RDF Data-Centric Storage

Justin J. Levandoski Mohamed F. Mokbel
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN
{justin,mokbel@cs.umn.edu}

***Abstract*—The vision of the Semantic Web has brought about new challenges at the intersection of web research and data management. One fundamental research issue at this intersection is the storage of the Resource Description Framework (RDF) data: the model at the core of the Semantic Web. We present a *data-centric* approach for storage of RDF in relational databases. The intuition behind our approach is that each RDF dataset requires a *tailored* table schema that achieves efficient query processing by (1) reducing the need for joins in the query plan and (2) keeping null storage below a given threshold. Using a basic structure derived from the RDF data, we propose a two-phase algorithm involving *clustering* and *partitioning*. The *clustering* phase aims to reduce the need for joins in a query. The *partitioning* phase aims to optimize storage of extra (i.e., null) data in the underlying relational database. Our approach does not assume a particular query workload, relevant for RDF knowledge bases with a large number of ad-hoc queries. Extensive experimental evidence using three publicly available real-world RDF data sets (i.e., DBLP, DBPedia, and Uniprot) shows that our schema creation technique provides superior query processing performance compared to state-of-the art storage approaches. Further, our approach is easily implemented, and complements existing RDF-specific databases.**

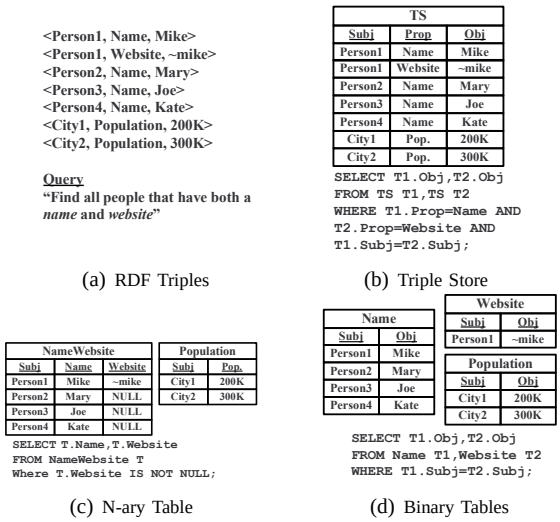
I. Introduction

Over the past decade, the W3C [1] has led an effort to build the Semantic Web. The purpose of the Semantic Web is to provide a common framework for data-sharing across applications, enterprises, and communities [2]. By giving data semantic *meaning* (through metadata), this framework allows machines to consume, understand, and reason about the structure and purpose of the data. In this way, the Semantic Web resembles a worldwide database, where humans or computer agents can pose semantically meaningful queries and receive answers from a variety of distributed and distinct sources. The core of the Semantic Web is built on the Resource Description Framework (RDF) data model. RDF provides a simple syntax, where each data item is broken down into a $<subject, property, object>$ triple. The *subject* represents an entity instance, identified by a Uniform Resource Identifier (URI). The *property* represents an attribute of the entity, while the *object* represents the value of the *property*. As a simple example, the following RDF triples model the fact that a person John is a reviewer for the conference ICWS 2009:

```
person1 hasName ''John''
confICWS09 hasTitle ''ICWS 2009''
person1 isReviewerFor confICWS09
```

While the ubiquity of the RDF data model has yet to be realized, many application areas and use-cases exist for RDF, such as education [3], mobile search environments [4], social networking [5], and biology and life science [6], making it an emerging and challenging research domain.

This work is supported in part by the National Science Foundation under Grants IIS0811998, IIS0811935, and CNS0708604



(a) RDF Triples (b) Triple Store (c) N-ary Table (d) Binary Tables

Fig. 1. RDF Storage Example

Efficient and scalable management of RDF data is a fundamental challenge at the core of the Semantic Web. Many popular RDF storage solutions use relational databases to achieve this scalability and efficiency. To illustrate, Figure 1(a) gives a sample set of RDF triples for information about four people and two cities, along with a simple query that asks for people with both a *name* and *website*. Figures 1(b) - 1(d) give three possible approaches to storing these sample RDF triples in a DBMS, along with the translated RDF queries given in SQL. A large number of systems use a *triple-store* schema [7], [8], [9], [10], [11], [12], where each RDF triple is stored directly in a three-column table (Figure 1(b)). This approach provides inefficient query execution due to a proliferation of *self-joins*, an expensive operation whereby the triple-store is joined against itself. Another approach is the *property table* [10], [13] (Figure 1(c)) that models multiple RDF properties as n-ary table columns. The n-ary table eliminates the need for a join in our example query. However, as only one person out of four has a website, the n-ary table contains a high number of nulls (i.e., the data is semi-structured), potentially causing a high overhead in query processing [14]. The *decomposed storage* schema [15] (Figure 1(d)) stores triples for each RDF property in a binary table. The binary table approach reduces null storage, but introduces a join in our example query.

