to duplication of data into the RDBMS, and additional system resources/human effort required to ensure that the duplicated data is kept synchronized with the original RDF store. On the other hand, for a small price in terms of response time, R2D offers an avenue for users to continue to take advantage of the vast assortment of visualization tools that are readily available without having to duplicate/synchronize RDF data.

VI. DISCUSSION

The R2D framework in this paper is an attempt at integrating relational concepts with semantic web concepts with the objective of permitting reusability of tools that are based on a relational model. Since current storage methods for RDF stores involve housing the triples in a relational database, some factions may consider R2D to be a "double-wrapping" application that provides a relational wrapper around RDF stores that are, in turn, stored in a relational database. However, almost every storage mechanism involves the creation of a generic, non-application-specific <s,p,o> table that would make the determination of the problem domain addressed by the model difficult without examining the actual data. Further, querying data, using SQL, from such a generic table, to arrive at meaningful information is not a trivial task. It would involve umpteen self-joins on the same table and would require the presence of a domain expert with detailed knowledge of the data. This is because, using these models, it would be impossible for a user to infer the schema and the entities, the attributes, and relationships comprising the same. R2D offers the users the ability to do just this and enables them to actually arrive at a complete domain-specific Entity-Relationship Diagram using the RDF-to-Relational Schema transformation process and fire SQL queries against the same.

Further, R2D, unlike other mapping efforts, can generate an equivalent relational schema even for "sloppy" data (in which ontological constructs/schema definitions are absent) through extensive examination of the data to identify groups of instances that have mostly the same properties associated with them. The degree of accuracy of the generated schema in the absence of structural information may not be as high as when such information is available due to uncertainties regarding similarity of the tables generated in the relational schema. Decisions such as "how similar should two tables be before they are considered to be the same and consolidated" depends, in the absence of structural information, on similarity thresholds set within the algorithm and accuracy varies depending on the thresholds.

REFERENCES

[1] RDF Primer. W3C Recommendation. 2004. http://www.w3.org/TR/rdf-primer/

[2] W.Teswanich, and S.Chittayasothorn, "A Transformation of RDF Documents and Schemas to Relational Databases", In *IEEE PacificRim Conferences on Communications, Computers, and Signal Processing*, 2007, pp. 38-41

[3] C.Bizer, R.Cyganiak, J.Garbers, and O.Maresch, "The D2RQ Platform", 2009, http://www4.wiwiss.fu-berlin.de/bizer/d2rq/

[4] O.Erling, and I.Mikhailov, "RDF Support in the Virtuoso DBMS", In *1st Conference on Social Semantic Web*, 2007, pp. 1617-5468.

[5] S.Auer, S.Dietzold, J.Lehmann, S.Hellmann, and D.Aumueller, "Triplify – Light-Weight Linked Data Publication from Relational Databases", In *18th International World Wide Web Conference,* 2009, pp. 621-630.

[6] L.Han, T.Finin, C.Parr, J.Sachs, and A. Joshi, "RDF123: From Spreadsheets to RDF", *International Semantic Web Conference*, 2008, pp. 451-466.

[7] Z.Pan, and J.Heflin, "DLDB: Extending Relational Databases to Support Semantic Web Queries", In *Practical and Scalable Semantic Systems,* 2003, pp. 109-113.

Semantic Schema Matching Without Shared Instances

Jeffrey Partyka, Latifur Khan, Bhavani Thuraisingham
Department of Computer Science, The University of Texas at Dallas
800 West Campbell Road
Richardson, TX, 75083-0688, United States
{jlp072000, lkhan, Bhavani.thuraisingham
} @utdallas.edu

Abstract

Semantic heterogeneity across data sources remains a widespread and relevant problem requiring innovative solutions. Our approach towards resolving semantic disparities among distinct data sources aligns their constituent tables by first choosing attributes for comparison. We then examine their instances and calculate a similarity value between them known as entropy-based distribution (EBD). One method of calculating EBD applies a state-of-the-art instance matching strategy based on N-grams in the data. However, this method often fails because it relies on shared instance data to determine similarity. This results in an overestimation of semantic similarity between unrelated attributes and an underestimation of semantic similarity between related attributes. Our method resolves this using clustering and a measure known as Normalized Google Distance. The EBD is then calculated among all clusters by treating each as a type. We show the effectiveness of our approach over the traditional N-gram approach across multi-jurisdictional datasets by generating impressive results.

1. Introduction

The problem of information integration has experienced a number of manifestations since its inception, which resulted from the meteoric popularity of relational databases after the 1960's. However, the core of this problem has always been the need to consolidate heterogeneous data sources under a single, unified schema. Over the last few decades, a tremendous amount of effort has been expanded to discover novel information integration strategies.

In this paper we attempt to compare two pairs of data sources by examining the instances of compared tables; the first pair of data sources contains tables describing similar models of transportation network over multiple jurisdictions, while the other pair contains tables detailing varying geographic features. The data sources contain large variations in the geographic areas covered, the number of attributes, and the number of instances.

To measure instance similarity between compared attributes we will attempt to match the respective distributions of their representative types. A type will be defined as a common representation of a group of related pieces of data. Once all types for the compared attributes have been accounted for, the semantic similarity between the attributes is calculated using a measure known as entropy-based distribution (EBD). EBD is based on the ratio of the conditional entropy within the types extracted for a pair of compared attributes with the entropy over all types.

We examine two different instance similarity algorithms. The first examines keywords in the compared attributes and extracts subsequences of their characters known as N-grams. The idea behind this method is that keywords that share more N-grams are more semantically similar to one another. However, this idea often proves to be incorrect in situations where few shared instances exist over multiple jurisdictions. The second approach, which we will dub as the TSim algorithm, executes instance matching by applying a similarity metric known as the Normalized Google Distance (NGD). The end result is a group of distinct clusters (hence types), each of which contains a unique set of keywords related to each other through common semantic features. The similarity between the attributes is then computed by calculating the EBD. Because we do not have to depend on shared N-grams for semantic similarity, our instance matching algorithm can derive a more realistic measure of the implicit semantics existing between any given pair of attributes from distinct data sources.

Our main contributions are as follows. First, we display the inadequacies of the N-gram approach by testing it on multiple datasets and highlighting its inability to identify correct semantic correspondences between attributes due to its reliance on shared instances. Second, we propose a new algorithm, called TSim, that derives semantic similarity between attributes of compared tables without the need for shared instances. This is accomplished through K-medoid clustering of the instance data associated with the attributes into distinct semantic types, with the help of NGD. Finally, we show the effectiveness of our approach relative to the traditional N-gram method through lucid results on two separate datasets.

The rest of this paper is organized as follows. In section 2, we discuss an overview of related work. Section 3 states the problem to be solved and our proposed solution. Section 4 presents in detail the TSim algorithm alongside the current, state-of-the-art approach that depends on shared N-grams. In Section 5 we present results. Finally, in section 6, we outline our future work.

2. Related work

A number of schema matching publications [1,2,3,4,5] tailored to the database community and instance-based ontology matching [9,10] from the ontology matching community, influenced our work. The survey of approaches to automated schema matching by Rahm and Bernstein[1] includes a taxonomy which uses several criteria to categorize the matching approaches such as schema and instance based methods, element-level and structure-level methods, and linguistic and constraint-based methods. Dai, Koudas et al. [2] discuss instance-based schema matching using distributions of N-grams among compared attributes. Bohannon et. al[3] investigate contextual schema matching, in which selection conditions and a framework of matching techniques are used to create higher quality mapping between attributes of compared schemas. Warren and Tompa [4] propose an iterative algorithm that deduces the correct sequence of concatenations of column substrings in order to translate from one database to another without the use of a set of training instances.

Our paper presents an innovative instance matching algorithm that possesses a number of advantages over the N-gram approach proposed by Dai, Koudas et al. First, our new instance matching approach leverages clustering of types for use on distinct keywords found between compared attributes. This approach is better able to capture the semantics of comparisons between attributes because words contain more implicit semantic information than N-grams. Using words, we can reference external data sources that allow for distance metrics to determine word relatedness. In general, this cannot be done with N-grams because they are usually just parts of words. Second, our new instance matching algorithm is flexible enough to allow for different types of semantic distance measures to be used. Treating the semantic distance measure as a pluggable component allows for a wider variety of experiments to be performed on a given instance set, which in turn leads to a better understanding of the kinds of semantic distance measures that best suits a particular type of data. Finally, the use of N-grams for instance similarity between data sources sometimes generates misleading results, especially in cases where data of different languages but similar semantics is being compared.

Since we use Google distance to calculate similarity there is some relevant work. Gligorov et al. [7] apply Google distance [6] to clearly distinguish between pairs of words which are not semantically related and pairs of words that possess a close semantic relation. However, our approach differs from their approach in the following ways. First, Gligorov et al. use Google distance to automatically assign appropriate weights (or importance) to the similarity between concepts associated with a concept hierarchy for the purposes of ontology matching. On the other hand, we use Google distance as a measure to aid in the construction of cohesive clusters containing similar-themed keywords which are then used to perform automated schema matching between individual concepts. Next, Gligorov et al. do not consider instance-based matching; they purely exploits concept labels while our idea of matching is based on the instances associated with the compared concepts.

3. Problem statement and proposal

3.1 Definitions

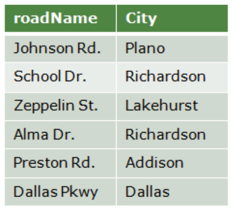
First, we will provide definitions that will assist in defining the problem and describing TSim.

Definition 1 (attribute) *An attribute of a table T, denoted as att(T), is defined as a property of T that further describes it.*

Definition 2 (instance) *An instance x of an attribute att(T) is defined as a data value associated with att(T).*

Definition 3 (type) *A type t associated with attribute att(T) is defined as a class of related entities grouped together.*

In figure 1 below, the two attributes for the given table are roadName and City, and two instances from the roadName attribute are “Johnson Rd.” and “School Dr.”.



roadName	City
Johnson Rd.	Plano
School Dr.	Richardson
Zeppelin St.	Lakehurst
Alma Dr.	Richardson
Preston Rd.	Addison
Dallas Pkwy	Dallas

Figure 1. Sample table containing two attributes and six instances

3.2. Problem statement

Given two data sources, S_1 and S_2, each of which is composed of a set of tables/relations where $\{T_{11}, T_{12}, T_{13}\ldots T_{1M}\}$ S_1 and $\{T_{21}, T_{22}, T_{23}\ldots T_{2N}\}$ S_2, the goal is to determine the semantic similarity between S_1 and S_2. This is done by comparing the respective attribute names and attribute values, or instances, between the tables from S_1 and those from S_2. S_1 and S_2 may be derived from any domain. Additionally, S_1 and S_2 may vary in regards to the number of constituent tables, the number of attributes and instances within a given table.

3.3. Proposed solution

We present two separate instance matching algorithms that generate semantic similarity values between compared attributes in different tables. The first, based on the ideas of mutual information and entropy, extracts features consisting of sequences of characters with length N known as N-grams from the values of the compared attributes [2]. Each N-gram extracted is considered a distinct value type, and the ratios of value types originating from each attribute is determined to be their overall semantic correspondence. While this method can be successful for certain datasets, it can produce incorrect results for others, such as a multi-jurisdiction dataset, where no/few shared instances exist. Section 4 outlines in detail one such situation. The second instance matching algorithm, based on the extraction and clustering of semantically relevant keywords as types, treats distinct keywords extracted from compared attributes, rather than N-grams, as features. Further details describing the algorithm are described in Section 4.3. However, it is our intention to clearly show that the use of TSim on distinct keywords is better able to capture the true semantics that exist between compared attributes contained within tables..

It is assumed that we perform 1:1 comparisons between attributes from distinct tables and data sources. After calculating a semantic similarity value between compared attributes, we will repeat the process for all compared attributes between the tables. Next, a final similarity value between the tables is calculated.

4. Matching algorithm: semantic similarity between two tables

4.1. Instance similarity using N-grams

Instance matching between two concepts involves measuring the similarity between the instance values across all pairs of compared attributes. This is accomplished by extracting instance values from the compared attributes, subsequently extracting a characteristic set of N-grams from these instances, and finally comparing the respective N-grams for each attribute. N may be any number, so during all of our experiments involving N-grams in this paper, the value of N was set equal to 2.

4.1.1. Feature Extraction of N-grams

We extract distinct N-grams from the instances and consider each unique N-gram extracted as a type. A type in this context is defined as 2-gram represented by an identifying string of length 2. As an example, for the string "Locust Grove Dr." that might appear under an attribute named Street for a given concept, some 2-grams that would be extracted are 'Lo', 'oc', 'cu', 'st', 't ', 'ov', 'Dr' and so on. Since each of these 2-grams are different, each one would represent a distinct type.

4.1.2. Measuring attribute similarity

N-gram similarity is based on a comparison between the concepts of entropy and conditional entropy known as Entropy Based Distribution (EBD):

$$EBD = \frac{H(C \mid T)}{H(C)} \qquad (1)$$

In this equation, C and T are random variables where C indicates the union of the attribute types C_1 and C_2 involved in the comparison (C indicates "column", which we will use synonymously with the term “attribute”) and T indicates the type, which in this case is a distinct N-gram. EBD is a normalized value with a range from 0 to 1.

Entropy is defined as the measure of the uncertainty associated with a random variable, whereas conditional entropy is defined as the uncertainty associated with one random variable given the value of a second random variable. Conditional entropy is defined as follows:

$$-\sum_{t \in T} \sum_{c \in C} p(c,t) \log p(c \mid t) \quad (2)$$

Our experiments involve 1:1 comparisons between attributes of compared tables, so the value of C would simply be C_1 U C_2. H(C) represents the entropy of a group of types for a particular column (or attribute) while H(C | T) indicates the conditional entropy of a group of types. For more details regarding the usage of EBD and its mathematical derivation, please see our previous work[8].

4.2. Motivation For TSim

4.2.1. Problems With N-grams as a Measure For Semantic Similarity

N-grams are susceptible to generating misleading results. For example, if an attribute named 'City' associated with a table from S_1 is compared against an attribute named 'ctyName' associated with a table from S_2, the attribute values for both concepts might consist of city names from different parts of the world. 'City' might contain the names of North American cities, all of which use English and other Western languages as their basis language, while 'ctyName', might describe East Asian cities, all of which use languages that are fundamentally different from English or any Western language. Using human intuition, it is obvious that the comparison occurs between two semantically similar attributes. However, because of the tendency for languages to emphasize certain sounds and letters over others, the extracted sets of 2-grams from each attribute would very likely be quite different from one another. For example, some values of 'City' might be "Dallas", "Houston" and "Halifax", while values of 'ctyName' might be "Shanghai", "Beijing" and "Tokyo". Based on these values alone, there is virtually no overlap of N-grams. Because most of the 2-grams belong specifically to one attribute or the other, the calculated EBD value would be low. This would most likely be a problem every time global data needed to be compared for similarity.

4.2.2. Overview of the TSim Algorithm

To overcome the problems of the N-gram approach, we need a method that is free from the syntactic requirements of N-grams and uses the keywords in the data in order to extract relevant semantic differences between compared attributes. This method, known as TSim, extracts distinct keywords from the compared attributes and determines their types by leveraging K-medoid clustering to group together keywords of the same type based on a semantic distance metric known as the Normalized Google Distance (NGD). The EBD is then calculated by comparing all instances of keywords representing each type, where a cluster is considered a distinct type.

4.3. The TSim algorithm

We determine semantic similarity between two separate data sources through K-medoid clustering of the keywords extracted from the compared attributes. The distance metric used in assigning keywords to clusters is known as Normalized Google Distance.

4.3.1. Normalized Google Distance

Before describing the process in detail, NGD must first be formally defined:

$$\mathrm{NGD}(x,y) = \frac{\max\{\log f(x), \log f(y)\} - \log f(x,y)}{\log M - \min\{\log f(x), \log f(y)\}} \quad (3)$$

In this formula, f(x) is the number of Google hits for search term **x**, f(y) is the number of Google hits for search term **y,** f(x,y) is the number of Google hits for the tuple of search terms **xy**, and M is the number of web pages indexed by Google. For more information about NGD, consult the work by Gligorov et al[7].

4.3.2. Clustering the Keywords

Once the keyword list for a given attribute comparison has been created, all related keywords are grouped into distinct clusters. From here, we calculate the conditional entropy of each cluster by using the number of occurrences of each keyword in the cluster, which is subsequently used in the final EBD calculation between the two attributes. The clustering algorithm used is the K-Medoid algorithm, which is described in the next section.

4.3.3. The K-Medoid Algorithm

The K-medoid algorithm begins by first determining the number of clusters, dubbed K. This is based on the size of $L_{keywords}$ for each attribute comparison. Second, exactly one keyword from the list is assigned to each of the K clusters in a process called initial seeding. The keywords assigned to the clusters in this step are known as medoids. Third, we assign each keyword in $L_{keywords}$ that is not a medoid to the cluster to which it is most semantically related, while subsequently determining if any cluster medoids need to be recomputed. To do this, we need to use the pairwise NGD values list between the keyword to be assigned to a cluster and all keywords already assigned to that same cluster. Finally, after all keywords have been assigned to clusters, we determine if the medoid for any cluster needs to be recomputed. This is accomplished by examining each of the keywords in a particular cluster and computing an NGD summation between a single keyword in that cluster and all other words in that cluster. The keyword in that cluster that produces the lowest NGD summation will be assigned as the new medoid for that cluster. If no medoids have changed in any cluster, then the K-medoid algorithm is finished, and control proceeds to the calculation of the EBD between the compared attributes. However, if at least one medoid has changed in a particular cluster, then we begin a new clustering iteration.

5. Experiments

We now present the experiment that we conducted regarding matching between distinct data sources in the GIS domain.

5.1. Experimental Setup

Two separate datasets from the GIS domain were used to evaluate the performance of TSim. The first dataset was created from instance data of the Road and Ferries package of a GIS data model known as GDF (Geographic Data Files). The second dataset details a wider assortment of GIS location features across the United States and their associated data beyond merely transportation networks. Some of the location features in this dataset include flight schools, piers, navigable waterways and Indian lands. For both sets of data, the number of attributes and instances vary widely; for example, in the GIS location dataset, the Flight Schools table has the fewest number of attributes (27) and the Piers table has the most (76). Because data from several different areas of the United States were employed in our experiments, we effectively created a *disjoint, multi-jurisdictional environment.* Table 1 below displays a summary of the relevant information regarding the data involved in our experiments with both datasets.

Table 1. Description of (a) transportation dataset & (b) GIS Location Dataset

Table Name	No. of Attributes	Area(s) Modeled	No. of Instances
Road(S_1), Road(S_2)	18,11	Fort Collins, CO Dallas, TX	9851, 5224
Ferry(S_1), Ferry(S_2)	3,8	Seattle, WA	24,42
Traffic Area (S_1), Traffic Area (S_2)	24,26	Virginia	329, 108
Residential Area (S_1), Address Area (S_2)	15,10	New Jersey, Texas	4263, 2122

Table Name	No. of Attributes	No. of Instances
Flight Schools (S_1), Flight Schools (S_2)	27	4653 and 4653, respectively
Schools (S_1), Schools (S_2)	41	57728 and 56730, respectively
Piers (S_1)	76	6159
Indian Lands (S_1)	52	15852
Ports (S_2)	56	4534
NavWaterways (S_2)	30	6879

Table 2a and 2b. Comparison of EBD values generated by the N-gram method and TSim for correct attribute correspondences. In table 2a (left), the N-gram method underestimates the similarity, and in table 2b (right), N-grams overestimate the similarity

	EBD from TSim	EBD from N-Gram
Flight Schools.BUSINESSNM – Schools.NAME	.691	.117
NavWaterways.Name – Piers.Name	.633	.083
NavWaterways. WTWY– Piers.WTRWY	.981	.095
Ports.COUNTY– Piers.COUNTY	.682	.043
Ports.Port – Piers.port	.735	.437

	EBD from TSim	EBD from N-Gram
Traffic Area.County– Road.City	.145	.525
Traffic Area.County – Ferry.DSP	.298	.560
Road.ADD1– Enclosed Traffic Area.STATE_TOTAL_VMT	.187	.466
Residential Area.countyname – Enclosed Traffic Area.District1	.187	.550
Residential Area.State – Enclosed Traffic Area.STATUS	.343	.808

5.2. Results

An illustration of the tendency of the N-gram method to underestimate the value of correct attribute correspondences relative to TSim and overestimate the value of incorrect correspondences is displayed in table 2a(left) above for the GIS location dataset and in table 2b(right) for transportation dataset. For table 2a, in all five comparisons, the attributes are clearly related (ie: Ports.COUNTY and Piers.COUNTY). However, the N-gram method generates low EBD values for these comparisons (right column of table), while TSim generates high EBD values (left column of table). The reason for this is the inability of the N-gram method to relate two attributes together without the use of shared instances. As long as the compared attribute values are made of widely varying N-gram types, this method will always produce a low EBD value. On the other hand, because TSim does not rely on shared instances to determine semantic similarity, it is able to correctly assign a high EBD score between the attributes. On average, for the five comparisons above, the N-gram method underestimates the EBD score by 77%. Table 3b illustrates that the use of shared instances by the N-gram method can also lead to the exaggeration of similarity scores between unrelated attributes. For example, Traffic Area.County and Ferry.DSP both contain county data including the word "county", but DSP (which stands for 'Destination Port') also contains the names of towns and other geographic features. The N-gram method will match any instances containing the word "county" as well as other instances sharing common words, thus incorrectly raising its EBD computation. On average, for the five comparisons in table 3b, the N-gram method overestimates the EBD score by 266 %.

The results of the alignment of S_1 and S_2 of the compared tables for both the transportation dataset and the GIS location dataset using TSim are shown in tables 3a and 3b, respectively. Each cell contains the EBD value produced using TSim between a table from S_1 (names listed along the vertical axis of the table) and a table from S_2 (names listed along the horizontal axis of the table).

Table 3a and 3b. EBD values generated between tables of S_1 and S_2 of (a: transportation dataset (left table) (b: GIS location dataset (right table)

	Road	Address Area	Enclosed Traffic Area	Ferry
Road	.553	.225	.276	.503
Residential Area	.210	.552	.433	.407
Traffic Area	.136	.219	.958	.235
Ferry	.127	.237	.424	.564

	Flight Schools	Schools	Ports	NavWater ways
Flight Schools	.720	.615	.532	.503
Schools	.388	.768	.395	.540
Indian Lands	.489	.513	.486	.533
Piers	.496	.489	.633	.616

In table 3a, the EBD values obtained using TSim for the comparisons between Road-Road, Residential Area-Address Area, Traffic Area-Enclosed Traffic Area, and Ferry-Ferry are 0.553, 0.552, 0.958, and 0.564 respectively. Each of these represented the correct correspondences, and TSim identified them as those with the highest semantic similarity. In addition, tables that are semantically dissimilar, such as Ferry-Road and Traffic Area-Address Area were correctly recognized as such by TSim, as scores of .127 and .219 were generated. Similar results are also obtained in table 3b. Both of these datasets illustrate the tendency for the N-gram approach to overestimate incorrect correspondences and underestimate correct correspondences. For example, in table 3a, some of the EBD values produced via TSim for Road-Address Area, Road-Enclosed Traffic Area, and Road-Ferry are 0.22, 0.27 and 0.28 respectively. On the other hand, using the N-gram method, the scores generated for these comparisons were 0.44, 0.43 and 0.48 respectively. The scores were overestimated by 100%, 59% and 71% respectively. In table 3b, using TSim, the EBD values produced for Flight Schools(S_1)-Schools(S_2), Piers-Ports and Piers-NavWaterways are .615, .633 and .616. Using the N-gram approach, the scores generated are .182, .388 and .137. In this case, the N-gram method underestimated the scores by 70.5%, 38.8% and 77.8%, respectively.

6. Conclusion & Future Work

We outlined two algorithms that align distinct data sources using instance similarity. The first algorithm aligns instances between compared attributes by extracting distinct N-grams from them and measuring their semantic similarity by calculating an EBD value. The second algorithm, TSim, determines the semantic types of keywords in compared attributes using clustering and an external data source which leverages the Normalized Google Distance. Future efforts will focus on exploring the possibility of a hybrid instance matching technique that combines selected elements of the N-gram approach and TSim.

7. References

[1] E.Ralun and P. A. Bernstein, “A survey of approaches to automatic schema matching”, *VLDB Journal*, vol. V10, pp. 334-350, 2001.

[2] Bing Tian Dai, Nick Koudas, Divesh Srivastava, Anthony K. H. Tung, and Suresh Venkatasubramanian, "Validating Multi-column Schema Matchings by Type," *24th International Conference on Data Engineering (ICDE),* 2008.

[3] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster, “Putting context into schema matching.” in VLDB, 2006, pp. 307–318.

[4] R. H. Warren and F. W. Tompa, “Multi-column substring matching for database schema translation.” in *Proc. VLDB*, 2006, pp. 331–342.

[5] W.S. Li and C. Clifon, “Semint: a tool for identifying attribute correspondence in heterogeneous databases using neural networks,” *Data Knowl. Eng.*, vol. 33, no. 1,pp.49-84, 2000.

[6] Rudi Cilibrasi, Paul M. B. Vitányi: The Google Similarity Distance CoRR abs/cs/0412098:(2004)

[7] R. Gligorov, W. Kate, Z. Aleksovski, F. Harmelen: Using Google distance to weight approximate ontology matches. WWW 2007:767-776

[8] Jeffrey Partyka, Neda Alipanah Latifur Khan, Bhavani Thuraisingham and Shashi Shekhar, “Content-based Ontology Matching for GIS Datasets”, University of Texas at Dallas (UTD Technical Report # UTDCS-22-08).

[9] Shenghui Wang, Gwenn Englebienne and Stefan Schlobach, "Learning Concept Mappings from Instance Similarity", *Proceedings of the 7th International Semantic Web Conference, ISWC 2008*, LNCS 5318, pp 339-355, 2008.

[10] Christian Wartena and Rogier Brussee, "Instance-Based Mapping Between Thesauri and Folksonomies", In: *Proc. of the 7th International Semantic Web Conference*, ISWC 2008, LNCS 5318, pp 356-370, 2008.

R2D: A FRAMEWORK FOR THE RELATIONAL TRANSFORMATION OF RDF DATA

SUNITHA RAMANUJAM, ANUBHA GUPTA, LATIFUR KHAN, BHAVANI THURAISINGHAM

The University of Texas at Dallas, Richardson TX 75080, U.S.A
{sxr063200, axg089100, lkhan, bxt043000}@utdallas.edu

STEVEN SEIDA

Raytheon Company, Garland TX 75042, U.S.A
steven_b_seida@raytheon.com

The astronomical growth the World Wide Web resulted in data explosion that has, in turn, has given rise to a need for data representation methodologies and standards to present required information in a rapid and automated manner. The Resource Description Framework is one such standard proposed by W3C to address the above need. The ubiquitous acceptance of RDF on the Internet has resulted in the emergence of a new data storage paradigm, the RDF Graph Model, which, as with any data storage methodology, requires data modeling and visualization tools to aid with data management. This paper presents R2D (RDF-to-Relational), a relational wrapper for RDF Data Stores, which aims to transform, at run-time, semi-structured RDF data into an equivalent normalized relational schema, thereby bridging the gap between RDF and RDBMS concepts and making the abundance of relational tools currently in the market available to the RDF Stores. The primary R2D functionalities and mapping constructs, the high-level system architecture, and deployment sequence diagrams are presented along with algorithms and performance graphs for every stage of the transformation process and screen-shots of a relational visualization tool using R2D as evidence of the feasibility of the proposed work.

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

1. INTRODUCTION

The unleashing of the Internet has resulted in a plethora of information sources becoming available, making today's world increasingly networked and progressively more reliant on electronic sources of data. The need to augment human reasoning and decision making abilities has resulted in the emergence of an evolutionary stage of the World Wide Web, namely, the Semantic Web. The Semantic Web is envisioned to facilitate the automated storage, exchange, and usage of machine-readable information interspersed throughout the web [1]. To this end various standards are being developed to enable users to access information more efficiently and realize the Semantic Web vision. One standard, which is the current buzzword in the Semantic Web Community, is the Resource Description Framework [2], which is the foundation for the Semantic Web and the focus of the research presented in this paper. The RDF standard is proposed by the World Wide Web consortium for encoding knowledge with the express purpose of changing the web from being a platform for distributed presentations to one for distributed knowledge [3]. RDF's suitability to unstructured and semi-structured data that is typically available on the web, and the simplicity and flexibility offered by RDF data models have resulted in increasing demand for data stores that use the RDF Graph model and offer the ability to store and query RDF data [4].