In this paper, we propose a new storage solution for RDF data that aims to avoid the drawbacks of these previous approaches. Our approach is *data-centric*, as it tailors a relational schema based on a derived *structure* of the RDF data with

the explicit goal of providing efficient query performance. We achieve this goal by taking into account the following trade-off of expensive query processing operations in relational databases: (1) reducing, on average, the need to join tables in a query by storing as much RDF data together as possible, and (2) reducing the need to process extra data by tuning extra storage (i.e., null storage) to fall below a given threshold. To handle these trade-offs, our approach involves two phases: *clustering* and *partitioning*. The *clustering* phase scans the RDF data to find groups of related properties (i.e., properties that always exist *together* for a large number of subjects). Properties in a *cluster* are candidates for storage together in an n-ary table. Likewise, properties *not* in a cluster are candidates for storage in binary tables. The *partitioning* phase takes clusters from the clustering phase and balances the trade-off between storing as many RDF properties in clusters as possible while keeping null storage to a minimum (i.e., below a given threshold). Our approach also handles cases involving *multi-valued* properties (i.e., properties defined multiple times for a single subject) and *reification* (i.e., extra information attached to a whole RDF triple). The output of our schema creation approach can be considered a balanced mix of binary and n-ary tables based on the structure of the data.

The performance of our data-centric approach is backed by experiments on three large publicly available real-world RDF data sets: DBLP [16], DBPedia [17], and Uniprot [6]. Each of these data show a range of schema needs, and a set of benchmark queries are used to show that our data-centric schema creation approach improves query processing compared to previous approaches. Results show that our data-centric approach achieves orders of magnitude performance improvement over the triple store, and speedup factors of up to 36 over a straight binary table approach.

II. Related Work

Previous approaches to RDF storage have focused on three main categories. (1) The *triple-store* (Figure 1(b)). Relational architectures that make use of a *triple-store* as their *primary* storage scheme include Oracle [10], Sesame [9], 3-Store [11], R-Star [12], RDFSuite [7], and Redland [8]. (2) The *property table* (Figure 1(c)). Due to the proliferations of *self-joins* involved with the *triple-store*, the *property table* approach was proposed. Architectures that make use of *property tables* as their *primary* storage scheme include the Jena Semantic Web Toolkit [13]. Oracle [10] also makes use of property tables as *secondary* structures, called materialized join views (MJVs). (3) The *decomposed storage model* [18] (Figure 1(d)) has recently been proposed as an RDF storage method [15], and has been shown to scale well on column-oriented databases, with mixed results for row-stores. Our work distinguishes itself from previous work as we (1) provide a *tailored* schema for each RDF data set, using a balance between n-ary tables (i.e., *property tables*) and binary tables (i.e.,*decomposed storage*), and (2) provide an *automated* method to place properties together in tables based on the structure of the data. Previous approaches to building property tables have involved the use of generic pre-computed joins, or construction by a DBA with knowledge of query usage statistics [10]. It is important to note, however, that our schema creation technique is *system-independent*, and can *complement* any related full-fledged RDF-specific databases (e.g., see [7], [10], [8], [11]).

Other work in RDF storage has dealt with storing pre-computed *paths* in a relational database [19], used to answer *graph* queries over the data (i.e., connection, shortest path). Other graph database approaches to RDF, including extensions to RDF query languages to support *graph* queries, have been proposed [20]. This work is outside the scope of this paper, as we do not study the effect of graph queries over RDF.

Automated relational schema design has primarily been studied with the assumption of *query workload statistics*. Techniques have been proposed for index and materialized view creation [21], horizontal and vertical partitioning [22], [23], and partitioning for large scientific workloads [24]. Our automated data-centric schema design method for RDF differs from these approaches in two main ways. (1) Our method does *not* assume a set of query workload statistics, rather, we base our method on the structure found in RDF data. We believe this is a reasonable assumption given the fact that queries over RDF knowledge bases tend to be ad-hoc. (2) Previous schema creation techniques cannot take into account the heterogeneous nature of RDF data, i‘.e., table design that balances its schema between well-structured and semi-structured data sets.