The growing number of RDF stores have, as with any data store with massive amounts of information, spawned an associated requirement for tools and technologies for the management and visualization of this data. However, most of the current data modeling, data visualization, data management, and business intelligence tools that widely are available in the market today are still based on the more mature models such as relational and tabular models [5]. The tools available for RDF data are fewer and less mature than the selection for RDBMSs. Further, small and medium-sized organizations that are typically resource constrained may not have the ability or inclination to take risks associated with investing in fledgling technologies such as RDF and the tools for the same [6]. Relational databases have been around for several decades more than semantic web technologies, giving them the advantage of time to refine their tools and methodologies. For the same reasons, skilled personnel experienced in relational methodologies are available in greater numbers than RDF experts. In order to avoid the learning curves associated with new tools and continue to leverage the advantages offered by the traditionally-oriented 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 data and we propose one such solution. Our approach, called R2D (RDF-to-Database), is a bridge that hopes to enable existing traditional tools to work seamlessly with RDF Stores without having to make extensive modifications or waste valuable resources by replicating data unnecessarily. This paper expands on the work in [7, 8] and provides a relational interface to data stored in the form of RDF triples. It includes the ability to handle blank nodes and RDF container objects along with enhancements to the SQL-to-SPARQL transformation that now permit aggregation on RDF data. As before, the RDF Store is explored and mapped to a relational schema at run-time and end-users of visualization tools are presented with the normalized relational version of the store on which they can perform operations as they would on an actual physical relational database schema. The contributions of this paper are as follows.

- We propose a mapping scheme for the translation of RDF Graph structures to an equivalent normalized relational schema. The proposed mapping schema builds on the schema presented in [7] and includes several constructs and rules to handle a variety of blank nodes and RDF Container objects such as Bags and Sequences.
- Based on the RDF-to-RDBMS map file created, we propose a transformation process that presents a normalized, non-generic, domain-specific, virtual relational schema view of the given RDF store.
- We propose a mechanism to transform any relational SQL queries issued against the virtual relational schema into the SPARQL equivalent, and return the triples data to the end-user in a relational format. The proposed mechanism includes string matching procedures and aggregation facilities.
- We provide all of the above in the form of a JDBC interface that can be plugged into existing visualization tools and we present the feasibility of our algorithms and

processes through experiments conducted using the LUBM Benchmark data set, and an open source visualization tool, RDF store, and relational database.

The organization of the paper is as follows. Section 2 presents a brief overview of related research efforts in the relational-to-rdf arena. Section 3 describes R2D's system architecture and modus operandi, mapping constructs and types of relationships handled. 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 the map file generation process with and without various sampling methods and for a diverse range of queries on databases of various sizes and, lastly, Section 6 concludes the paper.

2. RELATED WORK

Several research efforts exist that attempt to bring relational database and semantic web concepts together, albeit from a perspective that is opposite to that considered in our work. The most notable amongst these in terms of the objectives being very closely aligned with ours is the RDF2RDB project [5]. Like in R2D, the authors in [5] attempt to arrive at a domain-specific, meaningful relational schema equivalent for an RDF store but the similarity ends there. RDF2RDB, like most of the other transformation efforts described below, involves data replication with the triples data being dumped into a relational schema, and therefore is subject to synchronization and space issues. Moreover, in order to successfully map the RDF data into an equivalent relational schema, 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 information with or without explicit ontology information. In the absence of RDF Schema definitions, R2D discovers the mapping through extensive examination of the triple patterns and the relationships between resources.

Furthermore, the relational mapping in [5] 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 [5] also do not discuss the details of how blank nodes are handled by their research, if at all, while R2D is capable of wading through a variety of blank nodes and arriving at meaningful transformations of the same. The Hybrid model presented in [9] is another mapping methodology that is similar to [5] in terms of relational schema generation and, hence, has the same drawbacks as [5].

The D2RQ project [10], an extensively adopted open source project, and one that our work is very closely modeled on in terms of the components/modules of the system, is another significant player in the RDBMS-RDF mapping arena.. The goals of D2RQ are the exact reverse of the goals of our research. While they attempt to create a mapping from a relational database to an RDF Graph, and transform RDF queries into corresponding SQL queries, thereby making relational data accessible through RDF applications, our goal is to enable RDF triples to be accessed through relational applications. Hence, despite the concept of mapping files and query conversions being common between D2RQ and R2D, each of the two researches address very different needs. The work in [11] is yet another effort that, like D2RQ, also uses a declarative meta schema consisting of quad map patterns that define the mapping of SQL data to RDF ontologies. RDF123 [12], 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 the spreadsheet semantics and RDF graphs. Triplify [13] 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.

Other mapping efforts in the reverse direction include the work presented in [14, 15, 16]. In [14] the authors use relational.OWL to extract the semantics of a relational database and automatically transform them into a machine-readable and understandable RDF/OWL ontology. The authors in [15, 16] 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. 3Store [17] is an implementation where the model includes non-application-specific tables such as triples, symbols, datatypes, etc. Using this model, it would be impossible for the user to determine the problem domain addressed by the model or to infer the schema by identifying the entities, the attributes, and any relationships that exists between any of them. R2D offers the users the ability to do just this and enables them to actually arrive at a complete Entity-Relationship Diagram using the RDF-to-Relational Schema transformation process.

The query processing component of R2D which comprises the SQL-to-SPARQL transformation process, once again, has no comparable counterpart while many efforts are underway in the other direction. In [18], the authors propose an algorithm to translate SPARQL queries with arbitrary complex optional patterns to an equivalent SQL statement to be fired against a single relational table called Triples(subject, predicate, object) that stores the RDF triples. The authors in [19] discuss a methodology that supports integration of heterogeneous relational databases using the RDF model. Given a set of semantic mappings between relational schemas and RDF ontology, the goal in [19] is to effectively answer RDF queries by rewriting them into a set of equivalent source SQL queries. An SQL-based RDF Querying Scheme is presented in [20] where the RDF querying capability is made a part of the SQL, however, the RDF data is, once again, stored as a collection of triples in a single database table. In [21], the authors partition the RDF graph data by adding an extra column to the triples table to store sub-graph information with the objective of reducing join costs and improving query performance.

As can be seen from the discussions above, none of the research efforts address the issue of enabling relational applications to access RDF data without data replication and, hence, to the best of our knowledge, R2D is the first endeavor to address this issue.

3. R2D Preliminaries

As stated earlier, the principal goal of this research is to ensure seamless availability of RDF data to existing tools, in particular, data visualization tools, that are equipped to work with relational or tabular data. The architecture of the proposed system and the deployment sequence are illustrated in Figure 1.

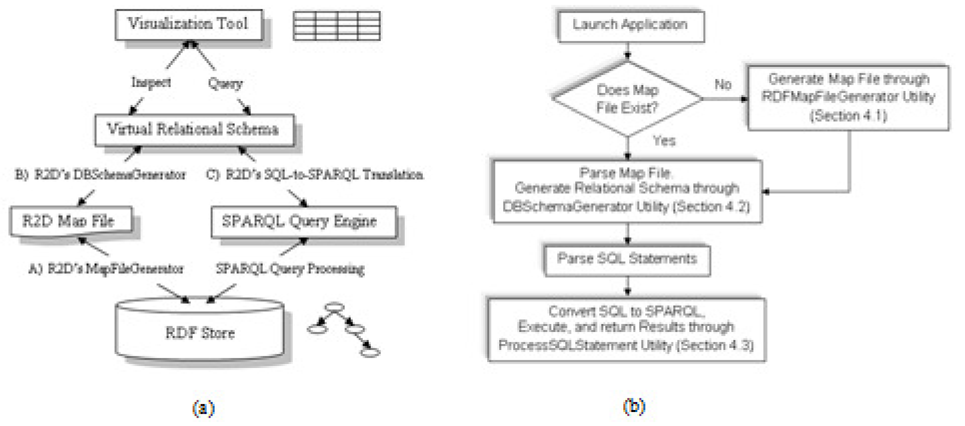

Visualization Tool
Inspect
Query
Virtual Relational Schema
B) R2D's DBSchemaGenerator
C) R2D's SQL-to-SPARQL Translation
R2D Map File
SPARQL Query Engine
A) R2D's MapFileGenerator
SPARQL Query Processing
RDF Store
(a)
Launch Application
Does Map File Exist?
No
Yes
Generate Map File through RDFMapFileGenerator Utility (Section 4.1)
Parse Map File. Generate Relational Schema through DBSchemaGenerator Utility (Section 4.2)
Parse SQL Statements
Convert SQL to SPARQL, Execute, and return Results through ProcessSQLStatement Utility (Section 4.3)
(b)

Figure 1. (a) R2D System Architecture; and (b) Deployment Sequence

The RDF Store at the bottom of Figure 1 (a) is examined by the RDFMapFileGenerator Algorithm (Item A in Figure 1(a)) and an RDF-to-RelationalSchema mapping file is generated, if it does not already exist, by the algorithm using the constructs discussed in Section 3.1. The DBSchemaGenerator Algorithm (Item B in Figure 1(a)) 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(a)) 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 transformation of RDF Graphs to virtual relational database schemas is the R2D mapping language which is a declarative language that expresses the mappings between RDF constructs and relational database schema constructs. In order to better explain the constructs comprising the R2D mapping language, examples from the sample scenario in Figure 2, based on the LUBM dataset, are included where applicable. The constructs of the current version of the mapping language are presented below.

3.1. *R2D Mapping Constructs*

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

The mapping constructs specific to an r2d:TableMap are as follows.

r2d:keyField: The r2d:keyField construct specifies the primary key attribute for the r2d:TableMap to which the field is attached. The data value associated with the field specified by r2d:keyField is the object of the "rdf:type" predicate belonging to the rdfs:class subject corresponding to its r2d:TableMap. ***Example:*** *An r2d:keyField (primary key) called "Student_PK" field is attached to the "Student" TableMap and one of its values, corresponding to the sample scenario in Figure 2, is "URI/StudentA".*

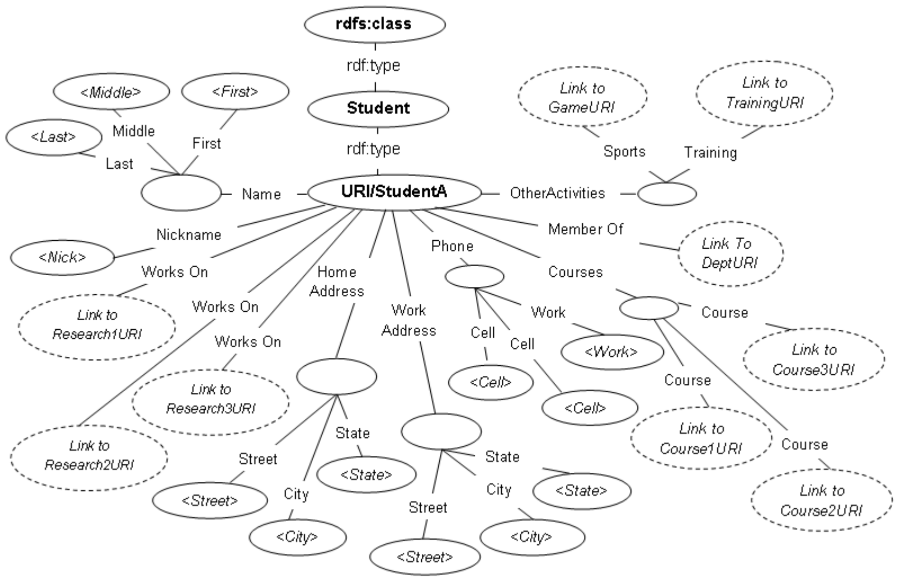


Figure 2: Sample Scenario based on LUBM Schema

r2d:ColumnBridge: r2d:ColumnBridges relate single-valued RDF Graph predicates/properties 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 "Nickname" and "Member Of" predicates in Figure 2 become r2d:ColumnBridges belonging to the "Student" r2d:TableMap*

r2d:MultiValuedColumnBridge(MVCB): Those RDF Graph predicates that have multiple object values for the same subject are mapped using the MVCB construct. MVCBs typically correspond to RDF constructs such as RDF containers (rdf:Bag, rdf:Alt, rdf:Seq) and RDF collections and are used to indicate N:1 and N:M relationships between the virtual relational schema tables. ***Example:*** *The "Works On" predicate in Figure 2 is an example of an MVCB mapping.*

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 "Student" r2d:TableMap.*

r2d:MultiValuedSimpleLiteralBlankNode (MVSLBN): 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 a MVSLBN.*

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/StudentA) and a predicate (Cell) concept associated with the blank node. The relational transformation for "Phone" involves the generation of an r2d:TableMap of the same name. This "Phone" r2d:TableMap includes as columns a "Type" field that holds the values of the predicates off of the MVBN (in our sample scenario, the "Type" field will hold a value of "Cell" and "Work"), and a "Value" field that holds the object values of the predicates off of the MVBN. Additionally, the r2d:TableMap also includes, as foreign key, the "Student_PK" column which references the primary key of the "Student" r2d:TableMap.*

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. RDF containers that represent collections of similar resource objects are represented using the SRBN construct. ***Example:*** *The object of the "Courses" predicate in Figure 2 is an example of a SRBN that has multiple resource objects that are instances of the "Course" 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 "Sports" and one "Training") class.*

r2d:MultiValued{Simple/Complex}ResourceBlankNode (MVSRBN and MVCRBN): Duplicate simple/complex resource blank nodes are represented using the MVSRBN and MVCRBN 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 "Courses" predicate in Figure 2 is replaced with multiple predicates each representing the courses taken in a particular year, such as "2007Courses, "2008Courses", and "2009Courses", each of which are SRBNs. The objects of these predicates together form an MVSRBN.*

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 which contains as fields every literal or resource leaf node object that is an element of the tree rooted at the r2d:MixedBlankNode.

The mapping constructs specific to single-valued and multi-valued column bridges and blank nodes are described below.

r2d:belongsToTableMap(BTTM): This construct connects a r2d:ColumnBridge or MVCB to an r2d:TableMap. Every r2d:ColumnBridge must specify a value for either this construct or the r2d:belongsToBlankNode construct. ***Example:*** *The "Nickname" predicate in Figure 2 is associated with the resource "URI/StudentA", an instance of the "Student" r2d:TableMap. Hence, the BTTM construct corresponding to "Nickname" r2d:ColumnBridge is set to a value of "Student", thereby connecting the ColumnBridge to a table.*

r2d:belongsToBlankNode (BTBN): This construct ties a r2d:ColumnBridge or MVCB to an SVBN or an MVBN. ***Example:*** *The "FirstName" r2d:ColumnBridge corresponding to the "First" predicate in Figure 2 is associated with the "Name" SVBN. Hence, for the "FirstName" r2d:ColumnBridge the BTBN construct is used to associate it to the "Name" blank node.*

r2d:refersToTableMap (RTTM): This construct is optional for column bridges and is only used for those triples that contain a resource object for a predicate. This construct is used to generate primary key-foreign key relationships within the virtual relational schema. ***Example:*** *The object of the "Member Of" predicate in Figure 2 is a resource that translates to another r2d:TableMap called "Department". Hence the "MemberOf" r2d:ColumnBridge includes the RTTM construct with a value of "Department".*

r2d:predicate: The r2d:predicate construct is used to store the fully qualified property name of the predicate which corresponds to the column bridge. 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

r2d:MultiValuedPredicate (MVP): The MVP construct is used in scenarios where there are multiple predicate names that refer to the same overall object type despite each individual object having a different value. r2d:MultiValuedPredicates are also used to keep track of the predicates associated with RDF containers and RDF collections. MVPs typically result in the creation of a "TYPE" column in the r2d:TableMap corresponding to the resource associated with the MVP. ***Example:*** *The predicates off of the "Phone" CLBN in Figure 2 are examples of a MVP called "Phone_Type" that represents the fact that multiple predicates (<Cell>, <Work>) refer to the same overall object type (i.e., a string representing phone number).*

r2d:datatype: This construct specifies the datatype of its column bridge and is derived from the rdfs:range predicate or, in its absence, by examination of the object values of the predicate.

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 elaborated on in Section 4.

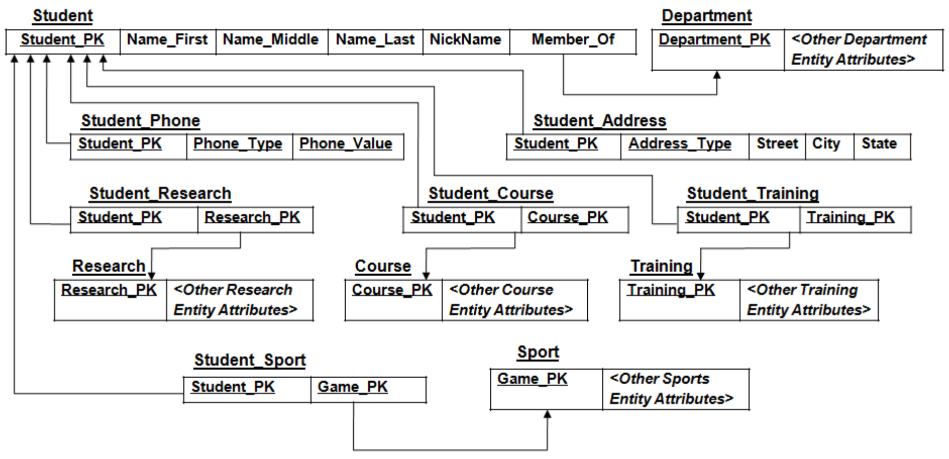


Figure 3: Equivalent Virtual Relational Schema generated by R2D for Figure 2

3.2. *Types of Relationships Addressed*

The predicates and various types of blank nodes in Figure 2 and the relationships they represent in the corresponding virtual relational schemata are discussed below. The simple predicates typically map to a column in a relational schema. Blank nodes with multiple distinct or non-distinct predicates such as "*Courses*", "*Phone*", and "*Other Activities*" typically highlight 1:N or N:M relationships, while blank nodes such as "*Name*", with literal predicates, are typically equivalent to columns.

(a) r2d:ColumnBridge Relationships (1:1 Relationships without Blank Nodes)
In this kind of a relationship, most often one side of the relationship translates into a column/attribute in the table represented by the other side of the relationship. An example of a 1:1 relationship without blank nodes in Figure 2 is the triple (<URI/StudentA> <Nickname> <Nickname>) referring to the relationship between an instance (<URI/StudentA>) of the Student class and his/her Nickname.

(b) r2d:SimpleLiteralBlankNode Relationships (1:1 Relationships with Blank Nodes)
These kinds of relationships are processed, for the purposes of transformation into a relational schema equivalent, by ignoring the blank node and treating the predicates of the blank nodes as multiple 1:1 relationships-without-blank-nodes to the subject of

the blank node. Each predicate of the blank node essentially becomes an attribute of the table representing the subject instance. An example of a 1:1 relationship with blank nodes in Figure 2 is the triple (<URI/StudentA> <Name> <blankNode>).

(c) r2d:ColumnBridge Relationships with r2d:refersToTableMap construct (N:1 Relationships without Blank Nodes)
In N:1 relationships without Blank Nodes, the primary key of the table representing the instance on the "1" side of the relationship is included as a foreign key in the table representing the instance on the "N" side of the relationship. An example of a N:1 relationship without blank nodes in Figure 2 is the triple (<URI/StudentA> <MemberOf> <Link to DepartmentID>) referring to the relationship between an instance (<URI/StudentA>) of the Student class and an instance of the Department class.

(d) r2d:MultiValuedColumnBridge (MVCB) Relationships with/without r2d:refersToTableMap construct (N:1 or N:M Relationships without Blank Nodes)
r2d:MultiValuedColumnBridges with literal objects (i.e., without r2d:refersToTableMap construct) are equivalent to multi-valued attributes in relational terminology and, hence, result are considered to represent 1:N relationship between the subject and the object of the predicate corresponding to the MVCB. Thus, for MVCBs, a new table is generated with a foreign key that references the table corresponding to the class to which the subject belongs. MVCBs with resource objects (i.e. with r2d:referstoTableMap construct) typically represent N:M relationships and hence, the processing of such MVCBs is similar to the processing discussed in category (f) below. An example of an MVCB with resource objects in Figure 2 is the triple (<URI/StudentA> <WorksOn> <Link to Research>) referring to the relationship between an instance (<URI/StudentA>) of the Student class and instances of the Research class.

(e) r2d:ComplexLiteralBlankNodes and r2d:MultiValuedSimpleLiteralBlankNodes (N:1 Relationships with Blank Nodes)
These relationships typically result in the generation of a separate table with a foreign key that references the table corresponding to the class to which the subject of this blank node object belongs. An example of a r2d:ComplexLiteralBlankNode in Figure 2 is (<URI/StudentA <Phone> <blankNode>) and an example of an r2d:MultiValuedSimpleLiteralBlankNode is (<URI/StudentA> <Home/Work Address> <blankNode>). Both these examples result in the generation of r2d:MultiValuedPredicates due to the presence of distinct predicates for the phone number and address nodes.

(f) r2d:SimpleResourceBlankNodes and r2d:ComplexResourceBlankNodes (N:M Relationships with Blank Nodes)
N:M relationships with or without blank nodes result in the generation of a new join table that has, as foreign keys, the primary keys of the tables corresponding to the classes to which the subject and the resource object belong. An example of an N:M relationship with a blank node leading to similar object resources (i.e., a blank node of type r2d:SimpleResourceBlankNode) in the scenario in Figure 2 is the triple (<URI/StudentA> <Projects> <blankNode>) and an example of one with different object resources (r2d:ComplexResourceBlankNode) is the triple (<URI/StudentA> <OtherActivities> <blankNode>).

This background on R2D fundamentals provides the foundation behind R2D functionalities, the details of which, along with the details of the algorithms that comprise the R2D framework, are presented in a comprehensive manner in the next section.

4. 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 the previous section. These include A) an algorithm that would, given an RDF Data Store, derive the mapping file automatically, B) an algorithm to parse the generated mapping file and generate, for the RDF Store, a list of relational tables, columns, and the relationships between them, and C) an algorithm to transparently transform any SQL statements issued against the virtual relational schema into its SPARQL equivalent, and return the results from the RDF Store in a relational/tabular format. The various modules highlighted in Figure 1 and the corresponding algorithms are described at length in the following subsections.

4.1. *RDFMapFileGenerator*

The first step in the R2D Framework is the map file generator process accomplished using the RDFMapFileGenerator algorithm that takes an RDF store as input and automatically generates an RDF-to-Relational mapping file as output. Notional mappings between some key OWL/RDFS Ontology terminologies and R2D constructs to relational concepts can be found in [8].

However, the transformation process is not always as straightforward and as well-defined as the notional mappings suggest. As mentioned earlier, 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 and the details are listed below.

Algorithm 1 RDFMapFileGenerator (*RDF*)

```
Input:  RDF: The RDF Store of Interest
Output: RDF-to-Relational Schema Mapping File
 1: Get sampling type, get/calculate sample size, calculate sample period (if systematic sampling)
 2: If exists(RDFSchema Information) then
 3:     For every resource that is an instance of rdfs:class
 4:         TableMaps ← resource_name    //add
 5:     End For
 6:     For every resource that is an instance of rdf:property
 7:         Get/Create TableMap, tblMap, corresponding to rdf:domain value of resource
 8:         tblMap.ColumnBridges ← PropertyResource_name    //add
 9:         tblMap.ColumnBridges.datatype ← PropertyResource's rdf:range value
10:     End For
11: End if
12: For every unprocessed (data) resource in the RDF store
```

```
13:   Create a TableMap, tblMap, for this resource
14:   For every predicate of the resource
15:      If object of predicate is literal then
16:         literalColumns += ProcessLiteralPredicate(resource, tblMap, predicate)
17:      If Object of predicate is a blank node then
18:         Call ProcessBlankNodePredicate(resource, tblMap, predicate)
19:      If Object of predicate is a resource then
20:         resourceColumns += ProcessResourcePredicate(resource, tblMap, predicate)
21:         ConsolidateResourcePredicates(resourceColumns)
22:         tblMap.setColumns(literalColumns); tblMap.setColumns(resourceColumns)
23:      End if
24:   End For
25:   If NOT(similarTableExists(tblMap)) then
26:      TableMaps += tblMap; otherwise discard tblMap
27:   End if
28:   If sampleSize reached
29:      Exit
30:   End if
31: End For
```

The RDFMapFileGenerator algorithm generates mappings for RDF Stores with and without ontological information in the form of RDF Schema definitions such as rdfs:class, rdf:property, etc. This algorithm arrives at an RDF-to-Relational mapping file through extensive exploration of the triples data in the RDF Store and, consequently, is a bottleneck in the transformation process in terms of the response times. As a result, a number of sampling methods have been incorporated in the algorithm as can be seen in line 1 of the RDFMapFileGenerator Algorithm.

For RDF Stores without ontological information, two types of data sampling have been implemented, namely, Convenience/Haphazard Sampling, and Systematic sampling. In the case of stores containing ontological information, two variations of Stratified Sampling have been implemented; one where the sample size for each class is proportional to the class size, and the other where the sample size is independent of the class size, i.e., the sample size is the same for each class. These sampling methods have resulted in large reductions in response times as can be seen in Section 5.

The data structure discovery process as illustrated in Figure 1 is as follows. When structural information about the triples database is present, lines 2-11 of RDFMapFileGenerator discover the schema definitions and create appropriate Table and Column mappings based on the schema information.

Lines 12-31 process instance data to identify and account for those predicates that may not have been defined through explicit rdf:property definitions. This is done using three procedures, ProcessLiteralPredicate (Line 16), ProcessResourcePredicate (Line 20), and ProcessBlankNodePredicate (Line 18). The **ProcessLiteralPredicate** procedure, as the name suggests, is used to process predicates that have literal objects (such as *Nickname* predicate). 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 **ProcessResourcePredicate** procedure 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 the duplicate predicates (i.e., predicates that have objects belonging to the same object class) are examined and eliminated and this is done through the **ConsolidateResourcePredicates** procedure (Line 21). During the consolidation process, any (duplicate) potential columns that refer to the same object resource class (such as the *WorksOn* predicate) 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 values (such as the predicates off the *Phone* blank node), a MultiValuedPredicate object is created which reflects this fact. These MultiValuedPredicates typically become "TYPE" fields in the corresponding relational schema.

Predicates leading to blank nodes are handled through the **ProcessBlankNode** procedure. In this procedure, for every blank node encountered an object of type BlankNode is created. If every predicate off of the blank node contains a literal object (such as the *Name* and *Phone* blank nodes) then, for each predicate off of the blank Node, the ProcessLiteralPredicate procedure is called which works as described above. 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) 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 *Courses* 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 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 tree 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 the set 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.

Further, every resource object encountered and processed is stored in memory in order to avoid duplicate processing of the same in the event of multiple triples containing the same resource object. This information serves to improve the performance of the algorithm. When these "similar" resources are encountered during instance data processing, the algorithm skips the potential TableMap creation process and the time-consuming duplicate-TableMap detection process, thereby resulting in better efficiency.

Handling of Predicates with Object Resources belonging to multiple r2d:TableMaps *(i.e., a Foreign Key that has multiple tables that it needs to reference):*
RDF Graphs consists of many examples where the relational transformation creates a situation where an attribute A_{FK} in Entity E_{FK} could hold values corresponding to multiple entities, say E_1 to E_N (Let the set of attributes of each of these E_is be A_1, A_2, … A_N). This situation is handled as follows.
The attribute list of E_{FK}, A_{EFK} is modified to include fields that reference the key field attributes of each of the entities, E1 to EN, which A_{FK} references. Thus,
$A_{EFK} = A_{EFK} \text{ U } A_{ReferencingE1PK} \text{ U } A_{ReferencingE2PK} \ldots\ldots \text{ U } A_{ReferencingENPK} - A_{FK}$
Lastly, each attribute $A_{ReferencingEiPK}$ in Entity E_{FK} is set to reference the key attribute of E_i (E_{iPK}). For every row in EFK, one or more of the attributes $\{A_{ReferencingE1PK}, A_{ReferencingE2PK}, \ldots\ldots, A_{ReferencingENPK}\}$ will have a value while the others will be null. Since the relational schema corresponding to the given RDF graph generated by R2D is virtual involving no physical space/resource utilization, having multiple columns, many of which could be null, to represent the above scenario (foreign key referencing multiple tables) does not result in any resource wastage and is a simple solution to this requirement. An example of such a scenario would be the triple

<StudentURI (Subject), Advisor(Predicate), AdvisorURI (Object)>.