III. Overview & Problem Definition

Overview. In general, two modules exist outside the database engine to handle RDF data and queries: (1) an RDF import module, and (2) an RDF query module. Our proposed data-centric schema creation technique exists inside the RDF import module. The schema creation process takes as input an RDF data set. The output of our technique is a schema (i.e., a set of relational tables) used to store the imported RDF data in the underlying DBMS.

Problem Definition. *Given a data set of RDF triples, generate a relational table schema that achieves the following criteria. (1) Maximize the likelihood that queries will access properties in the same table without causing a join and (2) Minimize the amount of unnecessary data processing by reducing extra data storage (e.g., null storage).*

Join operations along with extra table accesses produce a large query processing overhead in relational databases. Our schema creation method aims to achieve the *first* criterion by explicitly aiming to *maximize* the amount RDF data stored *together* in n-ary tables. However, as we saw in the example given in Figure 1, n-ary tables can lead to extra storage that affects query processing. Thus, our schema creation method aims to achieve the *second* criterion by keeping the *null storage* in each table *below* a given threshold.

IV. Data-Centric Schema Creation

The intuition behind our approach is that different RDF data sets require different storage structures. For example, a relatively *well-structured* RDF data set (i.e., data where the majority of relevant RDF properties are defined for the

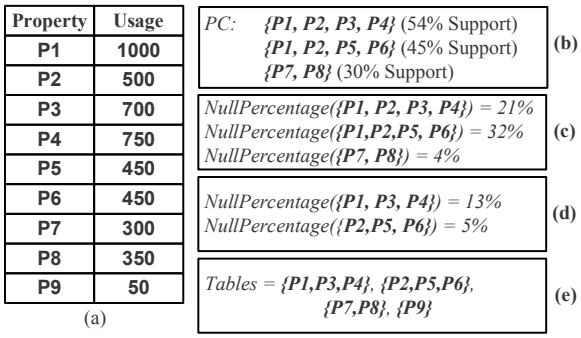


Fig. 2. RDF Data Partitioning Example

subjects) may require a few large n-ary tables. A *semi-structured* data set (i.e., data that does not follow a fixed pattern for property definition) may use a large number of binary tables as its primary storage schema.

A. Algorithm Overview and Data Structures

Our schema creation algorithm takes as parameters an RDF data set, along with two numerical values: *support threshold* and *null threshold*. *Support threshold* is a value used to measure strength of correlation between properties in the RDF data. If a set of properties meets this threshold, they are candidates to exist in the same n-ary table. The *null threshold* is the percentage of null storage tolerated for each table in the schema.

The data structures for our algorithm are built using an O(n) process that scans the RDF triples once (where n is the number of RDF triples). We maintain two data structures: (1) *Property usage list.* This is a list structure that stores, for each property defined in the RDF data set, the count of subjects that have that property defined. For example, if a property usage list were built for the data in Figure 1(a), the property *Website* would have a usage count of one, as it is only defined for the subject *Person1*. Likewise, the *Name* property would have a usage count of four, and *Population* would have a count of two. (2) *Subject-property baskets.* This is a list of all RDF subjects mapped to their associated properties (i.e., a property basket). A single entry in the subject-property basket structure takes the form $subjId \rightarrow \{prop_1, \cdots, prop_n\}$, where $subjId$ is the Uniform Resource Identifier of an RDF subject and its property basket is the list of all properties defined for that subject. As an example, for the sample data in Figure 1(a), six baskets would be created by this process: $Person1 \rightarrow \{Name, Website\}$, $Person2 \rightarrow \{Name\}$, $Person3 \rightarrow \{Name\}$, $Person4 \rightarrow \{Name\}$, $City1 \rightarrow \{Population\}$, $City2 \rightarrow \{Population\}$.

Our schema creation algorithm involves two phases: *clustering* and *partitioning*. The *clustering* phase (Phase I) aims to find groups of related properties in the data set using the *support threshold* parameter. Clustering leverages previous work from association rule mining to find related properties in the data. The idea behind the clustering phase is that properties contained in the clusters should be stored in the same n-ary table. The canonical argument for n-ary tables is that related properties are likely to be queried together. Thus, storing related properties together in a single table will reduce the number of joins during query execution. The clustering phase also creates an initial set of final tables. These initial tables consist of the properties that are *not* found in the generated clusters (thus being stored in binary tables) and the property clusters that do *not* need partitioning (i.e., in Phase II). The *partitioning* phase (Phase II) takes the clusters from Phase I and ensures that they contain a *disjoint* set of properties while keeping the null storage for each cluster below a given threshold. The final schema is the union of tables created from Phase I and II.