The Advisor object of a student resource could contain an instance from any one of the classes in the set {Full Professor, Associate Professor, Assistant Professor}. Again, the relational transformation of the above scenario would consist of four tables, namely, *Student*, *FullProfessor* (FP), *AssistantProfessor* (ASP), and *AssociateProfessor* (ACP). The *Student* table contains an *Advisor* column which is a foreign key. This foreign key needs to reference all three professor tables. As described above, this situation is handled by adding three separate columns to the *Student* table, *Advisor1* referencing the primary key of *FP*, namely *FP_PK*, *Advisor2* referencing *ASP*, and *Advisor3* referencing *ACP*. Further, for every row in the *Student* table only one of the three *Advisor* columns contains a value while the other two are null.
As another example, let us consider the following triple

<PublicationURI (Subject) Author (Predicate) AuthorURI(Object)>,

The Author object of a publication resource could contain instances from any of the classes in the set {Full Professor (FP), Associate Professor (ACP), Assistant Professor (ASP), Graduate Student (GS), Undergraduate Student (UGS)}. Applying the consolidate method described above (in the Advisor example) results in the addition of five separate columns to the *Publication* table, *Author1* through *Author5*, referencing the primary keys of *FP*, *ACP*, *ASP*, *GS*, and *UGS* tables respectively. In this example, the 5 fields are not mutually exclusive, unlike in the Student-Advisor scenario, and, thus, any or all of the 5 *Author* fields could contain values for each publication record.

4.2. *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 by the RDFMapFileGenerator algorithm in Section 4.1 and returns a virtual, appropriately normalized relational database schema consisting of entities/tables and the relationships between them.

The DBSchemaGenerator Algorithm is an enhanced version of the algorithm in [7] in terms of its ability to handle a variety of blank nodes including nested blank nodes. For each entry of type r2d:TableMap in the map file, a relational table, RelTable, is created in the virtual relational database schema. Entries of type r2d:ColumnBridge and r2d:MultiValuedColumnBridge whose r2d:belongsToTableMap value corresponds to the TableMap, RelTable, are processed as follows. Every entry of r2d:ColumnBridge simply becomes a column in RelTable. If the r2d:ColumnBridge refers to another resource (as indicated by the r2d:refersToTableMap construct), a foreign key relationship is established between RelTable and the referred-to table. For every entry of type r2d:MultiValuedColumnBridge, which is comparable to multi-valued attributes in relational database terminology, a new table, NormTable, is created and the r2d:MultiValuedColumnBridge as well as the primary key of RelTable are added as columns to NormTable. Further, if the predicate corresponding to the r2d:MultiValuedColumnBridge is a r2d:MultiValuedPredicate, an additional "TYPE" column is created and added to NormTable. If the r2d:MultiValuedColumnBridge is a literal the NormTable type is set to "LiteralMVCBTable"; otherwise it is set to "ResourceMVCBTable".

Non-nested blank nodes of various kinds are handled as follows. For r2d:SimpleLiteralBlankNodes (such as the blank node object of the *Name* predicate) of the kind illustrated in Section 3, Figure 2 every r2d:ColumnBridge entry that belongs to the blank node (as indicated by the r2d:belongsToBlankNode construct) is simply added as a column to the Table to which the r2d:SimpleLiteralBlankNode belongs (as indicated by the r2d:belongsToTableMap construct for the blank node). The processing of r2d:ComplexLiteralBlankNodes (such as the object of the *Phone* predicate) is very similar to the processing of r2d:MultiValuedColumnBridges described above with the difference being the table type of the created table, which is set to "CLBNTable". Entries of type r2d:SimpleResourceBlankNode (object of the *Courses* predicate) and r2d:ComplexResourceBlankNodes (object of the *OtherActivities* predicate) 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 r2d:MultiValuedSimpleLiteralBlankNode, results in the creation of a new table, contrary to the r2d:SimpleResourceBlankNode 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 r2d:MultiValuedSimpleLiteralBlanknode. The processing of r2d:MultiValuedComplexLiteralBlanknode, r2d:MultiValuedSimpleResourceBlanknode, and r2d:MultiValuedComplexResourceBlankNode 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". The table type values are set according to the type of blank nodes encountered. The reason for having table types and blank nodes is to maintain knowledge of the RDF graph structure in order to accurately translate SQL Statements issued against the relational schema into its appropriate SPARQL equivalent for precise data retrieval.

The final type of blank nodes processed by DBSchemaGenerator is mixed/nested blank nodes where the predicates off of the blank nodes are any combination of literals, resources, and other blank nodes. Due to the limitless kinds of such structured combinations that are possible, it would be impossible to even attempt to arrive at a corresponding normalized representation of the same. Hence, mixed/nested blank nodes of type r2d:MixedBlankNode are handled by creating a table, NormTable, which has as columns the primary key column of the table corresponding to the blank node's r2d:belongsToTableMap construct, and the literal and resource objects that are at the leaf nodes of the tree rooted at the topmost mixed/nested blank node. This is achieved through a recursive procedure that explores the predicates in a depth-first manner.

4.3. *SQL-to-SPARQL Translation*

The SQL-to-SPARQL Translation procedure, the last procedure in the deployment sequence illustrated in Figure 1 (b), corresponds to the final phase of the R2D transformation process where SQL statements issued against the virtual relational schema are parsed, translated into equivalent SPARQL queries that are executed against the RDF Store, and the results are returned in relational format. The SQL-to-SPARQL Translation algorithm, which takes an SQL Statement as input and returns an appropriate SPARQL equivalent as output, is an enhancement over the work in [7] with functionalities added to process queries involving underlying blank nodes, and to provide pattern matching and data aggregation abilities. The details of the algorithm are listed below.

Algorithm 2 SQL-to-SPARQL Translation (*SQL*)

```
Input:   SQL: SQL Query
Output:  Tabular results from execution of equivalent SPARQL Query
 1:  Parse the input SQL query
```

```
2:  listOfFields ← Array containing fields in the SELECT clause
3:  listOfTables ← Array containing tables in the FROM clause
4:  whereClause ← portion of the SQL Query after the WHERE keyword
5:  **If exists**(GROUP BY clause) **then**
6:      groupByField ← Array containing aggregated fields in SELECT clause
7:      groupByFunction ← Array containing aggregation functions on fields in SELECT clause
8:  **End if**
9:  SPARQLQuery ← **ProcessQuery**
10: **Execute** SPARQLQuery
11: **For** every row in the result set
12:     **For** every field in the SPARQL SELECT list
13:         **If isFieldPK**(field) **then**
14:             Replace field with the field's table's ?subject variable
15:         **End if**
16:         ResultRow  ← ResultRow U fieldValue of field from line 19's result set
17:     **End For**
18:     **If** GROUP BY Fields present **then**
19:         **For** every groupByField and groupByFunction in list
20:             Get the groupByField Value from line 19's result set
21:             **If** (current ResultRow == previous ResultRow) **then**
22:                 Perform aggregation per groupByFunction for the groupByField
23:             **Else**
24:                 ResultRow ← ResultRow U groupByField value
25:             **End if**
26:         **End For**
27:     **End if**
28:     QueryResults ← QueryResults U ResultRow
29: **End For**
```

Table 1: SQL-to-SPARQL Translation Algorithm - Supporting Procedures

SQL2SPARQL TRANSLATION ALGORITHM – SUPPORTING PROCEDURES	
Procedure, its Input, and its Output	**Short Description**
ProcessQuery **Input:** List of Fields, Tables, and Where Clause **Output:** Equivalent SPARQL Query.	This procedure takes the list of fields and tables, and the where clause in the original SQL query as input and generates a SPARQL equivalent of the same as output. The SPARQL SELECT list is generated within this procedure and the SPARQL WHERE and FILTER clauses are generated using the ProcessWhereClause and ProcessPredicatesForTables procedures called within this procedure.
ProcessWhereClause **Input:** SQL WHERE clause **Output:** SPARQL FILTER clause	This procedure examines the SQL WHERE clause to identify those fields that have been used in the WHERE clause but are not a part of the SELECT list. Resolution/conversion of the LIKE SQL construct

constructs	into an equivalent REGEX construct in SPARQL is also performed here.
ProcessPredicatesForTables **Input:** List of Tables and Fields in the SQL statement **Output:** SPARQL WHERE and FILTER clause constructs	Generation of the SPARQL WHERE clause and additions to the SPARQL FILTER clause are performed by this procedure. This is where most of the complexity of the SQL2SPARQL Translation algorithm lies as predicates corresponding to every table/column/blank node type are processed and transformed here.

The SQL-to-SPARQL Translation algorithm transforms single or multiple table queries with or without multiple where clauses (connected by AND, OR, or NOT operators) and Group By clauses. Within each individual where clause, the algorithm handles operators in the following set – {>, <, =, <=, >=, !=, LIKE}.

Lines 1-14 of the algorithm essentially perform parsing of the input query to identify the tables, fields, and the where clause and Group By clause, if present. The ProcessQuery procedure, called in line 15, transforms the SQL Query into its SPARQL form while lines 16-29 execute the generated SPARQL query, process the results, and present the same in a tabular format. Lines 18-27 perform data aggregation as per the Group By functions specified in the SQL Statement. Aggregation is achieved by appending an ORDER BY clause to the transformed SPARQL query and the actual group functions are calculated on the data obtained through the execution of the appended SPARQL query. Due to space constraints, a detailed description of this algorithm is omitted from this paper and can instead be found in [8].

5. IMPLEMENTATION SPECIFICS

The hardware used for the Map File creation process and the LUBM queries were executed was a personal computer running the Windows Vista operating system with 4GB RAM and 2 GHz Intel Dual Core processor. The software platforms and tools used include Jena 2.5.6 [22] to store the RDF triples data, MySQL 5.0 to house the RDF triples data persistently, Java 1.5 for development of the algorithms and procedures detailed in Section 4, DataVision v1.2.0 [23] to visualize and generate reports based on the RDF data, and GRUFF v1.0.19 [24] to compare the performance of R2D queries against.

5.1. *Experimental Dataset*

The LUBM dataset [25], which consists of a university domain ontology comprising resources such as Universities, Departments, Professors, Students, Courses, etc., was used in the experimentation process. In order to illustrate relationalization of blank nodes, we made certain modifications that involve additions of blank nodes to the LUBM Schema. These modifications include the addition of an *EmailAddress* r2d:SimpleLiteralBlankNode that involved altering the original simple literal *EmailAddress* property of resources into an SLBN consisting of two simple literal predicates, *PrimaryEmail* and *SecondaryEmail.* The second type of blank node added was an r2d:ComplexLiteralBlankNode called *ContactNo* which was created by modifying the original simple literal *Phone* property belonging to all *Professor* (and its subclasses) resources into a blank node with multiple simple literal *CellPhone* predicates and one simple literal *HomePhone* predicate. Query numbers 1, 4, and 8 in the LUBM test queries include selection of fields corresponding predicates belonging to the SLBN and CLBN and query performance of the same is illustrated in Figure 9.

5.2. *Experimental Results*

The relational equivalent of the RDF Graph in Figure 2 was generated using the RDFMapFileGenerator and DBSchemaGenerator Algorithms detailed in Section 4 and 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 time taken by the map file generation process without any data sampling incorporated for RDF stores of various sizes, with and without ontological information, was compared with time taken for the same process when several sampling methods are applied and the results are illustrated in Figure 4. The process is especially time-intensive for large databases without structural information but this is only to be expected since the RDFMapFileGenerator has to explore every resource to ensure that no property is left unprocessed.

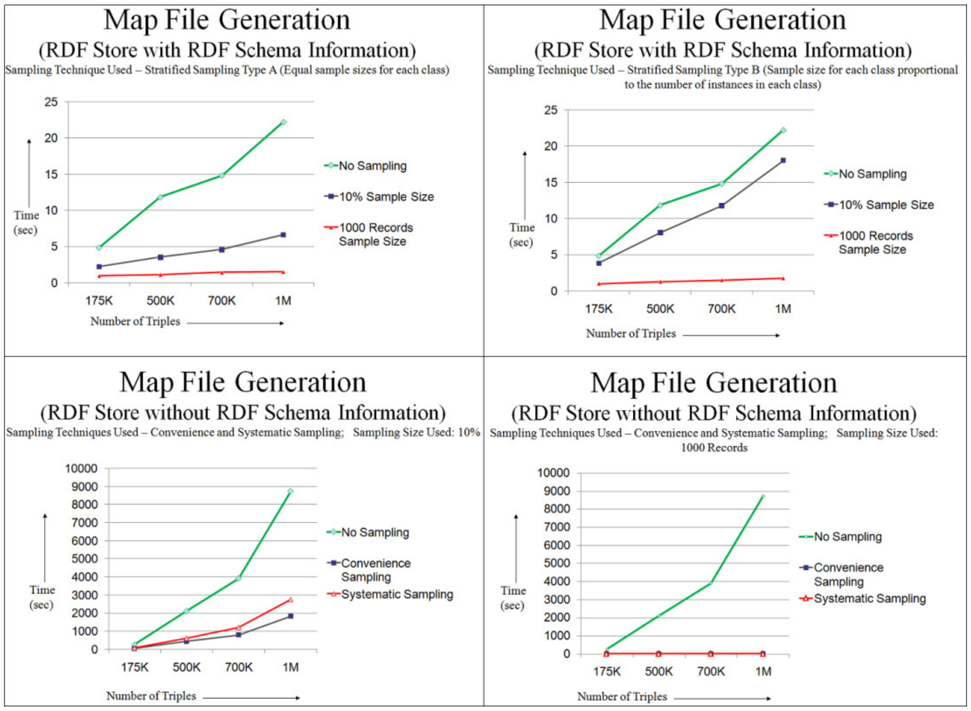
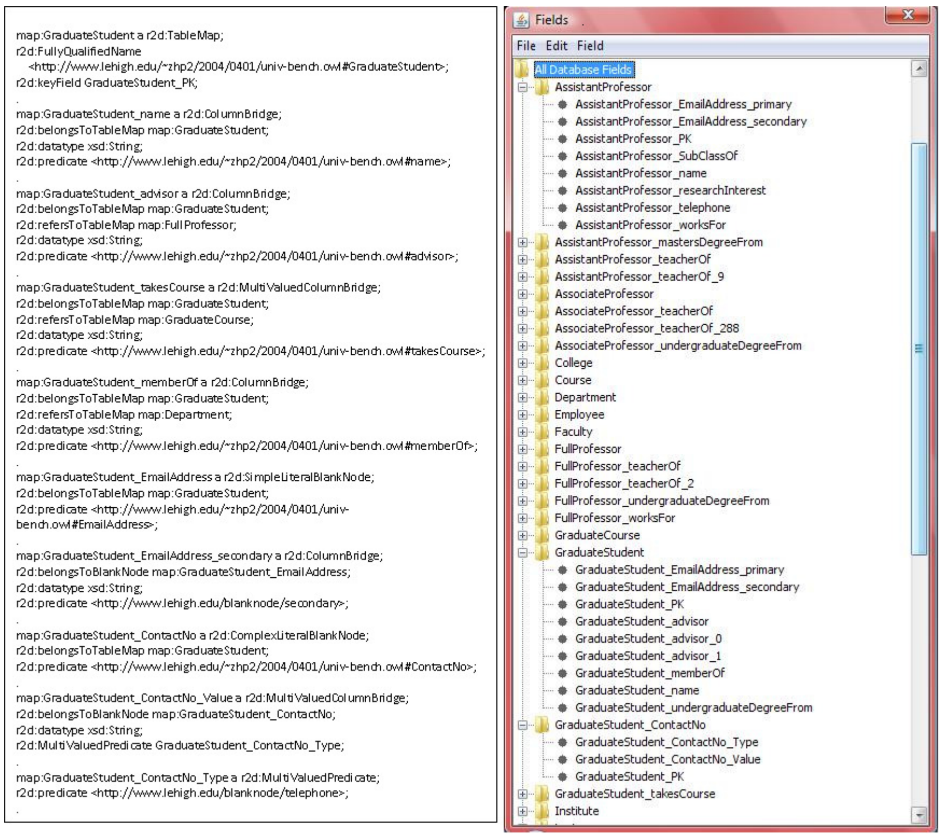


Figure 4: Map File Generation Times with and without Sampling

The sampling techniques applied improved the performance of the algorithm by a large factor, as can be seen in Figure 4. The processing times resulting from Convenience Sampling with sample sizes consisting of a fixed number of records are independent of the size of the data store and are almost constant since this technique only processes the first “n” rows regardless of the size of the database. Systematic sampling, on the other hand, does not yield as flat a line as Convenience sampling in the graphs above as it involves selecting samples periodically from the entire data store and, hence, is not as independent of the size of the data store as the former. For a similar reason, the Stratified Sampling scenario where the sample size is equally divided between the number of classes (Type B), regardless of the number of resources in each class, yields an almost constant response time contrary to its counterpart where the sample size for each class is proportional to the number of resources in each class (Type A).

Sampling techniques are especially useful in scenarios where the structure of similar resources are quite well defined with only minor variations as, in such situations, the sampling methods do not run the risk of overlooking structural information that is not evident in the chosen sample data subset. Further, if a domain expert with knowledge of the structural information of the RDF store is available, the automatic map file generation process becomes optional. This step can be bypassed, and the time saved, by providing the map file manually.

Figure 5 includes an excerpt from the map file generated by the RDFMapFileGenerator algorithm along with an inset of a part of the database schema as seen by DataVision.



```
map:GraduateStudent a r2d:TableMap;
r2d:FullyQualifiedName
   <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#GraduateStudent>;
r2d:keyField GraduateStudent_PK;
.
map:GraduateStudent_name a r2d:ColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#name>;
.
map:GraduateStudent_advisor a r2d:ColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:refersToTableMap map:FullProfessor;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#advisor>;
.
map:GraduateStudent_takesCourse a r2d:MultiValuedColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:refersToTableMap map:GraduateCourse;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#takesCourse>;
.
map:GraduateStudent_memberOf a r2d:ColumnBridge;
r2d:belongsToTableMap map:GraduateStudent;
r2d:refersToTableMap map:Department;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf>;
.
map:GraduateStudent_EmailAddress a r2d:SimpleLiteralBlankNode;
r2d:belongsToTableMap map:GraduateStudent;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#EmailAddress>;
.
map:GraduateStudent_EmailAddress_secondary a r2d:ColumnBridge;
r2d:belongsToBlankNode map:GraduateStudent_EmailAddress;
r2d:datatype xsd:String;
r2d:predicate <http://www.lehigh.edu/blanknode/secondary>;
.
map:GraduateStudent_ContactNo a r2d:ComplexLiteralBlankNode;
r2d:belongsToTableMap map:GraduateStudent;
r2d:predicate <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#ContactNo>;
.
map:GraduateStudent_ContactNo_Value a r2d:MultiValuedColumnBridge;
r2d:belongsToBlankNode map:GraduateStudent_ContactNo;
r2d:datatype xsd:String;
r2d:MultiValuedPredicate GraduateStudent_ContactNo_Type;
.
map:GraduateStudent_ContactNo_Type a r2d:MultiValuedPredicate;
r2d:predicate <http://www.lehigh.edu/blanknode/telephone>;
.
```

```
All Database Fields
  AssistantProfessor
    AssistantProfessor_EmailAddress_primary
    AssistantProfessor_EmailAddress_secondary
    AssistantProfessor_PK
    AssistantProfessor_SubClassOf
    AssistantProfessor_name
    AssistantProfessor_researchInterest
    AssistantProfessor_telephone
    AssistantProfessor_worksFor
  AssistantProfessor_mastersDegreeFrom
  AssistantProfessor_teacherOf
  AssistantProfessor_teacherOf_9
  AssociateProfessor
  AssociateProfessor_teacherOf
  AssociateProfessor_teacherOf_288
  AssociateProfessor_undergraduateDegreeFrom
  College
  Course
  Department
  Employee
  Faculty
  FullProfessor
  FullProfessor_teacherOf
  FullProfessor_teacherOf_2
  FullProfessor_undergraduateDegreeFrom
  FullProfessor_worksFor
  GraduateCourse
  GraduateStudent
    GraduateStudent_EmailAddress_primary
    GraduateStudent_EmailAddress_secondary
    GraduateStudent_PK
    GraduateStudent_advisor
    GraduateStudent_advisor_0
    GraduateStudent_advisor_1
    GraduateStudent_memberOf
    GraduateStudent_name
    GraduateStudent_undergraduateDegreeFrom
  GraduateStudent_ContactNo
    GraduateStudent_ContactNo_Type
    GraduateStudent_ContactNo_Value
    GraduateStudent_PK
  GraduateStudent_takesCourse
  Institute
```

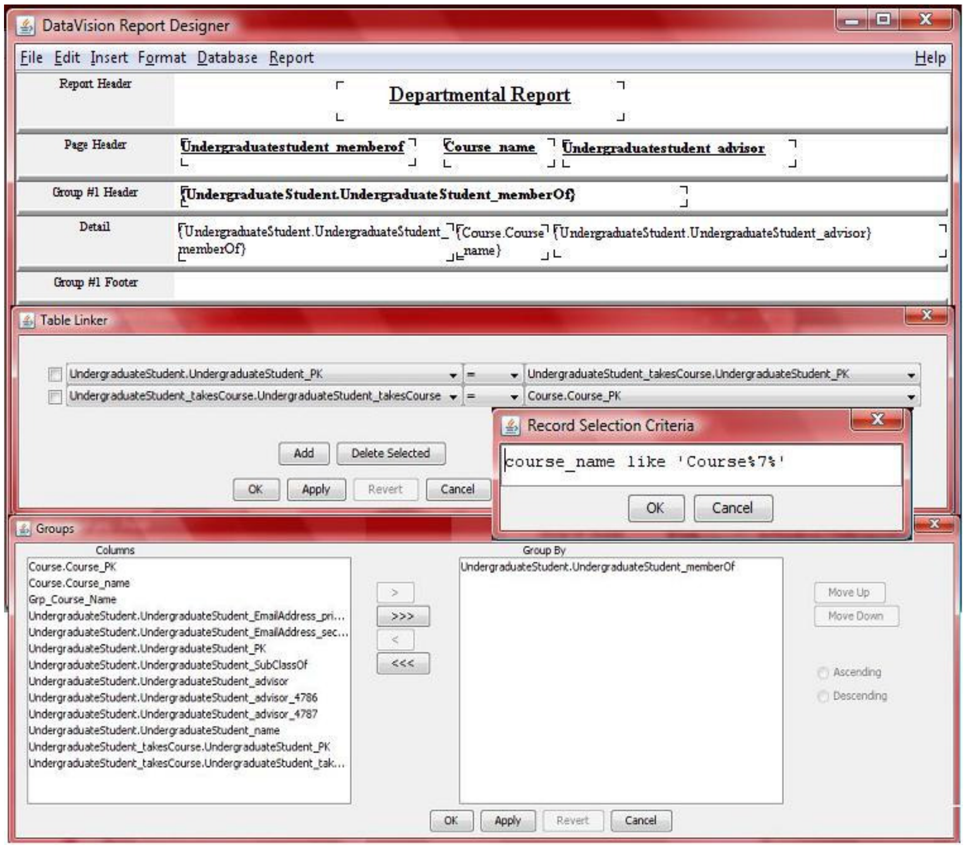
Figure 5: Map File Excerpt and a portion of the Equivalent Relational Schema as seen by DataVision

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. As can be seen, the various blank nodes that are part of the dataset are appropriately resolved and normalized into 1:N or N:M tables, or columns in existing tables using the algorithm described in Section 4.2. The r2d:SimpleLiteralBlankNode associated with *Professor/Student-EmailAddress* is resolved into columns belonging to the *Professor/Student* tables and the r2d:ComplexLiteralBlankNode associated with *GraduateStudent-ContactNo* is resolved into a 1:N table of the same name.

Note that there are several tables in the virtual relational schema that seem like duplicates (such as *AssistantProfessor_TeacherOf* and *AssistantProfessor_TeacherOf_9*, *FullProfessor_TeacherOf* and *FullProfessor_TeacherOf_2*). These tables are not actually duplicates. The first table in the pair is a join table for the N:M relationship that exists between *<Assistant/Full>Professor* and *Course* classes while the second table in the pair is the join table for the N:M relationship that exists between the *<Assistant/Full>Professor* and *GraduateCourse* classes. The join table names in R2D's virtual relational schema are derived from the relevant predicate names. Since the predicate names of the *Professor-Course* triples and the *Professor-GraduateCourse* triples are identical in the LUBM dataset, the RDFMapFileGenerator algorithm appended a unique identifier (the numbers at the end of the table names) to the second join table in order to avoid duplicate table names in the virtual relational schema.

Figure 6 is a screenshot of DataVision's Report Designer which illustrates DataVision's query building process for a sample query involving the SQL LIKE operator and a GROUP BY clause. Based on the fields chosen (in the "Report Designer" window), the table linkages (i.e., joins, illustrated in the "Table Linker" inset) specified, and additional record selection and grouping criteria specified (illustrated in the "Record Selection Criteria" and "Groups" insets respectively), DataVision generates an appropriate SQL query, as shown in the "SQL Query" inset in Figure 7, to extract the required data. At this juncture, the Statement Interface, the Prepared Statement Interface, and the ResultSet Interface that are part of the JDBC interface are invoked. These interfaces trigger the SQL-to-SPARQL Translation algorithm, which generates a SPARQL equivalent of the given SQL statement as illustrated in Figure 7, and return the obtained results to DataVision in the expected tabular format, as illustrated in Figure 8.

While 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 are as displayed in Figure 8 below.