B. Phase I: Clustering

The clustering phase involves two steps. *Step 1:* A set of clusters (i.e., related properties) are found by leveraging the use of *frequent itemset* finding, a method used in association rule mining [25]. For our purposes, the terms *frequent itemsets* and *clusters* are used synonymously. The *clustering phase* finds groups of properties that are found *often* in the *subject-property basket* data structure. The measure of how often a cluster occurs is called its *support*. Clusters with high support imply *many* RDF subjects have *all* of the properties in the cluster defined. In other words, high support implies that properties in a cluster are *related* since they often exist *together* in the data. The metric for high support is set by the support threshold parameter to our algorithm, meaning we consider a group of properties to be a cluster *only if* they have support greater than or equal to the support threshold. If we specify a *high* support threshold, the clustering phase will produce a small number of small clusters with highly correlated properties. For *low* support threshold, the clustering phase will produce a greater number of large clusters, with less-correlated properties. Also, for our purposes, we are only concerned with *maximum sized* cluster (or *maximum frequent itemsets*); these are the clusters that occur often in the data *and* contain the *most* properties, meaning we maximize the data stored in n-ary tables. It is important to note that maximum frequent itemset generation can produce clusters with *overlapping* properties.

Step 2: Construct an initial set of final tables. These tables contain (1) properties that are *not* found in generated clusters (thus being stored in binary tables) and (2) the property clusters that *meet* the null threshold and do *not* contain properties that *overlap* with other clusters, thus not necessitating Phase II. Clusters that are added to the initial final table list are *removed* from the cluster list. The output of the *clustering phase* is a list of initial final tables, and a set of clusters, *sorted* in decreasing order by their support value, that will be sent to the *partitioning* phase.

Example. Consider an example with a support threshold of 15%, null threshold of 20% for the six subject-property baskets given in Section IV-A for the data in Figure 1(a). In this case we have four possible property clusters: $\{Name\}$, $\{Website\}$, $\{Population\}$, and $\{Name, Website\}$. The cluster $\{Name\}$ occurs in 4 of the 6 property baskets, giving it a support of 66%, while the cluster $\{Name, Website\}$ occurs in 1 of 6 property baskets, giving it a support of 16%. In this case, the $\{Name, Website\}$ is generated as a cluster, since

Algorithm 1 Clustering

1: **Function** Cluster(Baskets B,Usage PU,$Thresh_{sup}$,$Thresh_{null}$)
2: $Clusters \leftarrow$ GetClusters($B, Thresh_{sup}$)
3: $Tables \leftarrow$ properties not in PC /* Binary tables */
4: **for all** $c_1 \in PC$ **do**
5: $OK \leftarrow false$
6: **if** $Null\%(c_1, PU) \leq Thresh_{null}$ **then**
7: $OK \leftarrow true$
8: **for all** $c_2 \in PC$ **if** $c_1 \cap c_2 \neq \phi$ **then** $OK \leftarrow false$
9: **end if**
10: **if** OK **then** $Tables \leftarrow Tables \cup c_1$; $Clusters \leftarrow Clusters - c_1$
11: **end for**
12: **return** $Tables$,$Clusters$

it meets the *support threshold* and has the *most* possible properties. Note the single property $\{Population\}$ is not considered a cluster, and would be added to the initial final table list. Also, Figure 2(b) gives three example clusters along with their support values, while Figure 2 (c) gives their null storage values (null storage calculation is covered shortly). The output of the clustering phase in this example with a support and null threshold value of 20% would produce an initial final table list containing $\{P9\}$ (not contained in a a cluster) and $\{P7, P8\}$ (not containing overlapping properties and meeting the null threshold). The set of output clusters to be sent to the next phase would contain $\{P1, P2, P3, P4\}$ and $\{P1, P2, P5, P6\}$.

Algorithm. Algorithm 1 gives the pseudocode for the clustering phase, and takes as parameters the subject-property baskets (B), the property usage list (PU), the support threshold ($Thresh_{sup}$), and the null threshold parameter ($Thresh_{null}$). *Step 1* of the algorithm generates clusters, sorts them by support value, and stores them in list *Clusters* (Line 1 in Algorithm 1). This is a direct call to a maximum frequent itemset algorithm [25], [26]. *Step 2* of the algorithm is performed by first setting a list $Tables$ to all binary tables returned from step 1. $Tables$ is then expanded by adding from list *Clusters* the clusters that do not contain overlapping properties that are below the given null threshold (Lines 1 to 1 in Algorithm 1). Calculation of the null storage for a cluster c is performed using the property usage list PU. Let $|c|$ be the number of properties in a cluster, and $PU.maxcount(c)$ be the maximum property count for a cluster in PU. As an example, in Figure 2 (b) if $c = \{P1, P2, P3, P4\}$), $|c| = 4$ and $PU.maxcount(c) = 1000$ (corresponding to $P1$). If $PU.count(c_i)$ is the usage count for the ith property in c, the *null storage percentage* for c is:

$$Null\%(c) = \frac{\sum_{\forall i \in c}(PU.maxcount(c) - PU.count(c_i))}{(|c| + 1) * PU.maxcount(c)}$$

Finally, the algorithm returns the initial final table list and remaining clusters (Line 1 in Algorithm 1).

C. Phase II: Partitioning

The objective of the *partitioning* phase is twofold: (1) Partitioning the given clusters (from Phase I) into a set of non-overlapping clusters (i.e., a property exists in a *single* n-ary table). Ensuring that a property exists in a single cluster reduces the number of table accesses and unions necessary in query processing. For example, consider two possible n-ary tables storing RDF data for publications: $TitleConf = \{subj, title, conference\}$ and $TitleJourn = \{subj, title, journal\}$. An RDF query asking for all published titles would involve two table accesses and a union, since titles exist in both the conference and journal tables. (2) Ensuring that each partitioned cluster, falls below the null storage threshold. Reducing null storage tunes the schema for efficient query processing.

We propose a greedy algorithm that attempts to keep the cluster with *highest* support intact, while *pruning* lower-support clusters containing overlapping properties. The intuition behind the greedy approach is that clusters with *highest* support contain properties that occur together *most often* in the RDF data. Support is the percentage of RDF subjects that have *all* of the cluster's properties. Keeping high support clusters intact implies that the most RDF subjects (with the cluster's properties defined) will be stored together in the table. Our greedy approach continually considers the *highest* support cluster, and handles two cases based on its null storage computation (from Section IV-B). *Case 1*: the cluster meets the null storage threshold, meaning the given cluster from Phase I meets the null threshold but contains overlapping properties. In this case, the cluster is considered a table and all lower-support clusters with overlapping properties are pruned (i.e., the overlapping properties are removed from these lower-support clusters). We note that pruning will likely create *overlapping* cluster fragments; these are clusters that are *no longer* maximum sized (i.e., *maximum frequent* itemsets) and contain similar properties. To illustrate, consider a list of three clusters $c_1 = \{A, B, C, D\}$, $c_2 = \{A, B, E, F\}$, and $c_3 = \{C, E\}$ such that $support(c_1) > support(c_2) > support(c_3)$. Since our greedy approach chooses c_1 as a final table, pruning creates overlapping cluster fragments $c_2 = \{E, F\}$ and $c_3 = \{E\}$. In this case since $c_3 \subseteq c_2$, these clusters can be *combined* during the pruning step. Thus, we *merge* any overlapping fragments in the cluster list. *Case 2*: the cluster does *not* meet the null storage threshold. Thus, it is *partitioned* until it meets the null storage threshold. The *partitioning* process repeatedly removes the property p from the cluster that causes the *most* null storage until it meets the null threshold. Removing p maximally reduces the null storage in one iteration. Further, support for clusters is *monotonic*: given two clusters c_1 and c_2, $c_1 \subseteq c_2 \Leftarrow support(c_1) \geq support(c_2)$. Thus, the partitioned cluster will still meet the given support threshold. After removing p, two cases are considered. *Case 2a*: p exists in a lower-support cluster. Thus, p has a chance of being kept in a n-ray table. *Case 2b*: p does *not* exist in a lower-support cluster. This is the worst case, as p must be stored in a binary table. Once the cluster is partitioned to meet the null threshold, it is considered a table and all lower-support clusters with overlapping properties are pruned.

Example. From our running example in Figure 2, two clusters would be passed to the partitioning phase: $\{P1, P2, P3, P4\}$ and $\{P1, P2, P5, P6\}$. The cluster $\{P1, P2, P3, P4\}$ has the highest support value (as given in Figure 2 (b)), thus it is handled first. Since this cluster does not meet the null threshold (as given in Figure 2 (c)) the cluster is partitioned (*Case 2*) by removing the property