Departmental Report

Undergraduatestudent memberof | Course name | Undergraduatestudent advisor

{UndergraduateStudent.UndergraduateStudent_memberOf}

{UndergraduateStudent.UndergraduateStudent_memberOf} {Course.Course_name} {UndergraduateStudent.UndergraduateStudent_advisor}

```
UndergraduateStudent.UndergraduateStudent_PK  =  UndergraduateStudent_takesCourse.UndergraduateStudent_PK
UndergraduateStudent_takesCourse.UndergraduateStudent_takesCourse  =  Course.Course_PK
```

```
course_name like 'Course%7%'
```

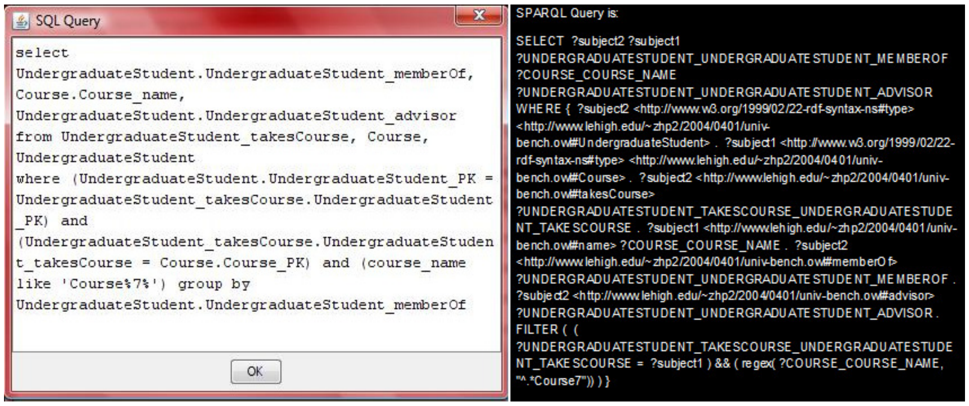
Columns:

```
Course.Course_PK
Course.Course_name
Grp_Course_Name
UndergraduateStudent.UndergraduateStudent_EmailAddress_pri...
UndergraduateStudent.UndergraduateStudent_EmailAddress_sec...
UndergraduateStudent.UndergraduateStudent_PK
UndergraduateStudent.UndergraduateStudent_SubClassOf
UndergraduateStudent.UndergraduateStudent_advisor
UndergraduateStudent.UndergraduateStudent_advisor_4786
UndergraduateStudent.UndergraduateStudent_advisor_4787
UndergraduateStudent.UndergraduateStudent_name
UndergraduateStudent_takesCourse.UndergraduateStudent_PK
UndergraduateStudent_takesCourse.UndergraduateStudent_tak...
```

Group By:

```
UndergraduateStudent.UndergraduateStudent_memberOf
```

Figure 6: DataVision Query Processing

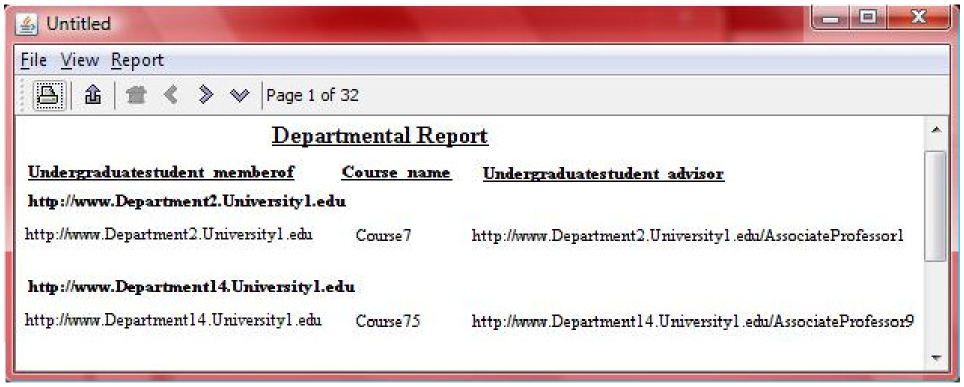


```
select
UndergraduateStudent.UndergraduateStudent_memberOf,
Course.Course_name,
UndergraduateStudent.UndergraduateStudent_advisor
from UndergraduateStudent_takesCourse, Course,
UndergraduateStudent
where (UndergraduateStudent.UndergraduateStudent_PK =
UndergraduateStudent_takesCourse.UndergraduateStudent
_PK) and
(UndergraduateStudent_takesCourse.UndergraduateStuden
t_takesCourse = Course.Course_PK) and (course_name
like 'Course%7%') group by
UndergraduateStudent.UndergraduateStudent_memberOf
```

```
SPARQL Query is:

SELECT  ?subject2 ?subject1
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_MEMBEROF
?COURSE_COURSE_NAME
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_ADVISOR
WHERE {  ?subject2 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#UndergraduateStudent> .  ?subject1 <http://www.w3.org/1999/02/22-
rdf-syntax-ns#type> <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#Course> .  ?subject2 <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#takesCourse>
?UNDERGRADUATESTUDENT_TAKESCOURSE_UNDERGRADUATESTUDE
NT_TAKESCOURSE .  ?subject1 <http://www.lehigh.edu/~zhp2/2004/0401/univ-
bench.owl#name> ?COURSE_COURSE_NAME .  ?subject2
<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#memberOf>
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_MEMBEROF .
?subject2 <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#advisor>
?UNDERGRADUATESTUDENT_UNDERGRADUATESTUDENT_ADVISOR .
FILTER (  (
?UNDERGRADUATESTUDENT_TAKESCOURSE_UNDERGRADUATESTUDE
NT_TAKESCOURSE  =  ?subject1 ) && ( regex( ?COURSE_COURSE_NAME,
"^.*Course7")) ) }
```

Figure 7: SQL-to-SPARQL Conversion



Departmental Report

Undergraduatestudent_memberof	Course_name	Undergraduatestudent_advisor
http://www.Department2.University1.edu		
http://www.Department2.University1.edu	Course7	http://www.Department2.University1.edu/AssociateProfessor1
http://www.Department14.University1.edu		
http://www.Department14.University1.edu	Course75	http://www.Department14.University1.edu/AssociateProfessor9

Figure 8: Tabular Results as seen through DataVision

In order to compare the performance of queries executed through the virtual relational schema offered by R2D against the query performance achieved through RDF visualization tools, XML files corresponding to the LUBM dataset were generated for RDF stores of various sizes and a selection of four queries were run using R2D and Allegrograph's Gruff. These queries were selected at random from the set of LUBM Benchmark SPARQL queries and their equivalent SQL versions were executed using R2D. Figure 9 displays the response times of each of the queries as the sizes of the databases vary.

As can be observed, R2D's performance is far superior to the existing direct RDF visualization. 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 the SQL-to-SPARQL conversion (SQL-to-SPARQL Translation Algorithm) is negligible and nearly constant. Thus, R2D does not add any overheads to the SPARQL query performance.

SQL queries issued against relational databases created by physically duplicating RDF data may possibly exhibit superior performance than their SPARQL equivalents since refined performance optimization options (such as indexes, mature query optimizers, etc.) have been at the disposal of relational databases for many decades now.

Further, for each row of the RDBMS with 'n' columns, there are 'n' triple tuples in the corresponding RDF Store. Thus, the RDBMS equivalent of the RDF Stores generally has a fraction of the data in the RDF Stores which could be yet another contributor to better RDBMS response times than the RDF data store. However, this improved performance comes at the expense of additional disk space that is required due to duplication of data into the RDBMS, and additional system resources and human effort required to ensure that the duplicated data is kept synchronized with the original RDF store. 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 the vast assortment of visualization tools that are readily available without having to "reinvent the wheel" for RDF stores or duplicate/synchronize RDF data. 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.

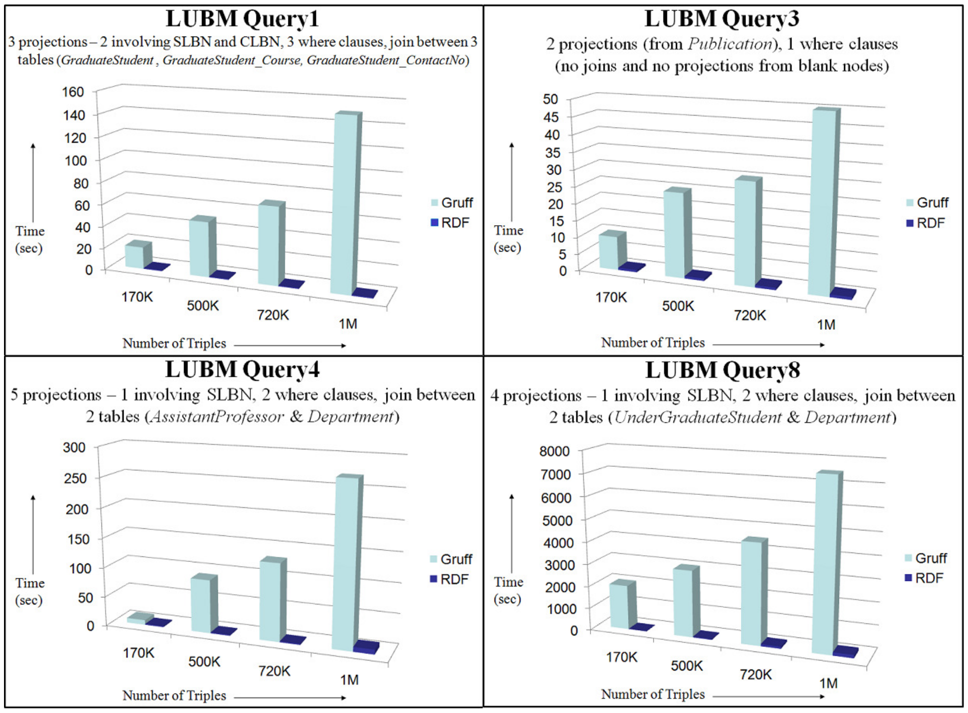


Figure 9: Response times for Selected LUBM Queries

6. CONCLUSION and FUTURE WORK

The R2D framework presented in this paper was motivated by a dearth in the number and variety of data modeling, management, and visualization tools for RDF graph data. Though there are a several ongoing research efforts that attempt to address these deficiencies, most of the efforts involve either the painstaking process of creating new tools or the uneconomical alternative of duplicating data into existing relational stores raising a fresh crop of concerns such as resource wastage and synchronization issues. The chief goal of R2D is to bridge the gap between RDF data sources and the relational model in order to continue to leverage the benefits offered by existing traditional tools without any customization for RDF. A JDBC interface aimed at accomplishing this goal through a mapping between RDF Graph constructs and their equivalent relational counterparts was presented. A detailed description of the mapping constructs, the system architecture, and the modus operandi of the proposed system was discussed along with in depth discussion on the algorithms comprising the R2D framework. 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 R2D include improvisation of the normalization process for mixed blank nodes in order to arrive at better and more appropriate tables corresponding to such blank nodes. Enhancements to the S QL-to-SPARQL translation algorithm that would enable the handling of nested and correlated sub-queries are other aspects that are being explored.

7. REFERENCES

[1] Wikipedia http://en.wikipedia.org/wiki/Semantic_Web

[2] RDF Primer. W3C Recommendation. Feb, 2004. http://www.w3.org/TR/rdf-primer/

[3] Tauberer, J. What is RDF. July, 2006. http://www.xml.com/pub/a/2001/01/24/rdf.html

[4] Muys, A.: Building an Enterprise-Scale Database for RDF Data. http://www.netymon.com/papers/muysa06buildforrdf.pdf (2006)

[5] Teswanich, W., Chittayasothorn, S.: A Transformation of RDF Documents and Schemas to Relational Databases. In: IEEE PacificRim Conferences on Communications, Computers, and Signal Processing, pp. 38-41(2007)

[6] Hendler, J.: RDF Due Diligence. http://civicactions.com/blog/rdf_due_diligence. (2006)

[7] Ramanujam, S., Gupta, A., Khan, L., Seida, S., and Thuraisingham, B.: R2D:Extracting Relational Structure from RDF Stores. In: International Conference on Web Intelligence, 2009, in press.

[8] Ramanujam, S., Gupta, A., Khan, L., Seida, S., and Thuraisingham, B.: R2D:A Bridge between the Semantic Web and Relational Visualization Tools. In: International Conference on Semantic Computing, 2009, in press.

[9] Pan, Z., and Heflin, J. DLDB: Extending Relational Databases to Support Semantic Web Queries. In Practical and Scalable Semantic Systems, 2003.

[10] Bizer, C., Cyganiak, R., Garbers, J., and Maresch, O. The D2RQ Platform. http://www4.wiwiss.fu-berlin.de/bizer/d2rq/

[11] Erling, O., and Mikhailov, I.: RDF Support in the Virtuoso DBMS, 1st Conference on Social Semantic Web, 2007 pp.1617-5468.

[12] Han, L., Finin, T., Parr, C., Sachs, J., and Joshi, A. RDF123: From Spreadsheets to RDF. International Semantic Web Conference, LNCS 5318, 2008, 451-466.

[13] S.Auer, S.Dietzold, J.Lehmann, S.Hellmann, and D.Aumueller, “Triplify – Light-Weight Linked Data Publication from Relational Databases”, 18th International World Wide Web Conference, 2009, pp. 621-630.

[14] Perez de Laborda, C., and Conrad, S. Bringing Relational Data into the Semantic Web using SPARQL and Relational.OWL. 22nd International Conference on Data Engineering Workshops, 2006.

[15] An, Y., Borgida, A., and Mylopoulos, J. Refining Semantic Mappings from Relational Tables to Ontologies. Second International Workshop on Semantic Web and Databases, 2004, 84-90.

[16] An, Y., Borgida, A., and Mylopoulos, J. Discovering the Semantics of Relational Tables through Mappings. Journal on Data Semantics, VII, 2006, 1-32.

[17] Harris, S. SPARQL Query Processing with Conventional Relational Database Systems. International Workshop on Scalable Semantic Web Knowledge Base Systems, 2005.

[18] 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, 2006.

[19] Chen, H., Wu, Z., Wang, H., and Mao, Y. RDF/RDFS-based Relational Database Integration. 22nd International Conference on Data Engineering, 2006.

[20] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. An Efficient SQL-based RDF Querying Scheme. VLDB, 2005.

[21] Yan, Y., Wang, C., Zhou, A., Qian, W., Ma, L., and Pan, Y. Efficiently Querying RDF Data in Triple Stores. 17th International Conference on World Wide Web, 2008, 1053-1054.

[22] Jena – A Semantic Web Framework for Java. http://jena.sourceforge.net/index.html

[23] DataVision. The Open Source Report Writer. http://datavision.sourceforge.net/

[24] GRUFF: A Grapher-Based Triple-Store Browser for Allegrograph. http://agraph.franz.com/gruff/

[25] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics 3(2), 2005, pp158-182.

Storage and Retrieval of Large RDF Graph Using Hadoop and MapReduce

Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani Thuraisingham

University of Texas at Dallas, Dallas TX 75080, USA

Abstract. Handling huge amount of data scalably is a matter of concern for a long time. Same is true for semantic web data. Current semantic web frameworks lack this ability. In this paper, we describe a framework that we built using Hadoop[1] to store and retrieve large number of RDF[2] triples. We describe our schema to store RDF data in Hadoop Distribute File System. We also present our algorithms to answer a SPARQL[3] query. We make use of Hadoop's MapReduce framework to actually answer the queries. Our results reveal that we can store huge amount of semantic web data in Hadoop clusters built mostly by cheap commodity class hardware and still can answer queries fast enough. We conclude that ours is a scalable framework, able to handle large amount of RDF data efficiently.

1 Introduction

Scalibility is a major issue in IT world. Basically what it means is that a system can handle addition of large number of users, data, tasks etc. without affecting its performance significantly. Designing a scalable system is not a trivial task. This also applies to systems handling large data sets. Semantic web data repositories are no exception to that. Storing huge number of RDF triples and the ability to efficiently query them is a challenging problem which is yet to be solved. Trillions of triples requiring peta bytes of disk space is not a distant possibility any more. Researchers are already working on billions of triples[13, 2]. Competitions are being organized to encourage researchers to build efficient repositories[4].

Distributed systems are always good candidates as a solution to scalibility problems. Distributed databases are widely used to handle large amount of relational data. However, a distributed repository for RDF data is yet to be built. The challenges and technologies needed to build such a system for semantic web data are unfolding as researchers move along that path. For building a distributed repository for RDF data, one option may be to use the readily available distributed database systems for relational databases and come up with

[1] http://hadoop.apache.org
[2] http://www.w3.org/RDF
[3] http://www.w3.org/TR/rdf-sparql-query
[4] http://challenge.semanticweb.org

a relational schema to store RDF data. Researchers are already exploring the optimal relational schema for that purpose [9]. Another option can be to build a distributed system from ground up. The advantage of this approach is that a system optimized to handle only RDF data can be built in this way. The disadvantage is that in many cases we have to reinvent the wheel. But instead of building the system from scratch, we can take a generic distributed storage system and build a sematic web repository on top of that.

Such a system is Hadoop. It is a distributed file system where files can be saved with replication. It provides high fault tolerance and reliability. Moreover, it provides an implementation of MapReduce[8] programming model. MapReduce is a functional programming model which is suitable for processing large amount of data in parallel. In this programming paradigm, MapReduce processes are run on independant chunks of data making parallelization easier.

Current semantic web frameworks like Jena[5] do not scale well. These frameworks run on single machine and hence cannot handle huge amount of triples. For example, we could load only 10 million triples in a Jena in-memory model running in a machine having 2 GB of main memory. In this paper, we describe our work with RDF data and Hadoop. We devise a schema to store RDF data in Hadoop. In a preprocessing stage, we process RDF data and put it in text files in the distributed file system according to our schema. We chose Lehigh University Benchmark[10] (LUBM) data generator to generate our data. We have a retrieval mechanism by MapReduce programming. We find that for many queries, one MapReduce job is not enough. We need to have an algorithm to determine how many jobs are needed for a given query. We devise such an algorithm which not only determines the number of jobs but also their sequence and inputs. We run all the LUBM benchmark queries. We run them on different sizes of data sets starting from 100 million triples to 1 billion triples.

In the work we have done, our contributions are as follows: first, a storage scheme to store RDF data in plain text files in Hadoop distributed file system, and next a scalable algorithm to determine the details of MapReduce jobs needed to answer a SPARQL query.

The remainder of this paper is organized as follows: in section 2 we discuss works done by other researchers and our novelties using MapReduce for large amount of data. In section 3 we discuss the architecture of our system. We describe our data storage system in section 4. We discuss how we answer a SPARQL query in section 5. We present the results of our experiments in section 6. Finally we draw some conclusions and discuss probable areas which we have identified for improvement in future in section 7.

2 Related Works

MapReduce is an evolving technology now. The technology has been well received by the community which handles large amount of data. Google uses it for web indexing, data storage, social networking [4]. It is being used to scale up classifiers for mining peta-bytes of data [5]. Data mining algorithms are being rewritten in different forms to take the advantage of MapReduce technology [6]. Other areas

[5] http://jena.sourceforge.net

where this technology is successfully being used are simulation [3]. Hadoop is an Apache project which is an open source implementation of Google's MapReduce technology. It is being used to handle large amount of data for quite some time now. Yahoo is using Hadoop's scale-out for search and information retrieval [1].

The closest match to what we have done is the BioMANTA[6] project. Researchers of that project have done some work regarding large RDF data storage in Hadoop. They proposed extensions to RDF Molecules[7] and implemented a MapReduce based Molecule store [2]. They used MapReduce to answer the queries. They have queried maximum 4 million triples. Our work differs in the following ways: first, we start with 100 million triples and queried 1 billion triple at the maximum. Second, we have devised a storage schema using files which is tailored to improve query execution performace for RDF data. We store RDF triples in files based on the predicate of the triple and the type of the object. Third, we have a query rewriting algorithm for SPARQL queries which leverages our storage schema and can convert a query to an equivalent shorter one under some conditions. Finally, we also have an algorithm to determine the jobs needed to answer a query. We can determine the input files of a job and the order in which they should be run. To the best of our knowledge, we are the first ones to come up with storage schema for RDF data with flat files, a query rewritting algorithm which takes advantage of the schema and a MapReduce job determination algorithm to answer a SPARQL query.

3 Proposed Architecture

Our architecture consists of several software components. We make use of Jena Semantic Web Framework in data preprocessing step and Pellet OWL Reasoner[7] in query execution. The architecture is shown in figure 1. The left column of the figure depicts preprocessing steps for the data and the right column shows the steps to answer a query.

We have three components for data generation and preprocessing. The LUBM [10] data generator creates data in RDF/XML serialization format. We take this data and convert it to N-Triples serialization format using our N-Triple Converter component. This component uses Jena framework to convert the data. The Predicate Based File Splitter takes the converted data and splits it into predicate files. The predicate based files are then fed into the Object-Type Based File Splitter which split the predicate files to smaller files based of type of objects. These steps are described in section 4.2. The output of the last component are then put into HDFS[8].

Our MapReduce framework has three sub-components in it. It takes SPARQL query from the user passes it to Job Decider and Input Selector. This component decides how many jobs are needed and selects the input files and passes the information to the Job Handler component which submits corresponding jobs to Hadoop. It then relays the query answer from Hadoop to the user. To answer queries that require inferencing, we use Pellet OWL Reasoner.

[6] http://www.itee.uq.edu.au/ eresearch/projects/biomanta
[7] http://clarkparsia.com/pellet
[8] http://hadoop.apache.org/core/docs/r0.18.3/hdfs_design.html

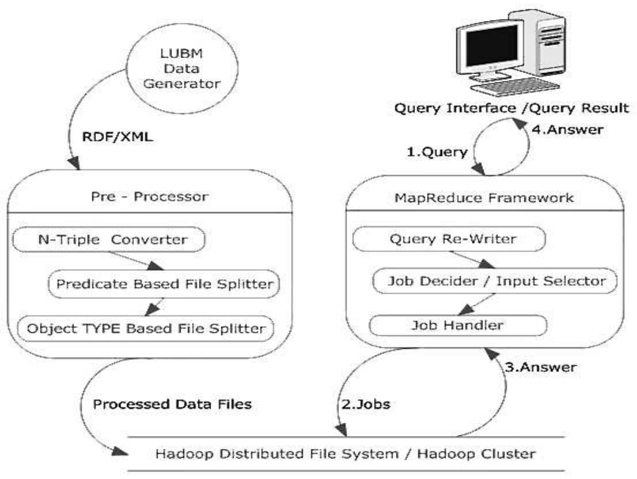

LUBM Data Generator
RDF/XML
Pre - Processor
N-Triple Converter
Predicate Based File Splitter
Object TYPE Based File Splitter
Query Interface /Query Result
1.Query
4.Answer
MapReduce Framework
Query Re-Writer
Job Decider / Input Selector
Job Handler
3.Answer
Processed Data Files
2.Jobs
Hadoop Distributed File System / Hadoop Cluster

Fig. 1. The System Architecture

4 Data Preprocessing

For our experiments, we use LUBM[10] dataset. LUBM is a benchmark data set designed to enable researchers evaluate their semantic web repository perfomances [11]. The data set has a data generator which generates data about a user specified number of universities. It has 14 benchmark queries. Researchers have used LUBM data sets to compare their performances with other semantic web repositories [14, 13, 12].

4.1 Data Generation and Storage

The LUBM data generator generates data in RDF/XML serialization format. This format is not suitable for our purpose because we store data in HDFS in flat files. If the data is in RDF/XML format then to retrieve even one triple we need to parse the whole file and also it is not suitable as an input for a MapReduce job. Instead we choose N-Triple to store the data in because we have a complete triple in one line of a file which is very convenient to use with a MapReduce job. We convert the data to N-Triple format and apply certain processing steps on it to get to our intended data format. These steps are described in following sections.

Hadoop is basically a distributed file system. It breaks large files into small blocks having default size 64 MB. For improved fault tolerance, Hadoop replicates these blocks. Hadoop does not cache any file for MapReduce jobs. The replication and no-cache policy warrants us to use optimal storage policy for storing RDF data.

4.2 File Organization

In HDFS a file takes space replication factor times its size. As RDF is text data, it takes a lot space in HDFS to store a file. To minimize the amount of space, we replace the common prefixes in URIs with some much smaller prefix strings. We keep track of this prefix strings in a separate prefix file. This reduces the space required by the data by a significant amount.

As there is no caching in Hadoop, each SPARQL query needs reading files from HDFS. Reading directly from disk always have a high latency. To reduce the execution time of a SPARQL query, we came up with an organization of files which provides us with the capability to determine the files needed to search in for a SPARQL query. The files usually constitute a fraction of the entire data set and thus making the query execution much faster.

We do not store the data in a single file because in Hadoop file is the smallest unit of input to a MapReduce job. If we have all the data in one file then the whole file will be input to MapReduce jobs for each query. Instead we divide the data in multiple steps. **Predicate Split (PS)**: in the first step, we divide the data according to the predicates. In real world RDF data sets, the number of distinct predicates are just more than 10 or 20 [14]. This division immediately enables us to cut down the search space for any SPARQL query which does not have a variable[9] predicate. For such a query, we can just pick a file for each predicate and run the query on those files only. For simplicity, we name the files with predicates e.g. all the triples containing a predicate *p1:pred* goes to a file named *p1-pred*. However, in case we have a variable predicate in a triple pattern[10] and if we cannot determine the type of the object, we have to consider all files. If we can determine the type of the object then we consider all files having that type of object. We discuss more on this in section 5.1.

Predicate Object Split (POS): In the next step, we work with the type information of objects. The *rdf_type* file is first divided into as many files as the number of distinct objects the *rdf:type* predicate has. For example, if in the ontology, the leaves of the class hierarchy are $c_1, c_2, ..., c_n$ then we will create files for each of these leaves and the file names will be like *rdf-type_*c_1*, rdf-type_*c_2*, ... , rdf-type_*c_n. Please note that the values $c_1, c_2, ..., c_n$ are no longer needed to be stored inside the file as they can be easily retrieved from the file name. This further reduces the amount of space needed to store the data. For each distinct object values of the predicate *rdf:type* we get a file like this.

We divide other predicate files according to the type of the objects. Not all the objects are URIs, some are literals. The literals remain in the file named by the predicate i.e. no further processing is required for them. The objects move into their respective file named as *predicate_type*. For example, if a triple has the predicate p and the type of the object is c_i, then the subject and object appears in one line in the file p_c_i. To do this division we need to join a predicate file with the *rdf-type* files. Queries run much faster with POS schema than PS schema. In one occasion, we could reduce time for query 2 in 1000 universities dataset from 9 hour and 51 minutes to only 10 minutes, a huge improvement.

Table 1 shows the size gain we get at each step for data of 1000 universities. LUBM generator generates files of total 24 GB size. After splitting the data according to predicates the size drastically comes down to only 7.1 GB which is

[9] http://www.w3.org/TR/rdf-sparql-query/#sparqlQueryVariables
[10] http://www.w3.org/TR/rdf-sparql-query/#sparqlTriplePatterns

Step	Files	Size (GB)	Space Gain
N-Triples	20020	24	
PS	18	7.1	70.42%
POS	18	6.6	7.04%

Table 1. Data size at various steps for 1000 universities

a 70.42% gain. This happens because of the absence of predicate columns and also the prefix substitution. In the final step, we again gain 7.04% space as the splitted *rdf-type* files no longer has the object column.

Listing 1.1. LUBM Query 1

```
SELECT ?X WHERE {
?X rdf:type ub:GraduateStudent .
?X ub:takesCourse <http://www.D0.U0.edu/GC0> }
```

We observed that all the LUBM SPARQL queries use the type information heavily. We also observed that there are a large number of triples with *rdf:type* predicates. These predicates are used in joins in all queries. Hence, the size of the join output is very large. To reduce the join output size and also the query execution time, we divided the *rdf-type* file according to the value of objects in the triples.

For example, listing 1.1 shows LUBM query 1 which has the *rdf:type* predicate in first its first triple patterns in line 4. Without the type split the input files for this query would be *rdf-type* and *ub-takesCourse*. But with the type split, we could use *rdf-type_GraduateStudent* and *ub-takesCourse* as the input instead of *rdf-type* and *ub-takesCourse*. The file *rdf-type_GraduateStudent* is significantly smaller than the file *rdf-type*. Thus with the type split, we are reducing the input size, hence reducing the join outputs. This will make the query execution much faster.

5 MapReduce Framework

Our MapReduce Framework is where we answer queries. The challenges we meet to answer a SPARQL query are as follows: first to determine the number of jobs needed to answer a query, second to minimize the size of intermediate files so that data copying and network data transfer is reduced and finally to determine number of reducers. We run one or more MapReduce jobs to answer one query. The following section gives a brief discussion about Hadoop MapReduce.

In Hadoop, the unit of computation is called a job. A user can submit a job to Hadoop JobTracker which is responsible for running a job. In each MapReduce job, there are two phases: Map and Reduce. In the Map phase the map method takes a pair of input key-value pair and may output zero or more key-value pairs. In the Reduce phase, the values for each key are grouped together in a collection and a key and iterator to values pair is passed to the reduce method. It can also output zero or more key-value paires.

When a job is submitted to Hadoop JobTracker, Hadoop creates map processes preferrably near the input data in the cluster. These map processes cannot talk to each other and work independently. Same goes for reduce processes. This lack of communication has the advantage of speed and simplicity. But the disadvantage is in one job we cannot perform all the joins necessary for a SPARQL query without this communication. To overcome this problem, we must have more than one joins for the queries where one job cannot do all the joins without communication between map processes. We write the output of a job to an intermediate file. This intermediate file is used as input to the subsequent job. The algorithm to determine the joins done in a job is described in section 5.2.

5.1 Input Files Selection

Before determining the jobs, we select the files that need to be inputted to the jobs. We take a query submitted by the user and iterate over the triple patterns. In a triple pattern, if the predicate is variable then we select all the files as input to the jobs and terminate the iteration. If the predicate is a variable but has a type information associated to it, then we select all predicate files having object of that file and add them to the input file set. If the predicate is concrete but has no type information, we add all files for the predicate to the input set. If it has a type information associated with it, we add the predicate file which has objects of that type to the input set.

If a type associated with a predicate is not a leaf in the ontology tree, we add all subclasses which are leaves in the subtree rooted at the type node in the ontology tree.

5.2 The ***DetermineJobs*** Algorithm

To answer a SPARQL query by MapReduce jobs, we may need more than one job. It is because we cannot handle all the joins in one job because of the way Hadoop runs its map and reduce processes. Those processes have no inter process communication and they work on idependent chunks of data. Hence, processing a piece of data cannot be dependent on the outcome of any other piece of data which is essential to do joins. This is why we might need more than one job to answer a query. Each job except the first one depends on the output of its previous job.