Algorithm 2 Partition Clusters

1: **Function** Partition(PropClust C,PropUsage PU,$Thresh_{null}$)
2: $Tables \leftarrow \phi$
3: **for all** $clust_1 \in C$ **do**
4: $C \leftarrow (C - clust_1)$
5: **if** $Null\%(clust_1, PU) > NullThresh$ **then**
6: **repeat**
7: $p \leftarrow$ property causing most null storage
8: $clust_1 \leftarrow (clust_1 - p)$
9: **if** p exists in lower-support cluster **do continue**
10: **else** $Tables \leftarrow Tables \cup p$ /* Binary table */
11: **until** $Null\%(clust_1, PU) \leq NullThresh$
12: **end if**
13: $Tables \leftarrow Tables \cup clust_1$
14: **forall** $clust_2 \in C$ **do** $clust_2 \leftarrow clust_2 - (clust_2 \cap clust_1)$
15: Merge cluster fragments
16: **end for**
17: **return** $Tables$

that causes the most null storage, $P2$, corresponding to the property with minimum usage in the *property usage* list in Figure 2 (a). Since $P2$ is found in the lower-support cluster $\{P1, P2, P5, P6\}$ (*Case 2a*), it has a chance of being kept in an n-ary table. Removing $P2$ from $\{P1, P2, P3, P4\}$ creates the cluster $\{P1, P3, P4\}$ that falls below the null threshold of 20% (as given in Figure 2 (d)), thus it is considered a final table. Since $\{P1, P3, P4\}$ and $\{P1, P2, P5, P6\}$ contain overlapping properties, $P1$ is then pruned from $\{P1, P2, P5, P6\}$, creating cluster $\{P2, P5, P6\}$. Since cluster $\{P2, P5, P6\}$ also falls below the null threshold (as given in Figure 2 (d)), it would be added to the final table list in the next iteration. Finally, Figure 2 (e) gives the final schema with the union of tables returned from both phases.

Algorithm. Algorithm 2 gives the psuedocode for the *partitioning* phase, taking as arguments the list of property clusters (C) from Phase I, sorted in decreasing order by support value, the *property usage* list (PU), and the null threshold value ($Thresh_{null}$). The algorithm first initializes the final table list $Tables$ to empty (Line 2 in Algorithm 2). Next, it traverses each property cluster $clust_1$ in list C, starting at the cluster with highest support (Line 2 in Algorithm 2). Next, $clust_1$ is removed from the cluster list C (Line 2 in Algorithm 2). The algorithm then checks that $clust_1$ meets the null storage threshold (Line 2 in Algorithm 2). If this is the case, it considers $clust_1$ a final table (i.e., *Case 1*), and all lower-support clusters with properties overlapping $clust_1$ are pruned and cluster fragments are merged. (Lines 2 to 2 in Algorithm 2). If $clust_1$ does not meet the null threshold, it must be partitioned (i.e., *Case 2*). The algorithm finds property p causing maximum storage in $clust_1$ (corresponding to the minimum usage count for $clust_1$ in PU) and removes it. (Lines 2 and 2 in Algorithm 2). If p exists in a lower-support cluster (i.e., *Case 2a*), iteration continues, otherwise (i.e., *Case 2b*) p is added to $Tables$ as a binary table (Lines 2 and 2 in Algorithm 2). Partitioning continues until $clust_1$ meets the null storage threshold (Line 2 in Algorithm 2). When partitioning finishes, the algorithm considers $clust_1$ a final table, and prunes all lower-support clusters of properties overlapping with $clust_1$ while merging any cluster fragments (Lines 2 to 2 in Algorithm 2).

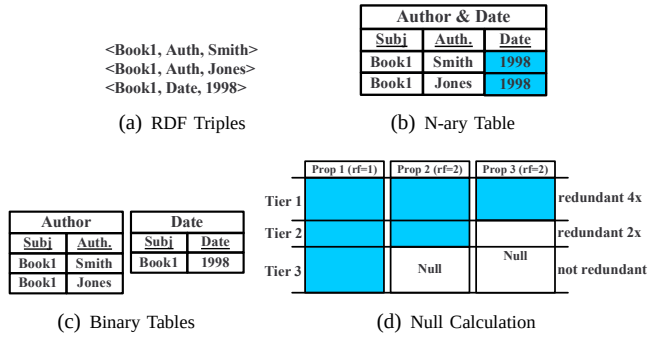


Fig. 3. Multi-Valued Attribute Example

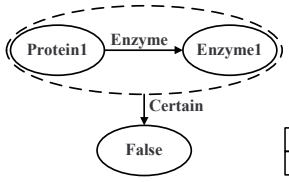
V. Important RDF Cases

We now outline two cases for RDF data important to schema creation. The first case deals with storage of *multi-valued* properties. The second case covers *refification*.