We devised Algorithm 1 which determines the number of jobs needed to answer a SPARQL query. It determines which joins are handled in which job and the sequence of the jobs. For a query Q we build a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. For each triple pattern in the query Q we build a vertex v which makes up the set V. Hence $|V|$ is equal to the number of triple patterns in Q. We put an edge e between v_i and v_j, where $i \neq j$, if and only if their corresponding triple patterns share at least one variable. We label the edge e with all the variable names that were shared between v_i and v_j. These edges make up the set E. Each edge represents as many joins as the number of variables it has in its label. Hence, total number of joins present in the graph is the total number of variables mentioned in the labels of all edges. An example illustrates it better. We have chosen LUBM [10] query 12 for that purpose. Listing 1.2 shows the query.

Listing 1.2. LUBM Query 12

```
SELECT ?X WHERE {
?X rdf:type ub:Chair .
?Y rdf:type ub:Department .
?X ub:worksFor ?Y .
?Y ub:subOrganizationOf <http://www.University0.edu> }
```

The graph we build at first for the query is shown in figure 2. The nodes are numbered in the order they appear in the query.

Algorithm 1 DETERMINEJOBS(*Query q*)

Require: A Query object returned by RewriteQuery algorithm.
Ensure: The number of jobs and their details needed to answer the query.

```
1: $jobs \leftarrow \phi$
2: $graph \leftarrow makeGraphFromQuery(q)$
3: $joins\_left \leftarrow calculateJoins(graph)$
4: while $joins\_left \neq 0$ do
5:     $variables \leftarrow getVariables(graph)$
6:     $job \leftarrow createNewJob()$
7:     for $i \leftarrow 1$ to $|variables|$ do
8:         $v \leftarrow variables[i]$
9:         $v.nodes \leftarrow getMaximumVisitableNodes(v, graph)$
10:        $v.joins \leftarrow getJoins(v.nodes, graph)$
11:    end for
12:    $sortVariablesByNumberOfJoins(variables)$
13:    for $i \leftarrow 0$ to $|variables|$ do
14:        if $|v.joins| \neq 0$ then
15:            $job.addVariable(v)$
16:            $jobs\_left \leftarrow jobs\_left - |v.joins|$
17:            for $j \leftarrow i + 1$ to $|variables|$ do
18:                $adjustNodesAndJoins(variables[j], v.nodes)$
19:            end for
20:            $mergeNodes(graph, v.nodes)$
21:        end if
22:    end for
23:    $jobs \leftarrow jobs \cup job$
24: end while
25: return $jobs$
```

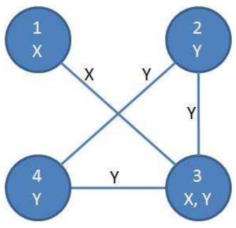


Fig. 2. Graph for Query 12 in Iteration 1

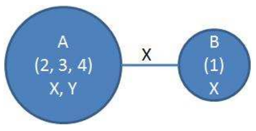


Fig. 3. Graph for Query 12 in Iteration 2

In figure 2, each node in the figure has a node number in the first line and variables it has in the following line. Nodes 1 and 3 share the variable X hence there is an edge between them having the label X. Similarly, nodes 2, 3 and 4 have edges between them because they share the variable Y. The graph has total 4 joins.

Algorithm 1 is iterative. It takes a *Query* object as its input, initializes the *jobs* set (line 1), builds the graph shown in figure 2 before entering first iteration (line 2). It also calculates the number of jobs left (line 3). It enters the loop in line 4 if at least one job is left. At the beginning of the loop it retrieves the set of variables (line 5) and creates a new empty job (line 6). Then it iterates over the variable (line 7 and 8), lists the maximum number of nodes it can visit by edges having the variable in its label (lines 9). It also lists the number of joins that exist among those nodes (line 10). For example, for variable Y we can visit nodes 2, 3 and 4. The joins these nodes have are 2-3, 3-4 and 4-2. The information it collects for each variable is shown in table 2.

Variable	Nodes	Joins	||Joins||
Y	2, 3, 4	2-3, 3-4, 4-2	3
X	1, 2	1-2	1

Table 2. Iteration 1 Calculations

	Variable	Nodes	Joins	||Joins||
✓	Y	2, 3, 4	2-3, 3-4, 4-2	3
	X	1		0

Table 3. Iteration 1 - After choosing X

Variable	Nodes	Joins	Total Joins
X	1, 2	1-2	1

Table 4. Iteration 2 Calculations

It then sorts the variables in descending order according to the number of joins they cover (line 12). In this example, the sort output is the same as table 2. Then, in greedy fashion, it iterates over the variables and chooses a variables if the variable covers at least one join (line 13 and 14). In each iteration, after it chooses a variable, it eliminates all the nodes it covers from subsequent variable entries (lines 17 to 19). It then calculates the number of joins still left in the graph (line 16). For example, once the algorithm chooses the variable Y, the nodes and joins for X becomes like table 3.

It also merges the nodes visited by the chosen variable in the graph (line 20). Hence, after choosing Y it will not choose X as it does not cover any join any more. Here the inner loop terminates. The joins it picked are the joins that will be done in a job. The algorithm then checks whether any join is not picked (line 4). If such is the case, then more jobs are needed and so the algorithm goes to the next iteration.

At the beginning of the subsequent iteration it again builds a graph from the graph of the previous iteration but this time the nodes which took part in joins by one variable will be collapsed into a single node. For our example, nodes 2, 3 and 4 took part in joins by Y. So they will collapse and form a single node. For clarity, we name this collapsed node as A and the remaining node 1 of the graph in figure 2 as B. The new graph we get like this is shown in figure 3. The graph has total 1 join. We have listed the nodes which were collapsed in braces.

After building the graph, the algorithm moves on to list the maximum number of nodes, joins and total number of joins each variable covers. This is shown in table 4. The algorithm chooses X and that covers all the joins of the graph. The algorithm determines that no more job is needed and returns the job collection.

5.3 Performing *Join*

In this section, we discuss how we implement joins needed to answer SPARQL queries using MapReduce framework of Hadoop. Algorithm 1 determines the number of job required to answer a query. It returns an ordered set of jobs. Each job has associated input information. The Job Handler component of our MapReduce framework runs the jobs in the sequence they apprear in the ordered set. The output file of one job is the input of the next one. The output file of the last job has the answer to the query.

Listing 1.1 shows LUBM query 1 which we will use to illustrate the way we do join using map and reduce methods. The query has two triple patterns and one join between them by the variable X. Our input selection algorithm selects the file *type_GraduateStudent* for the triple pattern of line 4 and for the triple pattern of line 5 all files having the prefix *takesCourse* as the input to the only job needed to answer the query.

In the *map* phase, we first tokenize the value which is actually a line of the input file. Then we check the input file name and if input is from *type_GraduateStudent*, we output a key value pair having the subject URI as the key and a flag string *GS* as the value. The value serves as a flag to indicate that the key is of type *GraduateStudent*. The subject URI is the first token returned by the tokenizer. If the input is not from that file then it must be from a file having the prefix *takesCourse*. We then retrieve the subject and object from the input line by the tokenizer and then check whether the object value is "http://www.D0.U0.edu/GC0". If that is the case, we output a key value pair having the subject URI as the key and the object value as the value.

In the *reduce* phase, Hadoop groups all the values for a single key and for each key provides the key and an iterator to the values collection. Using the iterator we simply count the number of values in the values collection. If the count is two then we know that the key is a URI to a graduate student who took the course "http://www.D0.U0.edu/GC0". It is because only if a URI satisfies both conditions in the *map* phase, it can appear as a key in two output key value pairs in that method.

6 Results

Due to space limitations we choose to report runtimes of six LUBM queries which we ran in a cluster of 10 nodes with POS schema. Each node had the same configuration: Pentium IV 2.80 GHz processor, 4 GB main memory and 640 GB disk space. The results we found are shown in table 5.

Universities	Triples (million)	Query1	Query2	Query4	Query9	Query12	Query13
1000	110	66.313	146.86	197.719	304.87	79.749	198.502
2000	220	87.542	216.127	303.185	532.982	95.633	272.521
3000	330	115.171	307.752	451.147	708.857	100.091	344.535
4000	440	129.696	393.781	608.732	892.727	115.104	422.235
5000	550	159.85	463.344	754.829	1129.543	132.043	503.377
6000	660	177.423	543.677	892.383	1359.536	150.83	544.383
7000	770	198.033	612.511	1067.289	1613.622	178.468	640.486
8000	880	215.356	673.0	1174.018	1855.5127	184.434	736.189
9000	990	229.18	727.596	1488.586	2098.913	214.575	821.459
10000	1100	273.085	850.503	1581.963	2508.93	286.612	864.722

Table 5. Query Runtimes

Table 5 has query answering times in seconds. The number of triples are rounded down to millions. As expected, as the number of triples increased, the time to answer a query also increased. Query 1 is simple and requires only one join. We can see that it took the least amount of time among all the queries. Query 2 is one of the two queries having most number of triple patterns. We can observe that even though it has three times more triple patterns it does not take thrice the time of query 1 answering time because of our storage schema. Query 4 has one less triple pattern than query 2 but it requires inferencing to bind 1 triple pattern. As we determine inferred relations on the fly, queries requiring inferencing takes longer times in our framework. Query 9 and 12 also require inferencing and query 13 has an inverse property in one of its triple patterns.

We can see that the ratio between the size of two datasets and the ratio between the query answering times for any query are not the same. The increase in time to answer a query is not proportionate to the increase in size of datasets. In fact, the increase in time is always less. For example, there are ten times triples in the dataset of universities 10000 than universities 1000 but for query 1 the time only increases by 4.12 times and for query 9 by 8.23 times. The later one is the highest increase in time which is still less than the increase in the size of the datasets. Due to space limitations, we do not report query runtimes with PS schema here. We observed that PS schema is much slower than POS schema.

We also ran few queries in a small cluster of 4 nodes where each node has the following configuration: Pentium IV 2.80 GHz processor, 1 GB main memory and 80 GB disk space. We ran queries for 1000 universities. Table 6 shows the runtimes in seconds where we can see that the bigger cluster with superior configuration is significantly faster than the smaller one.

Query	4-node cluster runtime	10-node cluster runtime	% Improvement
1	83.225	66.313	25.5
2	272.726	146.86	85.7
12	92.485	79.749	15.97

Table 6. Query Runtimes Comparison

We also ran query 1 and 2 by Jena SDB[11] model for 1000 universities in a machine having 8 GB main memory, 2.80 GHz quad core processor and 1 TB disk space. We found that the queries take 4215.017 and 4882.969 seconds respectively. These are about 63.56 and 33.25 times longer than our runtimes with the larger cluster.

7 Conclusions And Future Works

We have presented a framework solution for handling large amount of RDF data. Our system is a distributed one as it is based on Hadoop. Because of Hadoop our system is highly fault tolerant. It is also readily scalable. To add resource to our system, all one has to do is to add new nodes to the Hadoop cluster. We have proposed a schema to store RDF data in plain text files. Finally, we have proposed an algorithm to determine the jobs necessary to answer a SPARQL

[11] http://jena.hpl.hp.com/wiki/SDB

query. The experiment we ran showed that our system is highly scalable. Not only if we add data we do not decrease performance but also the delay introduced to answer a query does not increase as much as the increment in data size.

We have identified a few items as future work. We will enable Algorithm 1 to handle queries with optional triple pattern, devise a new algorithm to determine the minimum number of jobs, gather and use summary statistics about data and determine the optimal number of reducers for each job.

References

1. Yahoo Research Team: Content, Metadata, and Behavorial Information: Directions for Yahoo! Research IEEE Data Eng. Bull. 10-19 2006
2. Beijing, China Newman, A. and Hunter, J. and Li, Y. F. and Bouton, C. and Davis, M.: A Scale-Out RDF Molecule Store for Distributed Processing of Biomedical Data Semantic Web for Health Care and Life Sciences Workshop WWW 2008
3. Mcnabb, Andrew W. and Monson, Christopher K. and Seppi, Kevin D.: MRPSO: MapReduce particle swarm optimization GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation 177 ACM Press 2007
4. Fay Chang and Jeffrey Dean et al.: Bigtable: A Distributed Storage System for Structured Data OSDI Seventh Symposium on Operating System Design and Implementation November 2006
5. Christopher Moretti and Karsten Steinhaeuser and Douglas Thain and Nitesh V. Chawla: Scaling Up Classifiers to Cloud Computers IEEE ICDM 2008
6. C.-T. Chu and S. K. Kim and Y.-A. Lin and Y. Yu and G. Bradski and A. Y. Ng and K. Olukotun: Map-reduce for machine learning on multicore NIPS 2007
7. Ding, Li and Finin, Tim and Peng, Yun and da Silva, Paulo P. and Mcguinness, Deborah L.: Tracking RDF Graph Provenance using RDF Molecules Proc. of the 4th International Semantic Web Conference (Poster) 2005
8. Dean, Jeffrey and Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation 10-10 2004
9. Abadi, Daniel J. and Marcus, Adam and Madden, Samuel R. and Hollenbach, Kate: Scalable semantic web data management using vertical partitioning VLDB '07: Proceedings of the 33rd international conference on Very large data bases 411–422 2007
10. Y. Guo and Z. Pan and J. Heflin: LUBM: A Benchmark for OWL Knowledge Base Systems Journal of Web Semantics Volume 3 Number 2 158-182 2005
11. Yuanbo Guo and Zhengxiang Pan and Jeff Heflin: An evaluation of knowledge base systems for large OWL datasets In International Semantic Web Conference 274-288 2004
12. Haase, Peter and Wang, Yimin: A decentralized infrastructure for query answering over distributed ontologies SAC '07: Proceedings of the 2007 ACM symposium on Applied computing 1351–1356 2007
13. Rohloff, Kurt and Dean, Mike and Emmons, Ian and Ryder, Dorene and Sumner, John: An Evaluation of Triple-Store Technologies for Large Data Stores On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops 1105-1114 2007
14. Stocker, Markus and Seaborne, Andy and Bernstein, Abraham and Kiefer, Christoph and Reynolds, Dave: SPARQL basic graph pattern optimization using selectivity estimation WWW '08: Proceeding of the 17th international conference on World Wide Web 595-604 2008

Geographically-Typed Semantic Schema Matching

Jeffrey Partyka, Latifur Khan, B. Thuraisingham

Department of Computer Science

University of Texas at Dallas

{jlp072000, lkhan, Bhavani.thuraisingham}@utdallas.edu

ABSTRACT

Resolving semantic heterogeneity across distinct data sources remains a highly relevant problem in the GIS domain requiring innovative solutions. Our approach, called GSim, semantically aligns tables from respective GIS databases by first choosing attributes for comparison. We then examine their instances and calculate a similarity value between them called entropy-based distribution (EBD) by combining two separate methods. Our primary method discerns the geographic types from instances of compared attributes. If successful, EBD is calculated using only this method. GSim further facilitates geographic type matching by replacing missing instance values in attributes via reverse geocoding and applying attribute weighting to quantify the uniqueness of mapped attributes. If geographic type matching is not possible, we then apply a generic schema matching method, independent of the knowledge domain, which employs normalized Google distance. We show the effectiveness of our approach over the traditional N-gram approach across multi-jurisdictional datasets by generating impressive results.

Categories and Subject Descriptors

I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods – *semantic networks, representations (procedural and rule-based)*

General Terms

Algorithms, Experimentation, Measurement, Reliability, Human Factors

Keywords

Schema matching, geographic information systems, gazetteer, reverse geocoding, type extraction, semantic similarity

1. INTRODUCTION

The problem of information integration has experienced a number of manifestations over the last few decades, as solutions such as schema integration[5] and ontology alignment[4] have emerged to solve this problem for relational databases and ontologies, respectively. Whatever the form, the crux of this problem has always been the need to consolidate heterogeneous data sources with potentially different purposes under a single, unified representation. The alignment between different entities, such as tables in different geodatabases and concepts in ontologies, has emerged as one of the keys to determining the ultimate success of the Semantic Web. Furthermore, this problem has taken on special significance in domains such as geographic information systems, where effective approaches to problems as varied as geo-spatial tagging and geographic disambiguation are highly prized. As a result, a tremendous amount of effort has been expended to discover novel information integration strategies.

Some of the more popular approaches have used instance similarity to semantically align data between two distinct data sources[1,2]. Using this approach, we match similarity between tables by first determining pairs of attributes between the tables that are to be compared. Next, for each pair, we examine the respective attributes' instance data and compute a semantic similarity score. Combining the scores from aligned attributes will determine similarity between the tables as a whole.

In this paper we attempt to compare two pairs of GIS data sources using their respective table instances; the first pair contains tables describing similar models of transportation networks over multiple jurisdictions, while the other pair contains tables detailing varying geographic features beyond road networks. The data sources contain large variations in the geographic areas covered, the number of attributes and the number of instances.

To measure instance similarity between compared attributes, we apply an algorithm which we will dub as GSim, and it consists of two distinct approaches. The primary approach determines the geographic types (GT) over the instances associated with compared attributes. This is done by leveraging an external data source known as a gazetteer[12,13]. The gazetteer would be able to match one or more GTs for as many instances as possible. Semantic similarity is calculated by considering the collection of types extracted from instances between the compared attributes. It is based on a quantity known as entropy-based distribution (EBD), which is defined as the ratio of the conditional entropy within each type over a pair of compared attributes with the entropy taken over all types.

Whenever possible, GSim calculates semantic similarity between attributes using GTs alone. However, if too many instances within the compared attributes lack GT information, then GSim resorts to its secondary approach, which uses a generic schema matching algorithm based on a semantic distance measure known as normalized Google distance (NGD)[23]. This method is generic because it is applicable across knowledge domains.

Despite the utility of NGD, solely relying on it to determine semantic similarity is unwise, particularly in the GIS domain. The reason is that a number of situations exist where the instances are determined to be semantically similar due entirely to their close geographic proximity. One such situation is depicted in Section 4.3; GSim's usage of geographic semantic schema matching is justified in that case because two attributes, a city attribute and a county attribute, would match using NGD purely on the basis of their common geographic origins. The dissimilarity of the types of the attributes, therefore, could only be captured by using a GT lookup. On the other hand, the gazetteer does not contain information on every geographic feature, and so realistically, there will be many geodatabases which are unable to return a sufficient amount of GT information. This justifies the GSim's usage of generic semantic schema matching.

The challenges that we will address in this paper are as follows. First, only in the ideal case does the gazetteer match one specific GT for each of the instances. In reality, some instance names, such as "Clinton", are very common, and as a result, the gazetteer is likely to return several GTs. In other words, a 1:N mapping often exists between an instance of an attribute and the GTs which are assigned to it by a gazetteer. Thus, the challenge of handling multiple possible GTs for a given instance is addressed. The second challenge addressed by our research relates to the likely possibility of encountering missing or erroneous instances in attributes of geodatabases. Therefore, when attributes containing these instances are involved in a comparison with another attribute, then it is possible that a less-nuanced approach would compute an incorrect semantic similarity value. However, GSim takes this possibility into account by using the latitude and longitude values associated with an instance and performing reverse geocoding to retrieve the correct instance value. The third challenge addressed by our research is the problem of determining the most uniquely relevant attributes within a particular table. It is possible for two tables to share a high semantic similarity score based on matching attributes which are not relevant to the concept that the tables represent. GSim applies attribute weighting to measure table similarity by placing more weight on the most semantically relevant attributes. This way, the measured EBD value generated for any given table comparison will be based on attributes that represent the essence of the compared tables.

The rest of this paper is organized as follows. In section 2, we discuss an overview of related work. Section 3 states definitions, the problem to be solved and our proposed solution. Section 4 presents in detail the GSim algorithm, detailing both the geographic lookup component as well as the more generic NGD component. In Section 5 we present our results generated with GSim and compared them against those generated using N-grams. Finally, in section 6, we outline our future work.

2. RELATED WORK

In this section, we will first present other work related to schema matching. Second, we present work in the GIS domain making use of a gazetteer. Third, we present work making use of reverse geocoding. Finally, we contrast our work with another approach used to solve the schema matching problem.

The most closely related work in the GIS domain discusses schema and ontology matching for geodatabases and thesauri. Leme, Casanova et al[1] perform schema matching over GIS databases containing data represented by a dialect of OWL. Brauner, Intrator et al[2] perform instance matching over the exported schemas of geographical database Web services and apply their technique over the Geonames and ADL gazetteers. Brauner, Casanova et al[3] leverage instance mapping between distinct terms in feature type thesauri used to classify data in gazetteers, for the facilitation of successful thesaurus migration from one gazetteer to another. Some techniques involving the mapping of GIS ontologies also influenced our work. Most notably, Cruz, Sunna et al[4] describe AgreementMaker, a visual tool that provides a user with the ability to perform mappings between ontologies using a multi-faceted strategy involving automated techniques as well as manual specifications.

A number of schema matching publications [5,6,7,8] tailored to the database community influenced our work. The survey of approaches to automated schema matching by Rahm and Bernstein[5] includes a taxonomy which uses several criteria to categorize matching approaches such as schema and instance based methods, element-level and structure-level methods, and linguistic and constraint-based methods. Dai, Koudas et al. [6] discuss instance-based schema matching using distributions of N-grams among compared attributes. Bohannon et al.[7] investigate contextual schema matching, in which selection conditions and a framework of matching techniques are used to create higher quality mapping between attributes of compared schemas. Warren and Tompa [8] propose an iterative algorithm that deduces the correct sequence of concatenations of column substrings in order to translate from one database to another without the use of a set of training instances.

Within the AI community, a number of works in the schema matching area applied machine learning and statistical methods to learn attribute properties from data and examples. Li and Clifton [9] describe a tool known as SEMINT, which uses neural networks to determine match candidates by matching attributes from similar clusters between attributes in a 1:1 match. Berlin and Motro[10] describe a tool known as Autoplex which uses supervised machine learning techniques for automating the discovery of instance for virtual database systems. Embley et al. [11] explore both 1:1 and m:n schema mapping techniques by applying knowledge obtained from a domain ontology and data frames.

Much work in the GIS community making use of a gazetteer for information lookup influenced also our work. Zhou, Frankowski et al. [12] apply a deterministic, density-based clustering algorithm to semi-automatically discover gazetteers from users' travel data, as well as disambiguate between uninteresting and interesting results from the gazetteer using temporal techniques. Newsam and Yang[13] integrate a gazetteer with high-resolution remote sensed imagery to automate geographic data management more completely, and they also demonstrate how gazetteers can be effectively used as a source of semi-supervised training data for geospatial object modeling. Pouliquen, Steinberger et al.[14] use a gazetteer lookup, as opposed to linguistic analysis, to search through natural language text and produce geographic maps and animations that represent the area referred to in the text.

Some work in the GIS community involving reverse geocoding is related to our research. Zhou and Frankowski[15] evaluate the accuracy of personal place discovery using reverse geocoding and clustering through a set of evaluation metrics and an interactive evaluation framework. Joshi and Luo[16] employ reverse geocoding using location coordinates from image data to obtain nearby points of interest connecting an image with its geographic location. Wilde and Kofahl[17] describe the use of reverse geocoding in retrieving location types as an essential component for a geo-enabled Web browser.

Our paper presents an innovative instance matching algorithm that possesses a number of advantages over the N-gram approach proposed by Dai, Koudas et al., particularly in the GIS domain. An N-gram is a substring of length N consisting of contiguous characters. So for example, if N=2, then the word 'GSim' has N-grams 'GS', 'Si' and 'im'. First, GSim determines GTs for instances via a gazetteer as part of the process of determining an overall semantic similarity value. Because GSim uses domain-specific information to determine the GT for a given instance, it is better equipped than the N-gram approach to solve the information integration problem among geodatabases. Second, GSim can retrieve missing instance values in geodatabases by using associated latlong values to perform reverse geocoding. This ability is not available using solely the N-gram approach. Third, in case the geographic lookup component is unsuccessful, GSim leverages clustering of types for use on distinct keywords found between compared attributes via NGD. This approach is better able to capture the semantics of comparisons between attributes because words contain more implicit semantic information than N-grams. Using words, we can reference external data sources that allow for distance metrics to determine word relatedness. In general, this cannot be done with N-grams because they are usually just parts of words. Fourth, our new instance matching algorithm is flexible enough to allow for different types of semantic distance measures to be used. Treating the semantic distance measure as a pluggable component allows for a wider variety of experiments to be performed on a given instance set, which in turn leads to a better understanding of the kinds of semantic distance measures that best suits a particular type of data. Finally, the use of N-grams for instance similarity between data sources sometimes generates misleading results, especially in cases where data of different languages but similar semantics is being compared, or where there is a lack of shared instances between the compared attributes.

3. PROBLEM STATEMENT AND PROPOSAL

3.1 Definitions

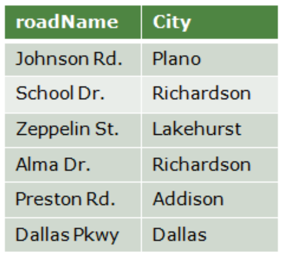
First, we will provide definitions that will assist in defining the problem and describing GSim.

Definition 1 (attribute) *An attribute of a table T, denoted as att(T), is defined as a property of T that further describes it.*

Definition 2 (instance) *An instance x of an attribute att(T) is defined as a data value associated with att(T).*

Definition 3 (keyword) *A keyword k of an instance x associated with attribute att(T) is defined as a semantically relevant word representing a portion of the instance.*

In figure 1 below, the two attributes for the given table are roadName and City, two instances from the roadName attribute are "Johnson Rd." and "School Dr.", and the two keywords associated with the instance "School Dr." are "School" and "Dr.".



roadName	City
Johnson Rd.	Plano
School Dr.	Richardson
Zeppelin St.	Lakehurst
Alma Dr.	Richardson
Preston Rd.	Addison
Dallas Pkwy	Dallas

Figure 1. Sample table containing two attributes and six instances

Definition 4 (type) *A type t associated with attribute att(T) is defined as a class of related entities grouped together.*

We define two kinds of types:

Definition 4a (geographic type (GT)) *A geographic type GT associated with attribute att(T) is defined as a class of instances of att(T) that represent the same geographic feature.*

Definition 4b (non-geographic type (NGT)) *A non-geographic type NGT associated with attribute att(T) is defined as a class of keywords from instances of att(T) that are semantically related to each other.*

Definition 5 (geographic type (GT) vector) *A geographic type vector* $\mathcal{T}_x = \{GT_1, GT_2, \ldots GT_m\}$ *associated with an instance x of attribute att(T) is defined as a set of GTs.*

Definition 6 (geographic weight (GW) vector) *A geographic weight vector* $\mathcal{W}_x = \{w_1, w_2, \ldots w_m\}$ *associated with a GT vector* $\mathcal{T}_x = \{GT_1, GT_2, \ldots GT_m\}$ *for an instance x of attribute att(T) is defined as a list of real numbers between 0 and 1 representing the influence of a GT on the instance.*

Note that for all i, $GT_i \in T_x$ of any instance x has weight $w_i \in W_x$.

Definition 7 (geographic type (GT) set of attribute) *A geographic type set of attribute att(T), denoted* $\mathcal{T}_{att}$ *, is the set of GTs derived from the union of the types from all GT vectors for the instances of att(T).*

Definition 8 (non-geographic type (NGT) set of attribute) *A non-geographic type set of attribute att(T), denoted* $\mathcal{NT}_{att}$*, is the set of NGTs associated with keywords from instances of att(T).*

Definition 9 (geographic type (GT) weight list) *A geographic type weight list* $\mathcal{W}_{att}$ *associated with attribute att(T) is the total type weights for each type in* $\mathcal{T}_{att}$*.*

In figure 2 below, the instances are "Victoria", "Anacortes", "Clinton" and "Edmonds". The GT 'City' contains the instances "Victoria" and "Clinton", The GT vector for "Victoria" = {City, State, Feature} and for "Anacortes", it is = {County}. The GW vector for "Victoria" is {1/3,1/3,1/3}, and for "Anacortes" it is {1}. If these four instances make up the entirety of attribute att, then $\mathcal{T}_{att}$ is {City, State, Feature, County}, and the GT weight list $\mathcal{W}_{att}$ is {5/6, 1/3, 5/6, 2}. The details of the computation of $\mathcal{W}_{att}$ for figure 2 is shown in section 4.1.2.

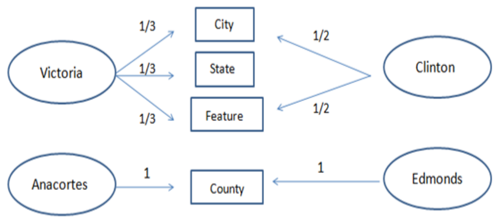


Figure 2. Sample instances of attribute *att* and their respective sets of GTs

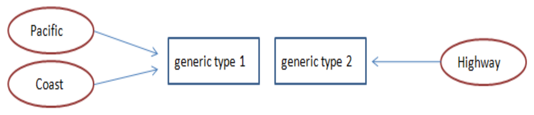


Figure 3. Sample keywords from an instance of attribute att and their respective NGTs

In figure 3 above, given an instance with a value of "Pacific Coast Highway", there are two NGTs named generic type 1 and generic type 2. The NGT set $\mathcal{NT}_{att}$ of attribute *att* that contains this instance would have {generic type 1, generic type 2}, as well as other types from other instances of this attribute.

3.2 Problem Statement

Given two data sources, S_1 and S_2, each of which is composed of a set of tables/relations where $\{T_{11}, T_{12}, T_{13}\ldots T_{1M}\} \in S_1$ and $\{T_{21}, T_{22}, T_{23}\ldots T_{2N}\} \in S_2$, the goal is to determine the semantic similarity between S_1 and S_2. This is done by comparing the respective attribute names and instances between the tables from S_1 and those from S_2. S_1 and S_2 may be derived from any domain. Additionally, S_1 and S_2 may vary in regards to the number of constituent tables, the number of attributes and instances within a given table. With this in mind, an effective data source similarity procedure would be expected to match up tables which describe semantically similar data. Our goal is to quantify this semantic similarity given the instance data for each schema.

3.3 Proposed solution

We present GSim, an instance matching algorithm that generates semantic similarity values between compared attributes in different tables of a geodatabase based on two separate approaches. The primary approach assigns GTs to instances involved in compared attributes within two tables of the geodatabase with the help of a gazetteer. This results in a pair of GT sets, one for each attribute. The semantic similarity between the compared attributes is then computed using EBD over their respective GT sets. However, since gazetteers do not contain information for all geographic features, it is possible that attribute matching via geographic-type extraction will be ineffective, due to lack of information. In this case, we apply a generic schema matching method, applicable over any knowledge domain, that is based on the extraction and clustering of semantically relevant keywords as types based on NGD. Further details describing the GSim in its entirety are described in Section 4.1. It is our intention to clearly show that the use of GSim is better able to capture the true semantics that exist between compared attributes contained within GIS tables as opposed to using N-grams.

It is assumed that we perform 1:1 comparisons between attributes from distinct tables and data sources. After calculating a semantic similarity value between compared attributes, we will repeat the process for all compared attributes between the tables. Next, a final similarity value between the tables is calculated using EBD. EBD is based on a comparison of the conditional entropy of the attributes, given a particular type, with the entropy of the attributes over all types:

$$\text{EBD} = \frac{H(A \mid T)}{H(A)} \qquad (1)$$

In this equation, A is the attribute, coming from either one table or another (since all table comparisons are 1-1), and T stands for the type of the attribute. For geographic matching, T would indicate a GT, such as 'City' or 'County', while for non-geographic matching, T would indicate a given generic type. For more details regarding the usage of EBD and its mathematical derivation, please see our previous work[19].

4. DETAILS OF ALGORITHM

This section describes GSim, our instance similarity algorithm, and its two components. The first, detailed in section 4.1, involves the use of a geographic lookup to determine whether the instances of compared attributes between two tables share similar GTs. If so, then an exact match for those instances is made using only GTs. If not, then section 4.2 describes the second component of GSim, which exclusively relies on a non-geographic measure of semantic similarity between instances of compared attributes. For our purposes, we use NGD as our non-geographic similarity measure. Despite the generalized utility of NGD, there are situations when this approach produces inaccurate results. One example such example is described in section 4.3, and we use this to justify our preference to match instances of compared attributes through GTs alone.

4.1 Overview of GSim

4.1.1 GSim Algorithm

For Algorithm 1 below, the input consists of the attributes $A_1 \in T$ in S_1 and $A_2 \in T'$ in S_2 and gazetteer G. Line 1 initializes $\mathcal{T}_{gaz}$, the set of all GTs recognized by gazetteer G, $\mathcal{T}_{A1UA2}$, the GT vector list for A_1 U A_2, $\mathcal{NT}_{A1UA2}$, the NGT vector list for A_1 U A_2, and $\mathcal{W}_{A1UA2}$, the GW vector list for A_1 U A_2. Lines 2 and 3 extract the distinct instances from A_1 and A_2. Line 4 determines whether semantic similarity can be performed strictly by relying on GTs, or if NGD similarity will be necessary. GT similarity is only possible if a gazetteer is available, and if it contains sufficient GT information about enough of the instances. For our purposes, we established a threshold, t_{min}, which represents the minimum number of instances that contain GT information in G. If GT information can be found for a number of instances greater than or equal to t_{min}, then EBD is calculated using only GTs. This process is initiated in lines 5-9, where line 5 retrieves all available GTs, $\mathcal{T}_{gaz}$, recognized by gazetteer G, lines 6-7 derives a GT vector list

$\mathcal{T}_{A1}$ and its associated GW vector list $\mathcal{W}_{A1,}$ consisting of GT vectors for each instance of A_1 and A_2. Lines 8-9 combine the GT vector lists and GW vector lists, respectively, for A_1 and A_2. If however, in line 4 if geotypingIsPossible() returns false, then we need to rely on a more generic measure like NGD to compute semantic similarity between the compared instances. This is done in line 11. The NGD component of GSim will be described in section 4.2. Line 13 calculates the final EBD value between A_1 and A_2 given the combined type vector lists and weight vector lists of A_1 and A_2, and line 14 returns that EBD value.

Algorithm 1 GSim (A_1, A_2, G)

Input:
- attribute A_1 ε T in S_1 , attribute A_2 ε T' in S_2, gazetteer G

Output: Semantic similarity value between A_1 and A_2 expressed as EBD

```
1: $\mathcal{T}_{gaz} = \Phi$, $\mathcal{T}_{A1 \cup A2} = \Phi$, $N\mathcal{T}_{A1 \cup A2} = \Phi$, $\mathcal{W}_{A1 \cup A2} = \Phi$
2: $IL_1$ = ExtractInstances ($A_1$)
3: $IL_2$ = ExtractInstances ($A_2$)
4: if (geotypingIsPossible (G, $IL_1$, $IL_2$)) {
5:     $\mathcal{T}_{gaz}$ = getGazetteerTypes(G)
6:     ($\mathcal{T}_{A1}$, $\mathcal{W}_{A1}$ ) = lookupGeoTypes($\mathcal{T}_{gaz}$, $IL_1$)
7:     ($\mathcal{T}_{A2}$, $\mathcal{W}_{A2}$ ) = lookupGeoTypes($\mathcal{T}_{gaz}$, $IL_2$)
8:     $\mathcal{T}_{A1 \cup A2} = \mathcal{T}_{A1} \cup \mathcal{T}_{A2}$
9:     $\mathcal{W}_{A1 \cup A2} = \mathcal{W}_{A1} \cup \mathcal{W}_{A2}$
10: } else {
11:     $N\mathcal{T}_{A1 \cup A2}$ = NGDSim($IL_1$, $IL_2$)
12: }//end if
13: EBD[$A_1$][$A_2$] = computeEBD($\mathcal{T}_{A1 \cup A2}$, $\mathcal{W}_{A1 \cup A2}$, $N\mathcal{T}_{A1 \cup A2}$)
14: return EBD[$A_1$][$A_2$]
```

4.1.2 Assigning GTs to Instances

We leverage a gazetteer as a way to help determine the GT of an instance. The gazetteer used for our purposes is Geonames [18], containing information on over 8 million geographic names. The gazetteer classifies locations into different categories, or types. Some examples of GTs include city, county, state and a general feature with several sub-classifications, such as lake, port, school, etc. Instances with more commonplace names are likely to be listed under multiple types in the gazetteer. As a result, a single instance may be associated with a list of GTs = $\{GT_1, GT_2 \ldots GT_n\}$, where n is the number of GTs recognized by the gazetteer. However, as will be described in Algorithm 2, because an instance may have multiple GTs, the weight of that instance for each of those types is divided proportionately. Finally, an EBD calculation over the different GTs is performed.

Algorithm 2 describes the process by which GTs and weights are assigned to instances. The input to the algorithm is the list of available GTs that are recognized by gazetteer G, along with IL, the list of instances associated with a given attribute, while the output is an ordered pair consisting of the GT vector list and GW vector list for the given attribute. Line 2 begins a loop that considers all instances in IL. Line 3 retrieves the set of GTs from $\mathcal{T}_{gaz}$ that instance x is associated with. Lines 4-5 assign the weight of each feature from $\mathcal{T}_x$ associated with the current instance. Line 6 assigns to $\mathcal{W}_x$ the individual weight values calculated in lines 4-5. Lines 7-8 aggregate the GT and weight vectors computed for instance x to $\mathcal{T}_{att}$ and $\mathcal{W}_{att}$, respectively. Finally, these vectors are returned as an ordered pair to GSim, which facilitates the EBD calculation between two compared attributes.

Algorithm 2 lookupGeoTypes ($\mathcal{T}_{gaz}$, IL)

Input:
- Set of geographic types $\mathcal{T}_{gaz}$ recognized by gazetteer
- List of instances IL associated with attribute att(T)

Output: an ordered pair ($\mathcal{T}_{att}$, $\mathcal{W}_{att}$) across all instances of att(T)

```
1: $\mathcal{T}_{att} = \Phi$, $\mathcal{W}_{att} = \Phi$
2: For each instance x ε IL {
3:     $\mathcal{T}_x$ = typeLookup ($\mathcal{T}_{gaz}$, x)
4:     For each t ε $\mathcal{T}_x$
5:         $w_t = 1 / | \mathcal{T}_x |$
6:     $\mathcal{W}_x = \{w_1 \ldots w_{last}\}$
7:     $\mathcal{T}_{att} = \mathcal{T}_{att} \cup \mathcal{T}_x$
8:     $\mathcal{W}_{att} = \mathcal{W}_{att} \cup \mathcal{W}_x$
9: }//end for
10: return ($\mathcal{T}_{att}$, $\mathcal{W}_{att}$)
```

Formally, let $\mathcal{T}_{gaz} = \{GT_1, GT_{2,\ldots\ldots}, GT_m\}$ be a set of GTs recognized by gazetteer G, with GT_i, $0 <= i <= m$, representing one of these types. For example, GT_i may be a county, city, state, etc. An arbitrary instance x associated with attribute att(T) will be associated with a GT vector $\mathcal{T}_x = \{GT'_1, GT'_2, \ldots GT'_n\}$, $n <= m$ and $n > 0$. Let $\mathcal{W} = (w_1, w_{2,\ldots\ldots} w_n)$ be a GW vector, where each w_j is associated with each GT'_j in $\mathcal{T}_x$ for instance x, where $|\mathcal{W}| > 0$ and all w_k in $\mathcal{W}$ for x have a value of $1 / |\mathcal{W}|$. For example, if x was associated with three GTs, then the weight w_j of each type t_j' for x would be 1/3.

As an example of illustrating the weighting of GTs, taking all instances from Figure 2 into account, the total weighting for the types listed are as follows: "City" = (1/3 + 0 + 1/2 + 0) = 5/6, "State" = (1/3 + 0 + 0 + 0) = 1/3, "Feature" = (1/3 + 0 + 1/2 + 0) = 5/6, and "County" = (0 + 1 + 0 + 1) = 2.

4.1.3 Handling Incomplete Data

GSim also possesses the ability to retrieve the appropriate instance information for a compared attribute if it is missing or incomplete. This is accomplished by leveraging the latitude and longitude value associated with that instance and performing reverse geocoding to retrieve the relevant instance information[22]. Specifically, this was accomplished using Google Maps reverse geocoding service [20]. Retrieving this missing information is important because it can affect the final EBD value calculated between two compared attributes despite many of the instance values missing. This can be useful, for example, in situations where tables containing attributes with sparse instances need to be compared.

4.1.4 Attribute Weighting

GSim also provides attribute weighting capabilities to penalize strong semantic correspondences between tables resulting from attribute mappings where the attributes in the mapped pair commonly-occur across all of the tables in their respective databases. Doing this allows us to refine the semantic similarity score generated between tables by focusing on the compared attributes that are unique relative to attributes found throughout all tables. Let $S_1 = (T_{11,}\ T_{12.....}T_{1M})$ be the set of tables belonging to data source S_1, and let $S_2 = (T_{21,}\ T_{22....}T_{2N})$ be the set of tables belonging to data source S_2, and suppose T_{1J} and T_{2K} are being compared for semantic similarity. Further suppose for the sake of simplicity that pairings between attributes of T_{1J} and T_{2K} have been set such that for all i, attribute i of T_{1J} is matched with attribute i of T_{2K}, and T_{1J} and T_{2K} have the same number of attributes. Before attribute weighting is applied, semantic similarity calculations between attribute i of T_{1J} and attribute i of T_{2K} occur. At this point, the EBD values of each attribute pair have equal weight. Recall that attribute-level EBD tells us which attributes are similar between compared tables. We will designate one such value between two attributes as EBD_{orig} (att(T_{1J}), att(T_{2K})). Realistically, however, some attribute pairs should be weighted higher than others. For example, given two tables, one called Road and another called Street, if the attribute 'roadType' in the Road table (let us call it Road.roadType) was mapped to an attribute 'streetType' in the Street table (let us call it Street.streetType), then this pair should contribute more substantially to the table similarity between Road and Street than a mapped attribute pair consisting of Road.roadName and Street.streetName. While Road.roadType and Street.streetType are two attributes that are not likely to be found in many other GIS tables, Road.roadName and Street.streetName are indeed likely to appear in other GIS tables, if, for example, these tables describe geographic objects that have some kind of street address such as a school, port or business.

The key to attribute weighting in GSim is determining the uniqueness of a given attribute. This is accomplished by calculating the frequency of any attribute participating in a match throughout the tables of its database. Equation 2 below, which is based on inverse document frequency, computes this uniqueness:

$$AU_{att} = \log \frac{|S_{att}|}{|\{t: att \in t\}|} >= 0 \qquad (2)$$

In equation 2, AU_{att}, the attribute uniqueness of *att*, is computed. S_{att} indicates the data source which contains attribute att (in one or more of its tables), and thus, $|S_{att}|$ indicates the number of tables in this data source. The variable t represents an arbitrary table in S_{att} which contains attribute att. A high AU_{att} value is achieved when attribute att appears infrequently across the tables of S_{att}, while a low value of AU_{att} occurs for an attribute that is commonly-occurring across the tables of S_{att}. The lowest possible value of AU_{att} is 0, which occurs when an attribute is found in every table of its database. Recall that a single EBD value is between two attributes, and thus, to measure pairwise uniqueness, we need a measure that accounts for the AU_{att} value for both attributes in a pair. This measure, called pair uniqueness and designated as $PU_{att1,att2}$, may be calculated by taking the arithmetic of the AU_{att} values for each attribute in a pair, the minimum AU_{att} value out of the pair, the maximum AU_{att} value out of the pair, and in a number of other ways. For our purposes, we achieved the most promising results when calculating $PU_{att1,att2}$ as the average of AU_{att1} and AU_{att2}. An important property of $PU_{att1,att2}$ is that it must be greater than or equal to 0, since the lowest value AU_{att} can have is 0.

Pair uniqueness is then multiplied by the EBD_{orig} value produced by the pair to give a corrected value called EBD_{corr}:

$$EBD_{corr}(att1, att2) = EBD_{orig}(att1, att2) \times PU_{att1,att2} \qquad (3)$$

The difference between $EBD_{corr(att1,att2)}$ and $EBD_{orig(att1,att2)}$, called pairwise semantic disparity ($PSD_{att1,att2}$), is then found between att1 and att2, and for all pairs of matching attribute pairs between T_{1J} and T_{2K}:

$$PSD_{att1,\ att2} = EBD_{orig(att1,att2)} - EBD_{corr(att1,att2)} \qquad (4)$$

Next, the arithmetic mean of the PSD values, dubbed PSD_{avg}, amongst all of the attribute pairs for a table comparison is found. An attribute pair with a PSD value above PSD_{avg} indicates that a greater discrepancy exists between EBD_{orig} and EBD_{corr} relative to other attribute pairs. As a result, this pair should have the weight of its EBD_{orig} value reduced in regards to table similarity. In contrast, an attribute pair with a PSD value below PSD_{avg} indicates that relative to other pairs, its EBD discrepancy was less, and because of this, its attributes are more unique. Thus its EBD_{orig} value should contribute more substantially to semantic similarity between the tables. The new weight assigned to the attribute pair depends upon how far above or below the PSD value is relative to PSD_{avg} and relative to the PSD value of other attribute pairs.

As an example, let four attribute mappings exist between two tables, designated att1a – att1b, att2a – att2b, att3a – att3b and att4a –att4b. The pair att1a – att1b has an PSD of .174, att2a – att2b has a PSD value of .113, att3a – att3b has a PSD value of .088 and att4a-att4b has a PSD value of .119.Then PSD_{avg} would be .1235, the weight of the EBD_{orig} between the attribute pairs would be changed as follows: att1a – att1b drops to 12.5%, att2a – att2b increases to 27.6%, att3a – att3b increases to 26.1%, and att4a – att4b increases to 33.8%.

Algorithm 3 AttributeWeighting (T, T')

```
Input: Tables T and T', which are being semantically compared
Output: A weight vector $W_{att(T)-att(T')}$ containing normalized weights for each
attribute pair among T and T'

1: $\mathcal{M}_{att(T),att(T')}$ = getAttributeMappings(T, T')
2: For each attribute pair (att1(T), att2(T')) ε $\mathcal{M}_{att(T),att(T')}$ {
3:    $AU_{att1(T)} = \log(|S_{att1(T)}| / |t: att1 \in t|)$
4:    $AU_{att2(T')} = \log(|S_{att2(T')}| / |t: att2 \in t|)$
5:    $PU_{att1(T),att2(T')} = AU_{att1(T)} + AU_{att2(T')} / 2$
6:    $EBD_{corr}(att1(T),att2(T')) = EBD_{orig}(att1(T),att2(T')) \times PU_{att1(T),att2(T')}$
7:    $PSD_{att1(T),att2(T')} = EBD_{corr}(att1(T),att2(T')) - EBD_{orig}(att1(T),att2(T'))$
8: }//end for
9: $PSD_{avg}$ = computeAvg($\mathcal{M}_{att(T),att(T')}$)
10: For each attribute pair (att1(T), att2(T')) ε $\mathcal{M}_{att(T),att(T')}$ {
11:    $PSDRel_{att1(T),att2(T')} = PSD_{att1(T),att2(T')} - PSD_{avg}$
12:    if ($PSDRel_{att1(T),att2(T')} > 0$)
13:        $W_{att1(T),att2(T')}$ = reduceWeight(att1(T), att2(T'), $PSDRel_{att1(T),att2(T')}$)
14:    if ($PSDRel_{att1(T),att2(T')} < 0$)
15:        $W_{att1(T),att2(T')}$ = increaseWeight(att1(T), att2(T'), $PSDRel_{att1(T),att2(T')}$)
16: }//end for
17: return $W_{att(T)-att(T')}$
```

Attribute weighting, as described above for a single table comparison, is illustrated in Algorithm 3.

4.2 Non-Geographic Matching

If GT matching between compared attributes is not possible, then a non-geographic semantic similarity measure is applied by GSim. The distance metric used for NGT matching is known as the normalized Google distance. The EBD is then calculated by extracting the keywords making up compared instances and assigning them generalized semantic types. These types are represented as clusters of keywords, whose semantic distance from each other is given by NGD.

Section 4.2.1 below first gives an overview of NGT matching using NGD. Section 4.2.2 provides further details on the K-medoid clustering process, which is instrumental to the success of NGT matching.

4.2.1 Overview of NGT Matching

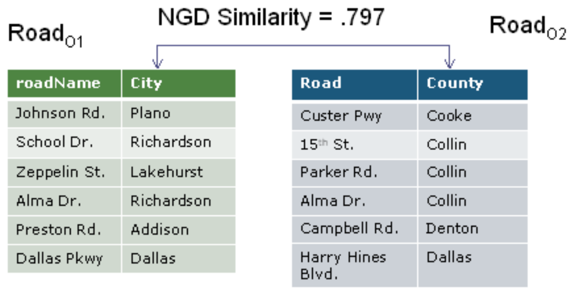
The algorithm for calculating the EBD between two compared attributes of tables in different data sources using NGT matching is as follows. The input is two compared attributes, with each one originating from a separate table, while the output is an EBD value indicating the semantic similarity between the input attributes. First, the respective keyword lists for each input attribute are extracted. Second, the keyword lists are combined into a single list for the comparison. This list is dubbed as $L_{keywords}$. Third, all pairwise distances between the keywords are computed with the help of an external NGD repository, resulting in a pairwise NGD dictionary. Fourth, the K-medoid() algorithm, which is described in Section 4.2.2., is executed, yielding a set of clusters, or NGTs, that represent generic semantic types. Finally, the calculation of EBD proceeds given the NGTs produced by K-medoid().

4.2.2 K-Medoid Clustering

The algorithm begins by determining the number of clusters K based on the size of $L_{keywords}$ for each pair of compared attributes. Second, exactly one keyword from $L_{keywords}$ is assigned to each of the K clusters in a process called initial seeding. Each of these keywords is then considered a medoid for its particular clustering. Third, we continuously assign each remaining keyword in $L_{keywords}$ that is not a medoid to the cluster to which it is most semantically related, while subsequently determining if any cluster medoids need to be recomputed. To do this, we need to use the NGD values between the keyword to be assigned to a cluster and all keywords already assigned to that same cluster. A given keyword, k_{new} is assigned to the cluster associated with the smallest summation of the NGD values between k_{new} and the cluster's constituent keywords. After all keywords have been assigned to clusters, finally, we determine if the medoid for any cluster needs to be recomputed. This is accomplished by examining each of the keywords in a particular cluster and computing an NGD summation between a single keyword in that cluster and all other words in that cluster. The keyword in that cluster that produces the lowest NGD summation will be assigned as the new medoid for that cluster. If no medoids have changed in any cluster, then the K-medoid algorithm is finished, and control proceeds to the calculation of the EBD between the compared attributes. However, if at least one medoid has changed in a particular cluster, then we begin a new clustering iteration.

4.3 Justification of GT Matching in GSim

Despite the utility of NGD over a number of domains, it tends to produce inaccurate results with regards to the GIS domain when the compared instances are geographically proximate, despite being completely different types. Figure 4 describes one particular example of this phenomenon.


NGD Similarity = .797
$Road_{O1}$
$Road_{O2}$

roadName	City
Johnson Rd.	Plano
School Dr.	Richardson
Zeppelin St.	Lakehurst
Alma Dr.	Richardson
Preston Rd.	Addison
Dallas Pkwy	Dallas

Road	County
Custer Pwy	Cooke
15th St.	Collin
Parker Rd.	Collin
Alma Dr.	Collin
Campbell Rd.	Denton
Harry Hines Blvd.	Dallas

Figure 4. Example of how NGD can produce an inaccurate semantic similarity computation in geodata if the instances being compared are geographically proximate

The attribute "City", associated with table $Road_{S1}$ is compared against the attribute "County" from table $Road_{S2}$. Although the instances are of different types, they are geographically proximate, as both the cities from "City" and the counties from "County" both describe the Dallas-Fort Worth area. As a result, even though the types are totally different, the exclusive usage of NGD will deem that the "City" attribute is semantically similar to the "County" attribute. This happens because NGD, by definition, is computed based on the probability of the co-occurrence of search terms x and y on a given web page indexed by the Google search engine. In many situations, a high probability of co-occurrence between x and y indicates that the terms are likely to be semantically similar to one another. However, as figure 4 shows, co-occurrence does not always imply similarity. Therefore, in order to correctly determine whether attribute pairs such as this one match, geographic type extraction is essential.

5. EXPERIMENTS

We now present three separate experiments that we conducted regarding matching between distinct data sources in the GIS domain. The first experiment measured GSim's ability to compute semantic similarity between two pairs of GIS databases. The second experiment tested GSim's ability to perform semantic similarity between tables containing a significant amount of missing geographic information. The third experiment illustrated GSim's attribute weighting feature, which gives it the ability to penalize table matches involving commonly-occurring attributes found through the GIS database and reward table matches containing attribute pairs that were unique to their respective tables.

5.1 Semantic Similarity Experiment

5.1.1 Dataset Details

In the first experiment, two datasets from the GIS domain were used to evaluate the performance of GSim. The first dataset was created from instance data of the Road and Ferries package of a GIS data model known as GDF (Geographic Data Files)[21]. The second dataset details a wider assortment of GIS location features across the United States and their associated data beyond merely transportation networks. Some of the location features in this dataset include flight schools, piers, navigable waterways and Indian lands. For both sets of data, the number of attributes and instances vary widely; for example, in the GIS location dataset, the Flight Schools table has the fewest number of attributes (27) and the Piers table has the most (76). Because data from several different areas of the United States were employed in our experiments, we effectively created a disjoint, multi-jurisdictional environment. Table 1 below displays a summary of the relevant information regarding the data involved in our experiments with both datasets.