A. Multi-Valued Properties

Thus far, only *single-valued* properties have been considered for storage in n-ary tables. We now propose a method to handle *multi-valued* properties in our framework. For example, Figure 3(b) gives an example of a multi-valued property *data* for the RDf triples in Figure 3(a). Each property is assigned a *redundancy factor* (rf), a measure of repetition *per subject* in the RDF data set. If N_b is the total number of subject-property baskets, the *redundancy factor* for a property p is computed as $rf = \frac{PU.count(p)}{support(p) \times N_b}$. The term $PU.count(p)$ is a count of the *actual* property usage in a data set, while the term $support(p) \times N_b$ is the usage count of a property if it were *single-valued*. We note that the property usage table (PU) stores the usage count (including redundancy) of each property in the data set (e.g., in Figure 3(a), $PU.count(auth) = 2$ and $PU.count(date) = 1$), while the subject-property basket stores a property defined for a subject only *once* (e.g., in Figure 3(a) the basket is $book1 \rightarrow \{auth, date\}$). For the data in Figure 3(a), the rf value for *auth* is 2 ($\frac{2}{1\times1}$), while for *date* it is 1 ($\frac{1}{1\times1}$). To control redundancy, a user can define a *redundancy threshold*, that defines the maximum rf value a property can have in order to qualify for storage in an n-ary table. The rf values multiply each other, thus if two multi-valued properties are stored in an n-ary table, the amount of redundancy is $rf_1 \times rf_2$. Properties *not* meeting the threshold are explicitly disqualified from the *clustering* and *partitioning* phases, and stored in a binary table. The example given in Figure 3(c) stores the *auth* property in a separate binary table, removing redundant storage of the *date* property. If the *redundancy threshold* is 1, multi-valued properties are not stored in n-ary tables.

If multi-valued properties are allowed, null calculation (Section IV-B) changes. Due to space constraints, we outline how the calculation changes using the example in Figure 3(d), where $Prop$ 1 is single-valued (with $rf = 1$), while $Prop$ 2 and $Prop$ 3 are multi-valued (with $rf = 2$). The shaded columns of the table represent the property usage for each property if they were *single valued* (as calculated in the rf equation). Using these usage values, the initial null storage



(a) Reification Graph

Reification Triples

<reifID1, Subj, Protein1>
<reifID1, Prop, Enzyme>
<reifID1, Obj, Enzyme1>
<reifID1, Certain, False>

Reification Table

Subj.	*Subj.*	*Prop.*	*Obj.*	Certain
reifID1	Protein1	Enzyme	Enzyme1	False

(b) Reification Table

Fig. 4. Reification Example

value for a table can be calculated as discussed in Section IV-B. However, the final calculation must account for redundancy. In Figure 3(d), the table displays three *redundancy tiers*. *Tier* 1 represents rows with all three properties defined, thus having a redundancy of 4 (the rf multiplication for $Prop$ 2 and $Prop$ 3). *Tier* 2 has a redundancy of 2 (the rf for $Prop$ 2). Thus, the repeated null values for the $Prop$ 3 column must be calculated. *Tier* 3 does not have redundancy (due to the rf value of 1 for $Prop$ 1).

B. Reification

Reification is an RDF property that allows *statements* to be made about other RDF *statements*. An example of reification is given in Figure 4, taken from the Uniprot protein annotation data set [6]. The graph form of reification is given in Figure 4(a), while the RDF triple format is given at the top of Figure 4(b). The Uniprot RDF data stores for each $< protein, Enzyme, enzyme >$ triple information about whether the relationship between protein and enzyme has been verified to exist. This information is modeled by the *Certain* property, attached as a vertex to the whole $< protein, Enzyme, enzyme >$ triple for the graph representation in Figure 4(a). The only viable method to represent such information in RDF is to first create a new subject ID for the reification statement (e.g., *reifID1* in Figure 4(b)). Next, the *subject*, *property*, and *object* of the reified statement are redefined. Finally, the property and object are defined for the reification statement (e.g., *certain* and *false*, respectively, in Figure 4(b)). We mention *reification* as our data-centric method greatly helps query processing over this structure. Notice that for reification a set of at *least* four properties must *always* exist together in the data. Thus, our schema creation method will *cluster* these properties together in an n-ary table, as given in Figure 4(b). Our framework also makes an exception to allow reification properties *subject*, *property*, and *object* to exist in multiple n-ary tables for each reification edge. This exception means that a separate n-ary table will be created for each reification edge in the RDF data (e.g., *Certain* in figure Figure 4), Section VI will experimentally test this claim over the real-world Uniprot [6] data set.

VI. Experiments

This section experimentally evaluates our RDF schema creation approach with existing RDF storage methods.