Table 1. Description of transportation dataset(top) & GIS Location Dataset (below)

Table Name	No. of Attributes	Area(s) Modeled	No. of Instances
Road(S_1), Road(S_2)	18,11	Fort Collins, CO Dallas, TX	9851, 5224
Ferry(S_1), Ferry(S_2)	3,8	Seattle, WA	24,42
Traffic Area (S_1), Traffic Area (S_2)	24,26	Virginia	329, 108
Residential Area (S_1), Address Area (S_2)	15,10	New Jersey, Texas	4263, 2122

Table Name	No. of Attributes	No. of Instances
Flight Schools (S_1), Flight Schools (S_2)	27	4653 and 4653, respectively
Schools (S_1), Schools (S_2)	41	57728 and 56730, respectively
Piers (S_1)	76	6159
Indian Lands (S_1)	52	15852
Ports (S_2)	56	4534
NavWaterways (S_2)	30	6879

Table 2a and 2b. Precision + recall values between tables of S_1 and S_2 using N-grams and GSim relative to a ground truth for (a: transportation dataset (b: GIS location dataset

t ε S_1	t ε S_2	P (N-gram)	R (N-gram)	P (GSim)	R(GSim)
Road	Road	.20 (1/2)	.20 (1/4)	.50 (2/4)	.50 (2/4)
Road	Address Area	.25 (1/4)	1.00(1/1)	1.00 (1/1)	1.00(1/1)
Road	Enclosed Traffic Area	.33 (1/3)	.50 (1/2)	.50 (1/2)	.50 (1/2)
Road	Ferry	0 (0/2)	0 (0/1)	.50 (1/2)	1.00 (1/1)
Residential Area	Road	.25 (1/4)	1.00 (1/1)	.50 (1/2)	1.00 (1/1)
Residential Area	Address Area	.50 (2/4)	.50 (2/4)	.75 (3/4)	.75 (3/4)
Residential Area	Enclosed Traffic Area	.25 (1/4)	.50 (1/2)	.50 (1/2)	.50 (1/2)
Residential Area	Ferry	------ (0/0)	0 (0/1)	1.00 (1/1)	1.00 (1/1)
Traffic Area	Road	.33 (1/3)	.50 (1/2)	.50 (1/2)	.50 (1/2)
Traffic Area	Address Area	.50 (1/2)	.50 (1/2)	1.00 (1/1)	.50 (1/2)
Traffic Area	Enclosed Traffic Area	.50 (1/2)	.50 (1/2)	1.00 (2/2)	1.00 (2/2)
Traffic Area	Ferry	.50 (1/2)	1.00 (1/1)	1.00 (1/1)	1.00 (1/1)
Ferry	Road	.50 (1/2)	1.00 (1/1)	1.00 (1/1)	1.00 (1/1)
Ferry	Address Area	1.00 (1/1)	1.00 (1/1)	1.00 (1/1)	1.00 (1/1)
Ferry	Enclosed Traffic Area	.50 (1/2)	1.00 (1/1)	1.00 (1/1)	1.00 (1/1)
Ferry	Ferry	.50 (1/2)	.33 (1/3)	.67 (2/3)	.67 (2/3)
AVERAGE VALUES		.38	.52	.70	.72

t ε S_1	t ε S_2	P (N-gram)	R (N-gram)	P (GSim)	R(GSim)
Flight Schools	Flight Schools	1.00 (2/2)	.29 (2/7)	1.00 (7/7)	1.00 (7/7)
Flight Schools	Schools	------ (0/0)	0 (0/3)	1.00 (2/2)	.67 (2/3)
Flight Schools	Ports	------ (0/0)	0 (0/3)	.33 (1/3)	.33 (1/3)
Flight Schools	NavWaterways	------ (0/0)	0 (0/3)	.33 (1/3)	.33 (1/3)
Schools	Flight Schools	------ (0/0)	0 (0/3)	1.00 (3/3)	1.00 (3/3)
Schools	Schools	.66 (2/3)	.33 (1/3)	1.00 (3/3)	1.00 (3/3)
Schools	Ports	------ (0/0)	0 (0/3)	1.00 (1/1)	.33 (1/3)
Schools	NavWaterways	------ (0/0)	0 (0/3)	1.00 (3/3)	1.00 (3/3)
Indian Lands	Flight Schools	------ (0/0)	0 (0/3)	.33 (1/3)	.33 (1/3)
Indian Lands	Schools	------ (0/0)	0 (0/3)	1.00 (1/1)	.33 (1/3)
Indian Lands	Ports	------ (0/0)	0 (0/3)	1.00 (1/1)	.33 (1/3)
Indian Lands	NavWaterways	------ (0/0)	0 (0/2)	1.00 (2/2)	1.00 (2/2)
Piers	Flight Schools	------ (0/0)	0 (0/5)	1.00 (2/2)	.40 (2/5)
Piers	Schools	------ (0/0)	0 (0/3)	1.00 (1/1)	.33 (1/3)
Piers	Ports	------ (0/0)	0 (0/3)	.50 (1/2)	.67 (2/3)
Piers	NavWaterways	------ (0/0)	0 (0/2)	.67 (2/3)	1.00 (2/2)
AVERAGE VALUES		.80	.06	.80	.61

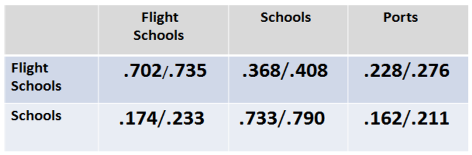
5.1.2 Measurements and Parameters

The results of the alignment of S_1 and S_2 of the compared tables for both the transportation dataset and the GIS location dataset using GSim and the N-gram method are shown in Tables 2a and 2b, respectively. For each table comparison, there are four values. From left to right, the first two are the precision and recall (denoted as P and R, respectively) produced using N-grams between an attribute from a table in data source S_1 and an attribute from a table in data source S_2. The last two values are the precision and recall values produced by GSim between an attribute from a table in data source S_1 and an attribute from a table in data source S_2. As an example, for the comparison of Road from S_1 and Ferry from S_2 in table 2a, the precision and recall generated using N-grams are 0 and 0, respectively, while the precision and recall generated for GSim is .50 and 1.00, respectively. Next to each value is a ratio enclosed in parentheses; the numerator indicates the number of attribute mapping "hits" for a given table comparison and matching method, while the denominator indicates the total number of attribute mappings to be "hit" for that table comparison. The values produced by both methods exist relative to a reference alignment, or ground truth, which contains the attribute pairs that are supposed to be semantically similar. The ground truth for both datasets was created by human experts knowledgeable in the area of GIS. For our experiments, we set two standards that affected the results. First, we decided that whenever an attribute pair produced a similarity value (an EBD value) measured to be greater than or equal to .6, then the method determined that pair to be a match. Second, we set N-grams to be of size 2, since any size greater than 2 would increase the number of possible N-grams by a margin significant enough such that the precision and recall values would almost always be too low to meet the match threshold for any dataset, thus rendering this method virtually useless as a semantic similarity measure for our experiments. Overall, the ground truth for the transportation dataset contained 29 correct mappings across all table comparisons, while the ground truth for the GIS location dataset contained 52 correct mappings across all table comparisons.

5.1.3 Analysis of Results

Table 2a shows the comparison of precision and recall values using both GSim and the N-gram method for the transportation dataset. Note that the precision and recall values generated by GSim are never lower than those produced by N-grams for any table comparison. In total, the average precision produced by GSim was .70, and its average recall was .72. In contrast, the average precision of N-grams was .38, and its average recall was .52. GSim achieved an 32% improvement over N-grams in precision, and a 20% improvement in recall. In fact, the only reason why N-grams even performed somewhat competently in this dataset was because of the large number of identical instances between many attribute pairs that happened to be semantically similar. Table 2b depicts even more dramatic improvements made by GSim. The precision and recall values for GSim are always higher than those produced by the N-gram method for any table comparison. In total, the average precision produced by GSim was .80, and its average recall was .61. In contrast, while the average precision of N-grams is .80, the average recall is a staggeringly low value of .06. In fact, the reason why N-grams' precision was able to match GSim's precision was due to the extremely low recall. The reason for the low recall value was primarily due to the lack of identical instances between the compared attributes. As a result, most of the comparisons using the N-gram method were not able to reach the .60 threshold in semantic similarity. We did not lower the match threshold below .6 because we felt that a match threshold of a value that was lower, such as .5, would not be a realistic match threshold for determining whether two schemas were similar or not. As a result, GSim's intrinsic semantic capabilities, largely resulting from GT extraction, allow it to achieve a 55% improvement on recall versus a syntactic method such as N-grams.

Table 3. EBD values generated between selected tables of S_1 and S_2 of GIS location dataset when incomplete values in data are filled in by gazetteer (number left of slash) and when the complete data is available (number right of slash)



	Flight Schools	Schools	Ports
Flight Schools	**.702/.735**	**.368/.408**	**.228/.276**
Schools	**.174/.233**	**.733/.790**	**.162/.211**

5.2 Reverse Geocoding Experiment

5.2.1 Measurements and Parameters

The results in Table 3 above were produced by our second experiment. There are two values in each cell. The value to the right of the slash indicates the final EBD value computed between compared tables (found by averaging all of the EBD values between the attribute pairs of the tables) where no instance data was missing. The value to the left is the EBD value computed when incomplete data existed in the compared attributes of both tables. In this experiment, we randomly removed 50% of the instance values from the "roadName" attribute common to all of the tables listed in Table 3, and from other attributes that participated in matches whose instance values represented geographic information. The idea was to determine the extent to which reverse geocoding, using latlong values for each instance and using our gazetteer, GeoNames, could approximate the correct semantic correspondence between tables.

5.2.2 Analysis of Results

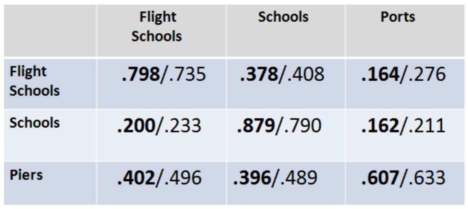
As table 3 shows, although reverse geocoding does not return every instance value of an attribute involved in a comparison, at worst, the EBD value computed with 50% of the instance values missing for at least one attributes is 25% less than the actual EBD value (Schools-Flight Schools), and on average, the decrease in EBD value across all table comparisons is 14.6%.

5.3 Attribute Weighting Experiment

5.3.1 Measurements and Parameters

To better illustrate the benefits of attribute weighting on matching tables, we preprocessed the attributes in a subset of the tables of the GIS location dataset from both databases to optimize GSim's ability to distinguish between commonly-occurring attributes and attributes that are more unique. The results of applying GSim's semantic weighting algorithm to a subset of the tables from the GIS location dataset are shown in Table 4 below. As can be seen, each cell contains two separate values; the value right of the slash represents the semantic similarity, measured as EBD, between the tables where all attribute mappings share equal weight, while the value left of the slash is the EBD produced when attribute weighting was used to distribute the weights among mappings according to uniqueness. The table names along the vertical axis of the table belong to S_1, while the tables across the horizontal axis of the table belong to S_2.

Table 4. Two separate EBD values computed between a table from S_1 and a table from S_2. The value in a given cell right of the slash indicates the EBD value produced with equal attribute weighting, while the value left of the slash is the EBD produced from attribute weighting



	Flight Schools	Schools	Ports
Flight Schools	**.798**/.735	**.378**/.408	**.164**/.276
Schools	**.200**/.233	**.879**/.790	**.162**/.211
Piers	**.402**/.496	**.396**/.489	**.607**/.633

5.3.2 Analysis of Results

The results of table 4 show that in most cases, using attribute weighting causes the EBD between corresponding tables to be strengthened and the EBD between dissimilar tables to be weakened. The only exception to this trend was the drop in EBD measured between the Piers and Ports tables. This might indicate that the fairly strong semantic similarity between the tables was due to the contributions of commonly occurring attributes that were mapped to each other. Otherwise, the use of attribute weighting increase the EBD between pairs of corresponding tables (Flight Schools – Flight Schools and Schools-Schools) by 8.5% and 11.3%, respectively. Additionally, attribute weighting was used to reduce the semantic similarity between dissimilar table pairs (not including Piers-Ports) by an average of 19.1%.

6. CONCLUSION AND FUTURE WORK

In this paper, we described GSim, an algorithm that computes the semantic similarity of two tables belonging to distinct GIS data sources. It computes semantic similarity using two separate approaches. The first uses a gazetteer to extract GTs for all possible instances within the compared attributes. The weights of the GTs taken over all instances results in GT sets and GT weight lists, where each attribute features its own GT set and GT weight list. The similarity of these distributions determines the semantic similarity between the attributes, and the average over all attribute pairs determines the table similarity. GSim also compensates for situations when a lack of GT information for the instances is available by executing a domain independent semantic similarity algorithm leveraging normalized google distance. This results in the extraction of NGTs from the instances of the attributes, and semantic similarity is subsequently computed. GSim also uses reverse geocoding to make table comparisons possible, even with incomplete data. Additionally, GSim provides attribute weighting capabilities across tables in a GIS database that penalizes the similarity between table matches involving a high number of commonly occurring attributes found throughout the database, while enhancing table matches containing unique attribute mappings. Future efforts to improve GSim will focus on refining our GT extraction techniques so that we can leverage multiple gazetteers making use of heterogeneous feature type thesauri while enhancing our recall of the correct type information. Also, we plan to take into account geo-tagged information associated with images in order to enhance the accuracy of our matching scores.

7. REFERENCES

[1] Luiz André P. Paes Leme, Marco A. Casanova, Karin Koogan Breitman, Antonio L. Furtado: Instance-Based OWL Schema Matching. ICEIS 2009: 14-26.

[2] Daniela F. Brauner, Chantal Intrator, João Carlos Freitas, Marco A. Casanova: An Instance-based Approach for Matching Export Schemas of Geographical Database Web Services. GeoInfo 2007: 109-120.

[3] Daniela F. Brauner, Marco A. Casanova, Ruy Luiz Milidiú: Towards Gazetteer Integration Through an Instance-based Thesauri Mapping Approach. GeoInfo 2006: 189-198.

[4] Isabel F. Cruz, William Sunna, Nalin Makar, Sujan Bathala: A visual tool for ontology alignment to enable geospatial interoperability. J. Vis. Lang. Comput. 18(3): 230-254 (2007).

[5] E.Ralun and P. A. Bernstein, "A survey of approaches to automatic schema matching", *VLDB Journal*, vol. V10, pp. 334-350, 2001.

[6] Bing Tian Dai, Nick Koudas, Divesh Srivastava, Anthony K. H. Tung, and Suresh Venkatasubramanian, "Validating Multi-column Schema Matchings by Type," *24th International Conference on Data Engineering (ICDE),* 2008.

[7] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster, "Putting context into schema matching." in VLDB, 2006, pp. 307–318.

[8] R. H. Warren and F. W. Tompa, "Multi-column substring matching for database schema translation." in *Proc. VLDB*, 2006, pp. 331–342.

[9] W.S. Li and C. Clifon, "Semint: a tool for identifying attribute correspondence in heterogeneous databases using neural networks," *Data Knowl. Eng.*, vol. 33, no. 1,pp.49-84, 2000.

[10] J. Berlin and A. Motro, "Autoplex: Automated discovery of instance for virtual databases," in *Proc. CoopIS*, 2001, pp. 108-122.

[11] D.W. Embley, L. Xu, and Y. Ding, "Automatic direct and indirect schema mapping: experiences and lessons learned," SIGMOD Rec., vol. 33, no. 4, pp. 14–19, 2004.

[12] Changqing Zhou, Dan Frankowski, Pamela J. Ludford, Shashi Shekhar, Loren G. Terveen: Discovering personal gazetteers: an interactive clustering approach. GIS 2004: 266-273.

[13] Shawn Newsam, Yi Yang: Integrating gazetteers and remote sensed imagery. GIS 2008: 26.

[14] Bruno Pouliquen, Ralf Steinberger, Camelia Ignat, Tom De Groeve: Geographical information recognition and visualization in texts written in various languages. SAC 2004: 1051-1058.

[15] Changqing Zhou, Dan Frankowski, Pamela J. Ludford, Shashi Shekhar, Loren G. Terveen: Discovering personally meaningful places: An interactive clustering approach. ACM Trans. Inf. Syst. 25(3): (2007).

[16] Dhiraj Joshi, Jiebo Luo: Inferring generic activities and events from image content and bags of geo-tags. CIVR 2008: 37-46.

[17] Erik Wilde, Martin Kofahl: The locative web. LocWeb 2008: . 1-8.

[18] www.geonames.org

[19] Jeffrey Partyka, Neda Alipanah Latifur Khan, Bhavani Thuraisingham and Shashi Shekhar, "Content-based Ontology Matching for GIS Datasets", University of Texas at Dallas (UTD Technical Report # UTDCS-22-08).

[20] http://code.google.com/apis/maps/documentation/services.html#ReverseGeocoding

[21] http://www.ertico.com/en/about_ertico/links/gdf_-_geographic_data_files.htm

[22] Michael D. Lieberman, Hanan Samet, Jagan Sankaranarayanan, and Jon Sperling: STEWARD: architecture of a spatio-textual search engine. In Proceedings of the 15th International Symposium on Advances in Geographic Information Systems. (ACM GIS 2007).

[23] Rudi Cilibrasi, Paul M. B. Vitányi: The Google Similarity Distance CoRR abs/cs/0412098:(2004)

Smarter Searches using Location Driven Knowledge Discovery and Mining

Satyen Abrol Tahseen Al-khateeb Latifur Khan
Department of Computer Science
University of Texas at Dallas
{*sxa079300, tahseen, lkhan*}@utdallas.edu

ABSTRACT

In the present world scenario, everybody is on the lookout for suitable housing options, each having different needs (e.g. elderly are looking for safe, quiet neighborhood, while students are looking for affordable apartments close to the university/school). For e.g. *Craigslist* currently does not have a map version, making the process of apartment searching a very long and laborious process. This creates a need for software that is significantly superior to current web search tools. We demonstrate the development of such a tool which takes the *craigslist* apartment listings as the input, and provides the user with the output on Google maps. We then integrate this functionality with the information collected from location based extraction of various web sources such as the city police blotter which makes apartment searching simpler and faster, helping the user to make a better decision. We also discuss the challenges that are faced in the development process, the raw and unstructured nature of the documents, the existence of Geo-Non Geo & Geo-Geo disambiguities and our approach in identifying the location of the apartment from informal text (geo-parsing and geo-tagging of content) to ensure maximum coverage of the listings. In this prototype we also show integration of point of interests such as locations of grocery stores, religious places, hospitals etc. along with the advertisement on the maps.

Categories and Subject Descriptors
D.3.3 [**Information Search and Retrieval**]

General Terms
Design, Algorithm, Experimentation, Verification

Keywords
Information retrieval, Text mining, Natural language processing, Gazetteer, Geo-parsing, Disambiguation

INTRODUCTION

The task of identifying the correct location of documents such as emails, news, web pages etc. has always been greatly beneficial for the purposes of data mining and information retrieval. The identification of location can be used for location based services e.g. to find the nearest ATM machine to your house. The internet, now days, is filled with locally targeted advertisement. So, if an end user says that he is right now in Dallas, the website displays ads for restaurants in and around Dallas. And similarly the search engines, after determining the user's location give a higher weight to search results which refer to his geographic area.

Geo-parsing is the process of determining geographic coordinates of textual words and phrases that occur in unstructured content, such as "six miles east of Paris". You can also geo-parse location references from other forms of media, e.g. audio content in which a speaker mentions a place. With geographic coordinates the features can be mapped and entered into Geographic Information Systems. Once the coordinates are identified the applications plot the geo-parsed text on to a map.

Geo-parsing goes beyond geo-tagging (or geo-coding) as it deals with ambiguous and unstructured text. There are two types of ambiguities that exist: Geo/Non-Geo and Geo/Geo ambiguities. Geo/non-geo ambiguity is the case of a place name having another, non geographic meaning, e.g. Paris might be the capital of France or might refer the socialite, actress Paris Hilton. Geo-geo ambiguity arises from the two having the same name but different geographic locations, e.g. Paris is the capital of France and is also a city in Texas. Smith *et al.* report that 92% of all names occurring in their corpus are ambiguous [7].

Researchers have used a variety of methods to tackle the problem of correctly geo-parsing the documents. In the domain of NLP, the techniques of machine learning are employed to identify the location from their structure and context. We take the Craigslist advertisements consisting of raw unstructured text, and identify locations from them. However, the extracted locations are ambiguous and we use our weight based algorithm to identify one single correct location.

In our work, we have made several contributions. First, a tool is developed that facilitates to display Craigslist advertisement onto Google maps along with other relevant information. This integrated information is obtained from aggregation and analysis of data from POI databases and police blotters in an efficient and timely manner. Second, we devise an efficient algorithm to identify and disambiguate the correct location from the unstructured text of the Craigslist advertisement. Third, we describe various scenarios that our algorithm exploits to devise heuristics and strategies that increase the coverage and accuracy on the map. Finally, we have developed a fully functional prototype and tested on real dataset collected from the Craigslist website.

The research paper is organized as follows. Section 2 analyzes craigslist and its problems as a source of geo-information. Section 3 describes the types of ads and the technical challenges faced involved in each of them. Section 4 discusses the disambiguation algorithm, and describes how it identifies and disambiguates the location in the text. Section 5 and 6 show how we associate a confidence value with the location and the integration of web sources respectively. Section 7 surveys and compares the related work in this domain. Section 8 and 9 discusses the results and concludes with some pointers for the future work.

CRAGSLIST AND ITS PROBLEMS

Craigslist is a centralized network of online communities, featuring free online classified advertisements – with sections devoted to jobs, housing, personals, for-sale, services, community, gigs, résumés, and discussion forums [1].

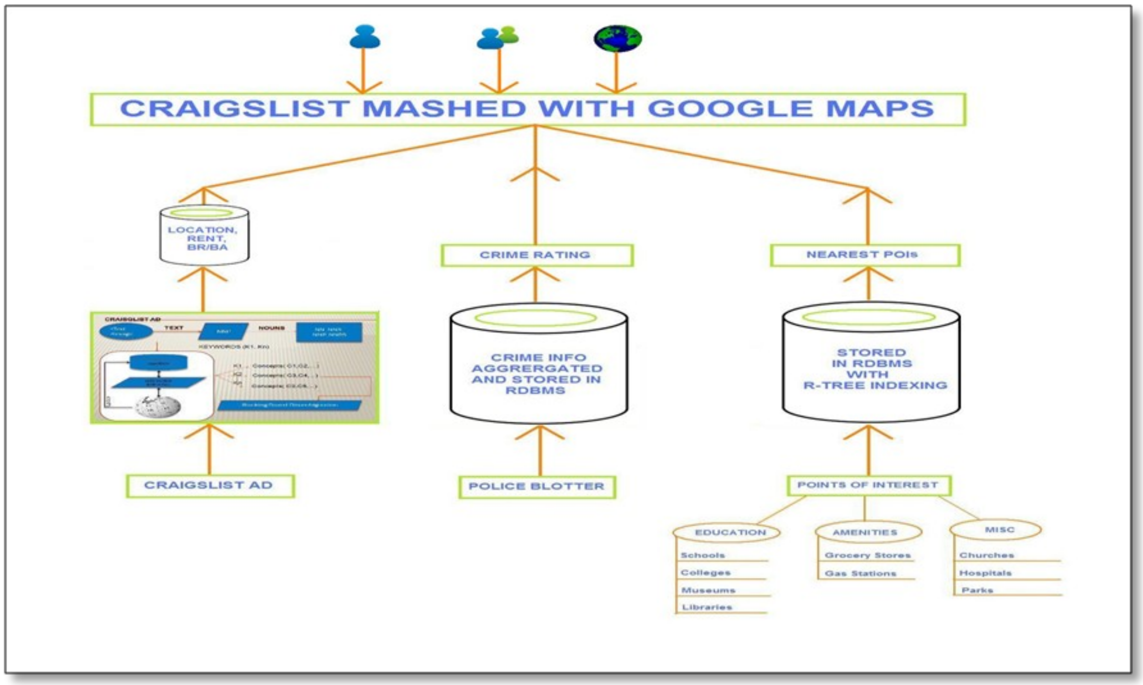


Fig 1: Architecture of the Craigslist-Google Maps system

The site serves over twenty billion page views per month, putting it in 22nd place overall among web sites worldwide, eighth place overall among web sites in the United States (according to Alexa.com on June 19, 2009).

Currently Craigslist does not support a map version. So, even if someone is looking for an apartment near a particular location, he/ she has to browse through hundreds of listings manually before one can come across a good potential apartment. This makes the process of apartment searching a long and unpleasant process. In addition to this, the user has to separately look on the internet for the other things like crime, median family income, and point of interests like grocery stores, religious places, hospitals etc. This creates a need for a tool that displays the Craigslist ads on the Google Maps, integrated along with crime statistics, school information, and other points of interest (POIs), so that it becomes easier for the user to make a decision.

The content of the *craigslist* ads consists of text that is unstructured and consists of a lot of grammatical and spelling errors. Therefore, it becomes more difficult to identify and disambiguate the location of the apartment/house.

Figure 1 illustrates the architecture of our system. Left most entry of the architecture shows the processing and storing of essential information from an advertisement. The middle and right databases store the crime information and the location based Points of Interests (POIs).

TECHNICAL CHALLENGES

As mentioned earlier, the Craigslist ads consist of unstructured data, usually having a location embedded in the text. Here, we describe six scenarios into which all of the Craigslist ads can be broadly categorized. We then describe how we deal with each of them so as to identify and disambiguate the location.

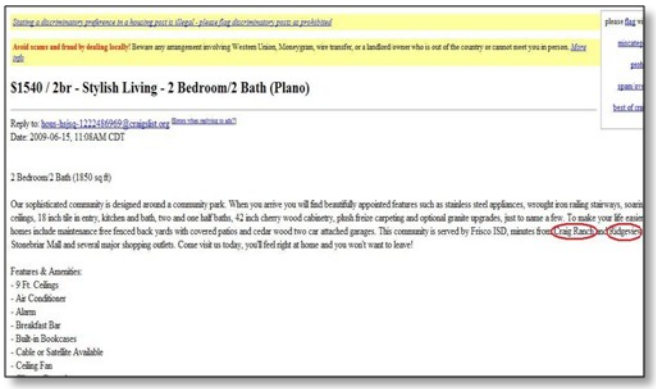
3.1 Ads with Physical Address (APA)

APA contains ads which has a complete physical address mentioned with house number, street name and zip. The ads with Google/Yahoo map links also fall in this category. For such ads the location extraction is done through Regular Expression matching and these ads usually have a CAF value (see section 5).

3.2 Ads with just One Street name (ASN)

ASN consists of the ads that have a street name or a location embedded in the usual unstructured text. It is for this case that we use the disambiguation algorithm, to identify the potential location of the apartment. The ads in this category in the absence of a block number fall in the medium or low confidence level category.

3.3 Ads with Intersections (AwI)



Stating a discriminatory preference in a housing post is illegal - please flag discriminatory posts as prohibited

Avoid scams and fraud by dealing locally! Beware any arrangement involving Western Union, Moneygram, wire transfer, or a landlord owner who is out of the country or cannot meet you in person. More info

$1540 / 2br - Stylish Living - 2 Bedroom/2 Bath (Plano)

Reply to: hous-hajsq-1222486969@craigslist.org
Date: 2009-06-15, 11:08AM CDT

2 Bedroom/2 Bath (1850 sq ft)

Our sophisticated community is designed around a community park. When you arrive you will find beautifully appointed features such as stainless steel appliances, wrought iron railing stairways, soari ceilings, 18 inch tile in entry, kitchen and bath, two and one half baths, 42 inch cherry wood cabinetry, plush frieze carpeting and optional granite upgrades, just to name a few. To make your life easie homes include maintenance free fenced back yards with covered patios and cedar wood two car attached garages. This community is served by Frisco ISD, minutes from Craig Ranch and Ridgevie Stonebriar Mall and several major shopping outlets. Come visit us today, you'll feel right at home and you won't want to leave!

Features & Amenities:
- 9 Ft. Ceilings
- Air Conditioner
- Alarm
- Breakfast Bar
- Built-in Bookcases
- Cable or Satellite Available
- Ceiling Fan

Fig. 2: A typical craigslist ad having the location as an intersection

Sometimes, the ad publisher describes the location of the apartment as "near A and B" or "A at B", where 'A' and 'B' are the street names. If the disambiguation algorithm returns two different street names with comparable weights and close proximity to each other, we check for the streets in the intersection database, for the possibility of an intersection and its coordinates.

3.4 Ads with just Phone Numbers (APN)

Ads with just a phone number and no mention of the street name or intersection are located using the White Pages reverse lookup. We get the location of the person to whom the phone is registered. Such ads associated with very low values of confidence since we have no proof whether the address is of the realtor or the actual apartment location. The same strategy is used to boost up the confidence level for ASN and AwI ads (see Disambiguation Algorithm for details)

3.5 Ads having just Neighborhood (AwN)

There is a major portion of the ads that has just the name of the neighborhood such as Uptown, Downtown, Turtle Creek etc. This can help us in narrowing down the area and we can increase the accuracy of the location. We search only the ads where the algorithm returns no address. We maintain a table of all popular neighborhoods for each city, created from information extracted from *Wikipedia* listings. We search the ads for the neighborhoods obtained from this table and on a match allocate the location of the apartment as the neighborhood. This also means a low CAF value (see Section 5) as compared to a physical address. This is especially helpful to users who are looking for an apartment in a particular area or neighborhood.

3.6 Ads with No Information (ANI)

This section is formed by ads where there is no mention of any street name or potential address, does not have a phone number or it is a mobile or unpublished number. For such ads, the identification of an accurate location is not possible and we just specify the city as the location.

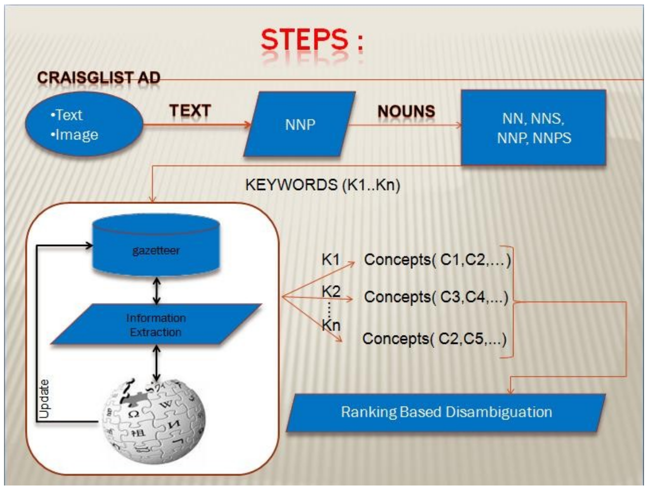


Fig. 3: Diagram showing the process of identification and disambiguation of locations

DISAMBIGUATION ALGORITHM

The algorithm LocationFinder(Ads) is divided into several steps. In this section we describe the each of the steps that go into the process of identification and disambiguation of the apartment location.

Algorithm 1 LocationFinder (Ads)

Input: Set of Ads for determining location
Output: location of the each of the ads

```
 1: For each ad A ε Ads
 2:     (S, W) ← Street_Disambiguation (A)
 3:     RS ← Street [Reverse_Phone (A)]
 4:     for each $S_i$ ε S do
 5:         NW ← Find Preceeding_Word ($S_i$)
 6:         $W_i$ ← $W_i$ + Wt_func(NW)
 7:         If ($S_i$==RS) then $W_i$ ← $W_i$ + $W_{RS}$    // If street from
            reverse lookup matches $S_i$ , then boost score
 8:   ($S_1$, $S_2$, R)  ← Intersect (S, W)   /* Pick two intersecting streets
 with highest weights with close mutual weights */
 9:     If (R==true) then
10:       location ← LatLong($S_1$, $S_2$)
11:       else location ← $S_{max-weight}$ (S, W)
```

In line 2, for each ad, we call the method, Street_Disambiguation that helps to identify and partly disambiguate the locations. The method returns the vector containing all possible street names with their weights. In line 3, we get search the text for a 10-digit number and use the White pages to do a reverse phone number lookup and extract the street name from the address. Next, for each street in the vector, we identify the words preceding and succeeding it. In line 6, on the basis of this we then boost up the weights of the street concepts. Then we boost up the scores of all those street concepts where the street names match from those obtained from the reverse phone lookup. After this iteration, we first check for the possibility of an intersection. For this we pass the whole vector to a method Intersect which returns true with the street names, S_1 and S_2; false otherwise. In case of an absence of an intersection, we choose the street with the maximum weight to be the location.

We now describe the Street_Disambiguation method in detail. The first step of the method involves removal of all those words from the craigslist text that are not references to geographic locations. For this, we use the CRF Tagger, which is an open source tagger for English with an accuracy of close to 97% and a tagging speed of 500 sentences per second [2]. The CRF tagger identifies all the proper nouns from the text and term them as keywords $\{K_1, K_2, \ldots, K_n\}$. In the next step, the TIGER (**T**opologically **I**ntegrated **G**eographic **E**ncoding and **R**eferencing system) [3] dataset is searched for identifying the street and city names from amongst them. The TIGER dataset is an open source gazetteer consisting of topological records and shape files with coordinates for counties, zip codes, street segments, etc. for the entire US.

Algorithm 2 Street_Disambiguation (A)

Input: A: Craigslist Ad

Output: Vector (S, W): Streets and weights vector

```
 // Phase 1
 1:  for each keyword, $K_i$
 2:      for each $S_j \in K_i$     //$S_j$ - Street Concept
 3:          for each $T_f \in S_j$
 4:              type $\leftarrow$ Type ($T_f$)
 5:              If ($T_f$ occurs in A) then $W_{Sj} \leftarrow W_{Sj} + W_{type}$
 // Phase 2
 6:  for each $K_i$
 7:      for each $S_j \in K_i$
 8:          for $T_f \in S_j$ , $T_s \in S_L$
 9:              If ($T_f = T_s$) and ($S_j \neq S_L$) then
10:                  type $\leftarrow$ Type ($T_f$)
11:                      $Weight_{Sj} \leftarrow Weight_{Sj} + W_{type}$
12:  return (S, W)
```

We search the TIGER gazetteer for the concepts $\{C_1, C_2 \ldots C_n\}$ pertaining to each keyword. Now our goal for each keyword would be to pick out the right concept amongst the list, in other words disambiguate the location. For this, we use a weight based disambiguation method. In the phase 1, we assign the weight to each concept based on the occurrence of its terms in the text. Specific concepts are assigned a greater weight as compared to the more general ones. In phase 2, we check for correlation between concepts, in which one concept subsumes the other. In that case the more specific concept gets the boosting from the more general concept. If a more specific concept C_i is part of another C_j then the weight of C_j is added to that of C_i.

4.1 Creating the Intersection Database

The intersection database is created from the TIGER shape files. TIGER dataset contains the streets divided into segments, each uniquely identified by the starting and the ending Nodes IDs. If two streets intersect, they will have at least one of the two nodes common to both as shown in Fig 4 and Fig 5.

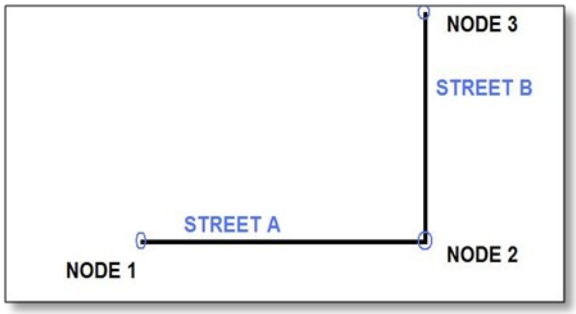
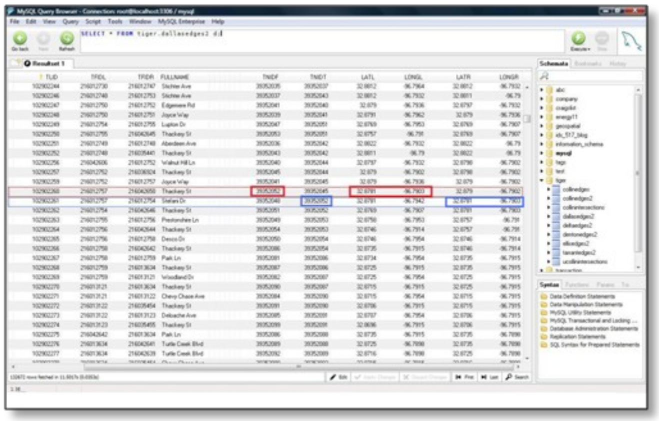


Fig 4: showing the creation of intersection database



SELECT * FROM tiger.dallasedges2 d;

TLID	TFIDL	TFIDR	FULLNAME	TNIDF	TNIDT	LATL	LONGL	LATR	LONGR
102902244	[illegible]	[illegible]	Stichter Ave	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902246	[illegible]	[illegible]	Stichter Ave	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902247	[illegible]	[illegible]	Edgemere Rd	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902248	[illegible]	[illegible]	Joyce Way	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902249	[illegible]	[illegible]	Lupton Dr	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902250	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902251	[illegible]	[illegible]	Aberdeen Ave	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902252	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902256	[illegible]	[illegible]	Walnut Hill Ln	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902257	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902259	[illegible]	[illegible]	Joyce Way	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902260	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902261	[illegible]	[illegible]	Stefani Dr	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902262	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902263	[illegible]	[illegible]	Prestonshire Ln	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902264	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902265	[illegible]	[illegible]	Desco Dr	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902266	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902267	[illegible]	[illegible]	Park Ln	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902268	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902269	[illegible]	[illegible]	Woodland Dr	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902270	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902271	[illegible]	[illegible]	Chevy Chase Ave	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902272	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902273	[illegible]	[illegible]	Debache Ave	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902274	[illegible]	[illegible]	Thackery St	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902275	[illegible]	[illegible]	Park Ln	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902276	[illegible]	[illegible]	Turtle Creek Blvd	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
102902277	[illegible]	[illegible]	Turtle Creek Blvd	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Fig 5 : showing two streets (Thackery St. and Stefani Dr.) intersecting and hence having a common Node ID

We perform this pre-processing, check the database for two different named streets having one node in common. We also store the geometric coordinates of the intersection to be able to identify an intersection. Algorithm 3 describes the algorithm to find the intersection from the street-weight vector.

Algorithm 3 Intersect (S, W)

Input: (S, W): Streets and weights vector

Output: (S_1, S_2, R): Two intersecting streets and R is true for intersection; false otherwise

```
1:  $S_1 \leftarrow Street_{max-weight}$ (S, W)
2:  for each $S_0 \varepsilon S$
3:      $S_2 \leftarrow \{ S_0 : S_0 \varepsilon S, S_0 \neq S_1$ and $|W(S_0) - W(S_1)| < W_{min-diff} \}$
4:      if TIGER_INTERSECT ($S_1$, $S_2$) == true then
            return ($S_1$, $S_2$, true)
5:  return (null, null, false)
```

In the first step, select the street with the maximum weight as S_1. Then, among the remaining street concepts it looks for a street-concept, S_2 that is different from S_1, but has similar weight and is in close proximity to it. It then checks the intersection database for an intersection. If there is an intersection, it returns true along with the names of the two streets; false otherwise.

E.g. City carries 10 points, state 5 and a street name carries 15 points. For the keyword "Campbell", consider the concept of {Street} Campbell St. / {City} Dallas/ {County} Dallas/ {State} Texas/ {Country} USA. The concept gets 15 points because Campbell is a street name, and it gets an additional 10 points if Dallas is also mentioned in the text. In phase 2, we consider the relation between two keywords. Considering the previous example, if {Campbell St., Dallas} are the keywords appearing in the text, then amongst the various concepts listed for "Campbell" would be {Street} Campbell St./{City} Dallas/{County} Dallas/ {State} Texas/{Country} USA and one of the concepts for "Dallas" would be {City} Dallas/{County} Dallas/{State} Texas/ {Country} USA. Now, in phase 2 we check for such correlated concepts, in which one concept subsumes the other. In that case the more specific concept gets the boosting from the more general concept. Here, the above mentioned Dallas concept boosts up the more specific Campbell concept. After the two phases our complete we re-order the concepts in descending order of their weights.

Next, we prune out the concepts where the county in the domain. E.g. the domain of Dallas-Fort Worth area would comprise of 12 counties. If we have a concept having the county to be not amongst those twelve, we remove it. The next step involves boosting of a concept based on what occurs just before the keyword in the original ad. E.g. if it is a sequence of digits, or words like "at", "near", "close to", "around", it increases the possibility of the keyword being a street name we boost up the weight accordingly. Another heuristic technique we use is the reverse phone number lookup. If the ad has a phone number, we do a reverse phone number lookup using White pages and for published numbers get the address. If the address also mentions Campbell, we further boost up the weight of the street-concepts having Campbell as the street name. Now, we send the vector having all the streets and their weights to the Intersect (S, W) and check for an intersection. If the street-weight vector has Campbell Rd. with the highest weight and Coit Rd. with comparable weight and the intersection database has an entry with Coit and Campbell, the intersection is returned as location.

CONFIDENCE-ACCURACY FACTOR

After we have identified and disambiguated the highest weighing concept from the text, which refers to the street name or city name, we calculate its Confidence-Accuracy Factor (CAF). CAF, a number between 0 and 1, is a measure of the accuracy and the confidence of the apartment's location. Accuracy defines the exactness, or correctness we have of the location. e.g. a street name will have a lower accuracy as compared to street-intersection which will have a lower accuracy as compared to an address with a house number and street name. The confidence part of CAF describes the source of the location. It ascertains the belief in correctness of the source. Hence, an address obtained from the reverse phone number lookup will have a low confidence as compared to a Google/Yahoo maps link. Depending on the CAF value, we map the apartment either as a cloud for low, a dart for medium and a house for high confidence-accuracy factor.

$CAF = CAF_{confidence} + CAF_{accuracy}$

$CAF_{confidence} = \sum \alpha_i / (2 * W_{max})$

Where, α_i is the confidence factor of the source and W_{max} is the maximum confidence (e.g. a google/yahoo link with a phone no. which verifies it). $\alpha_{google/yahoo} > \alpha_{disambiguation\text{-}algo} > \alpha_{reverse\text{-}phone}$

$CAF_{accuracy} = \beta_i / (2 * \beta_{max})$

Where β_i is the accuracy factor and β_{max} is the accuracy value for a location with block number, street and city (most accurate). $\beta_{block\text{-}street\text{-}city} > \beta_{intersection\text{-}city} > \beta_{street\text{-}city} > \beta_{city}$

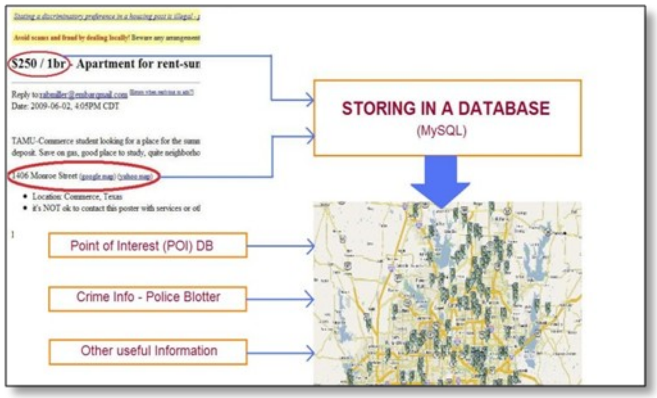

$250 / 1br - Apartment for rent-sum
1406 Monroe Street
STORING IN A DATABASE
(MySQL)
Point of Interest (POI) DB
Crime Info - Police Blotter
Other useful Information

Fig. 6: Flowchart showing the working of the Mash up

INTEGRATION OF DATA SOURCES

While looking for an apartment, apart from the basic things like rent, location etc., the user is also interested in other facts like the safety of the neighborhood, the nearby public amenities like parks, schools etc. Fig4. shows the flowchart describing this integration. From the ad we extract the information like rent, location, number of bedrooms and bathrooms and store it in a relational database. We also have location based information like crime, point of interests stored. In this we describe how we collect, analyze and integrate this information and show it on the map in a way that makes more meaning to the user.

6.1 Points of Interest (POI)

Points of Interest refer to the various specific point locations that someone may find useful or interesting. We had a comprehensive database of the over 300 POIs, provided by Homeland Security Information Program (HSIP). Amongst them, we selected 10 that pertained to the interests of someone looking for an apartment. These included grocery stores, places of worship, parks, gas stations, schools etc. So when a user is looking an apartment apart from the basic things like rent, no. of bedrooms he also can see the nearest POIs on the same map.

To search the database for the nearest POIs to the potential apartment we use the **R-tree** indexing. R-trees are typically the preferred method for indexing spatial data. Objects (shapes, lines and points) are grouped using the minimum bounding rectangle (MBR). Objects are added to an MBR within the index that will lead to the smallest increase in its size. We create a minimum bounding rectangle for each city which allows for efficient and effective query processing, which is one of the key aspects of the application.

6.2 Crime Statistics and other Information

Safety of the neighborhood is a key considering while someone is looking for an apartment. For this reason, we try to provide certain pointers for the type of neighborhood such as the crime rating, median family income and the percentage of high school graduates. For the crime rating, we periodically scan the police blotter, aggregate the information, and present it in a scale from 0 to 10 so that it makes more sense to the user. Similarly we integrate the median family income and percentage of high school graduates and display it along with other details of the apartment.

Fig 9 shows a potential apartment with nearest gas stations and hospitals. Clicking on the apartment icon gives other information about it including location, rent, bedrooms, crime rating, median family income, high school graduates and the link to the ad on Craigslist.

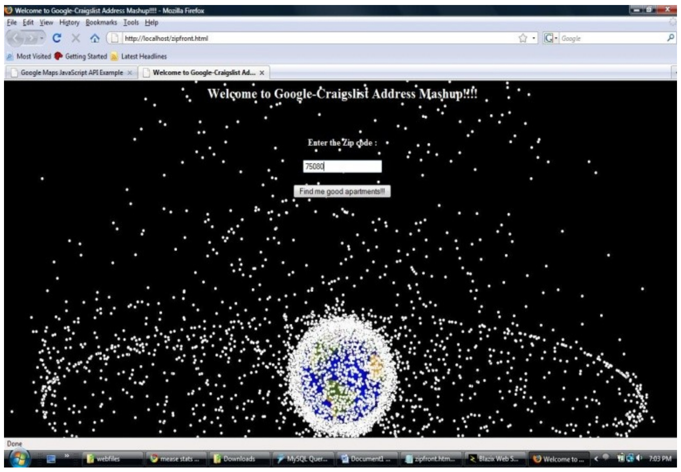

Welcome to Google-Craigslist Address Mashup!!!
Enter the Zip code :
75080
Find me good apartments!!!

Fig 7: Screenshot #1 shows the front page of the application

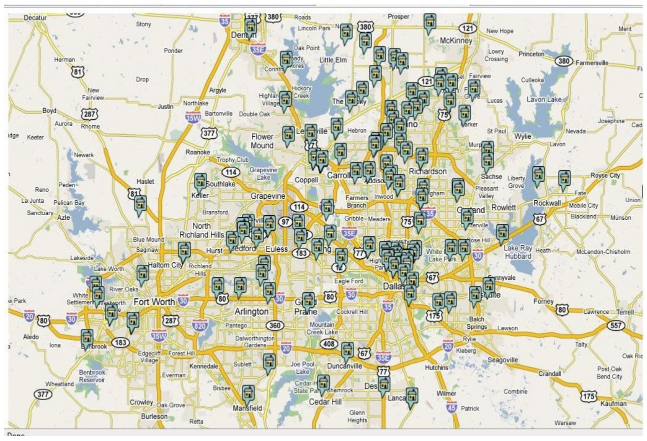
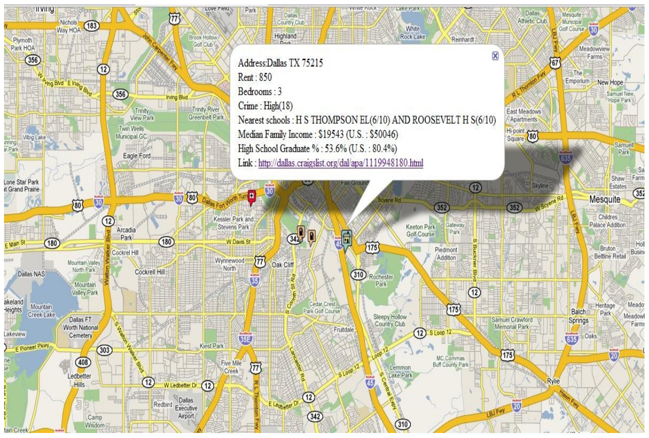


Fig 8: Screenshot #2 shows all apartments for the Dallas-Fort Worth area



Address:Dallas TX 75215
Rent : 850
Bedrooms : 3
Crime : High(18)
Nearest schools : H S THOMPSON EL(6/10) AND ROOSEVELT H S(6/10)
Median Family Income : $19543 (U.S. : $50046)
High School Graduate % : 53.6% (U.S. : 80.4%)
Link : http://dallas.craigslist.org/dal/apa/1119948180.html

Fig 9: Screenshot #3 showing a potential apartment with information and nearby hospitals and gas-stations.

RELATED WORK

The problem of geographic location identification and disambiguation has been dealt with mostly two approaches. One involving the concepts of machine learning and NLP and the other using data mining approach with the help of gazetteers.

In NLP and machine learning a lot of previous work is done on the more general topic of Named Entity Recognition (NER). Most of the work makes use of structured and well-edited text from news articles or sample data from the conferences.

Most research work relies on NLP algorithms and less on machine learning techniques. The reason for this is that machine learning algorithms require training data that is not easy to obtain. Also, their complexity makes them less efficient as compared to the algorithms using the gazetteers.

Other researchers use a 5-step algorithm, where the first two steps of the algorithm are reversed. First, only terms appearing in the gazetteer are short listed. Next, they use NLP techniques to remove the non-geo terms. Li *et al* [6] report a 93.8% precision on news and travel guide data.

McCurley [8] analyzes the various aspects of a web page that could have a geographic association, from its URL, the language in the text, phone numbers, zip codes etc. Names appearing the text may be looked up in White Pages to determine the location of the person. His approach is heavily dependent on information like zip codes etc. and is hence successful in USA, where it is available free but is hard to obtain for other countries. Their techniques rely on heuristics and do not consider the relationship between geo-locations appearing in text.

The gazetteer based approach relies on the completeness of the source and hence cannot identify terms that are not present in the gazetteer. But on the other hand they are less complex than NLP, machine learning techniques are hence faster.

Amitay *et al.* [7] present a way of determining the page focus of web pages using the gazetteer approach and after using techniques to prune the data. They are able to correctly tag individual name place occurrences 80% of the time and are able to recognize the correct focus of the pages 91% of the time. But they have a low accuracy for the geo/non geo disambiguation.

Lieberman *et al. [9]* describe the construction of a spatio-textual search engine using the gazetteer and NLP tools, a system for extracting, querying and visualizing textual references to geographic locations in unstructured text documents. They use an elaborate technique for removing the stop words, using a hybrid model of *Part-of-Speech (POS)* and *Named-Entity Recognition* tagger. POS helps to identify the nouns and NER tagger annotates them as person, organization, and location. They consider the proper nouns tagged as locations. But this system doesn't work well for text where name of a person is ambiguous with a location. E.g. Jordan might mean Michael Jordan, the basketball player or it might mean the location. In that case the NER tagger might remove Jordan considering it to be name of a person. For removing geo-geo ambiguity they use the *pair strength* algorithm. Pairs of feature records are compared to determine whether or not they give evidence to each other, based on the familiarity of each location, frequency of each location, as well as their document and geodesic distances. They do not report any results for accuracy of the algorithm so comparison and review is not possible.

Craigslist acts as a medium for realtors and owners/renters for free and easy interaction. Existing apartment lookup sites are not dynamic and will not show an apartment or house that has been vacated very recently and is for rent/sale. Nor will it show any special deals that the realtor is offering. Previous attempts to mash Craigslist and Google maps [10, 11, and 12], focus only on the graphical interface and functionality, and lack a sophisticated location extraction and hence are not able to display it or fail to do so accurately. These sites only display the listings having the Google/Yahoo maps link in it and use no geo-tagging technique. Other sites such as http://www.allurstuff.com [13] do a good job of giving a measure of accuracy but pick only the address that comes after the term "Location" in the advertisement. In addition to these there is no site that integrates data sources to provide useful other meaningful information like crime, points of interest etc.

RESULTS

We used a dataset comprising of 2500 randomly chosen ads from the Craigslist website for one day for the Dallas Fort Worth Listings. We then removed the HTML content and cleaned the data of text that was recurring in each ad but was irrelevant for all purposes. We then geo-tagged these first by looking for Google/Yahoo maps link, a direct physical address or reverse phone lookup in them. This portion of the results guarantees 100% accuracy. Fig 8 shows the internal distribution of output.

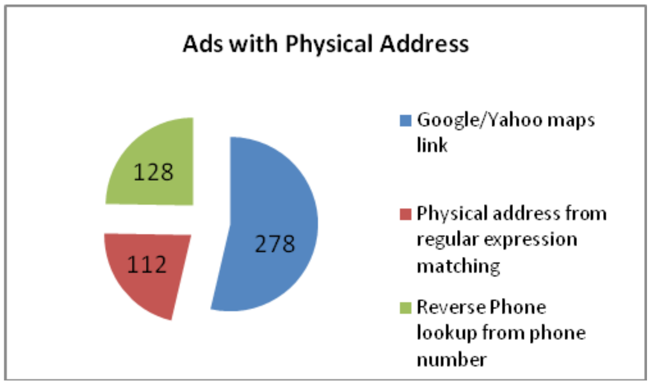


Fig 10: The internal distribution of ads with physical address

More than half of these had a map link; rest was almost equally divided between the exact physical address and the reverse phone number lookup. Then, for the remaining we applied our Disambiguation algorithm and then manually checked the geo-tags for correctness. The algorithm either returned a location or in the absence of a high weight street returned "null" indicating the absence of a street name. For those ads for which the algorithm returned "null", we tried to find the neighborhood from the list of neighborhoods gathered from crawling Wikipedia.

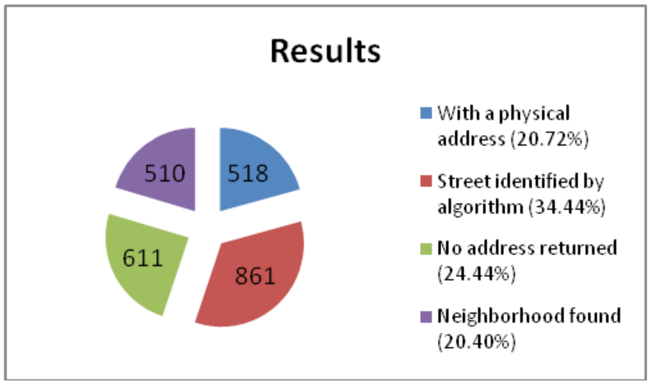


Fig 11: Results of the Craigslist-Google Maps System

Fig 9 shows the results of our approach. For 518 (20.72%) documents we were able to either get an address from Google/Yahoo map link or physical address from regular expression matching or White pages reverse phone lookup. Next, on this set of 1982 we used the algorithm, for which, the algorithm returned street names for 861 (34.44%) and returned "null" for the remaining 1121. Finally, we applied our "neighborhood determining" technique to successfully identify neighborhoods for 20.40% ads. For the remaining 24.44% we still do not have an address and hence we cannot plot on the map.

We then tested the correctness of our algorithm by manually annotating the entire set of documents and comparing it with the results. Fig 10 shows the Precision, Recall and F-measure values for the dataset using different approaches.

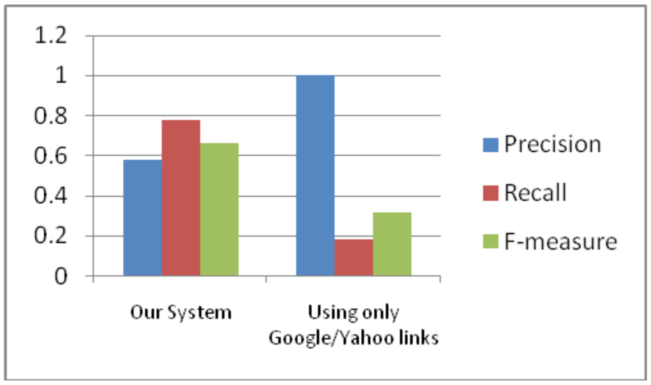


Fig 12: shows the Precision and Recall values of our system compared to just Google/Yahoo Maps-link approach.

Using our approach we get a precision of 0.577, recall of 0.776 and an F-measure of 0.662. Obviously, precision value for the Google/Yahoo maps is 1, but clearly the recall value for out approach is much higher as compared to the simple Google/Yahoo links approach showing a 3 times more coverage. The coverage is even lesser for other known websites which create a Craigslist-Google Maps mash-up.

CONCLUSION AND FUTURE WORK

We developed an apartment searching tool, which takes the Craigslist ads as the source and shows them on Google Maps, integrated with other services such as crime ratings, points of interest etc making it easier for the user to come to a decision. We also make use of a disambiguation algorithm to correctly identify the location of the apartment and to increase the coverage. With each apartment we also associate a CAF value to give the user an idea the confidence and accuracy we have in the correct positioning of the location.

The results show a significant increase in the coverage as compared to other sites. Since the data is so unstructured and the annotation of street names is a difficult task, there is a still a segment of ads for which we still couldn't find a location. In future we would like to extend our system to increase the coverage by improving the algorithm. Other future work includes improving the GUI, making the system more user oriented by giving him preferences to narrow down the search, and include user reviews.

REFERENCES

[1] Hover's Craigslist - Company Overview.

[2]CRF Tagger: http://sourceforge.net/projects/crftagger/

[3] TIGER gazetteer : http://www.census.gov/geo/www/tiger/

[4] ZIP code statistics by U.S. Census Bureau : http://www.census.gov/epcd/www/zipstats.html

[5] E. Amitay, N. Har'El, R. Sivan, and A. Soffer. Web-a-Where:geo-tagging web content. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* pages 273-280, Sheffield, UK, July 2004.

[6] H. Li, R. K. Srihari, C. Niu, and W. Li. Location normalization for information extraction. In *Proceedings of the 19th International Conference on Computational Linguistics*, pages 1{7, Taipei, Taiwan, Aug. 2002.

[7] D. A. Smith and G. Crane. Disambiguating geographic names in a historical digital library. In *Proceedings of the 5th European Conference on Research and Advanced Technology for Digital Libraries (ECDL'01), Lecture Notes in Computer Science, pages 127–136, Darmstadt, September 2001. Springer.*

[8] K. S. McCurley. Geospatial mapping and navigation of the web. In *Proceedings of the 10th int. conference on World Wide Web, pages 221–229. ACM Press, 2001.*

[9] Michael D. Lieberman, Hanan Samet, Jagan Sankaranarayanan, and Jon Sperling: STEWARD: architecture of a spatio-textual search engine. In *Proceedingsof the 15th International Symposium on Advances in Geographic Information Syst*ems. (ACM GIS 2007).

[10]Padmapper : http://www.padmapper.com

[11] Housingmaps : http://www.housingmaps.com

[12] MapsKreig : http://www.apskreig.com

[13]Allurstuff : http://www.allurstuff.com

[14] J. L. Leidner, G. Sinclair, and B. Webber. Grounding spatial named entities for information extraction and question answering. In *Proceedings of the HLT-NAACL 2003 Workshop on Analysis of Geographic References.*

[15] D. Wu, G. Ngai, M. Carpuat, J. Larsen, and Y. Yang : Boosting for named entity recognition. In *Proceedings of the 6th Conference on Natural Language Learning.*