A. Experimental Setup

We use three real-world data sets in our experiments: DBLP [16], DBPedia [17], and Uniprot [6]. All data sets

Statistic	DBLP	DBPedia	Uniprot
# Total Properties	30	19K	86
% total props stored in binary tables	40%	99.59%	69%
% total props stored in n-ary tables	60%	0.41%	31%
# Multi-Val Properties	4	6080	35
Min *rf* value for multi-val properties	3.4	4	1.2
% multi-val prop stored in n-ary tables	0%	0%	17%

(a) Schema Breakdown

Data Set	Binary	3-ary	4-ary	5-ary	(6+)-ary	Total
DBLP	12	2	6	4	6	30
DBPedia	18922	8	6	8	56	19K
Uniprot	60	4	9	8	5	86

(b) Table Distribution (by Property)

Fig. 5. Data Centric Schema Tables

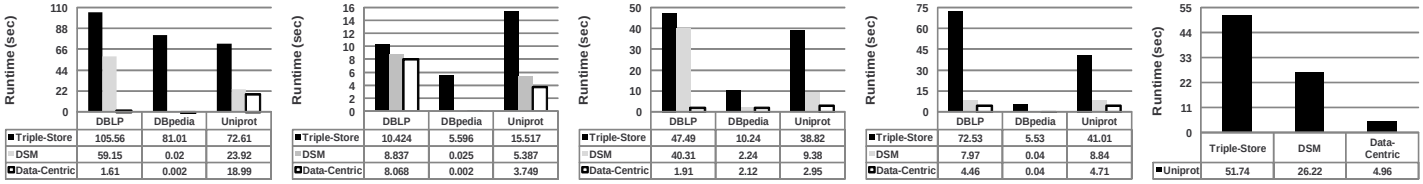
contain more than 10M triples; specific details for these data can be found in our companion technical report [27]. To create data-centric tables, the *support* parameter was set to 1% (a generally accepted default value [28]), the null threshold value was set to 30%, and the redundancy threshold was set to 1.5. Figure 5(a) gives the breakdown of the percentage of all properties for each data set that are stored in n-ary tables or binary tables (rows 1-3). This table gives the number of multi-valued properties in each data set (row 4), along with the *minimum* redundancy factor from all these properties (row 5). Only the Uniprot data set had multi-valued properties that met the redundancy threshold of 1.5, thus six of these properties (17%) were kept in n-ary tables (row 6). Figure 5(b) gives the table type (binary or n-ary) and the distribution of properties stored in each table type.

The experimental machine used is a 64-bit, 3.0 GHz Pentium IV, running Ubuntu Linux with 4Gbytes of memory. Our schema creation module was built using C++, and integrated with the PostgreSQL 8.0.3 database. We implemented a triple-store similar to many RDF storage applications (e.g., see [7], [9], [11], [12]), which is a single table containing three columns corresponding to an RDF subject, property, and object. We implemented the decomposed storage method by allowing each table to correspond to a unique property in the RDF dataset. Our data-centric approach built both n-ary and binary tables according to the structure of each data set. In-depth implementation details (e.g., indexing) are found in our companion technical report [27].

B. Experimental Evaluation

This section provides performance numbers for a set of queries based on previous benchmarks for Uniprot [10] and DBPedia [29]. Since the benchmarks were originally designed for their respective data, we first generalize the query in terms of its signature, then give the specific query for each data set. For each query, Figure 6 gives the query runtimes (in seconds) for each of the three storage approaches: triple-store, decomposed storage model (DSM), and our proposed data-centric approach. All times given are the average of ten runs, with the cache cleared between runs.

1) Query 1: Predetermined props/all subjects: Query 1 (Figure 6(a)) asks about a predetermined set of RDF properties. The general signature of this query is to select *all* records



(a) Query 1 (b) Query 2 (c) Query 3 (d) Query 4 (e) Query 5

Fig. 6. Queries

for which certain properties are defined. For specific data set queries (along with SQL), the reader is encouraged to see our companion technical report [27]. Overall, the data-centric approach shows better relative runtime performance for Query 1. Interestingly, the data-centric approach showed a factor of 65 speedup over the triple-store for the DBLP query, and a factor of 36 speedup over the decomposed approach. The DBLP data is relatively well-structured, thus, our data-centric approach stores a large number of properties in n-ary tables. In fact, the data-centric approach involved a single table access and no joins, while the triple-store and decomposed approach both used six table accesses and five self-joins. Similarly, the data-centric approach shows modest speedup for the DBPedia and Uniprot data sets again due to the decreased table accesses and joins. For instance the data-centric approach required five table accesses in the Uniprot query and four subject-to-subject joins, compared to six table accesses and five joins for the triple-store and decomposed approaches.

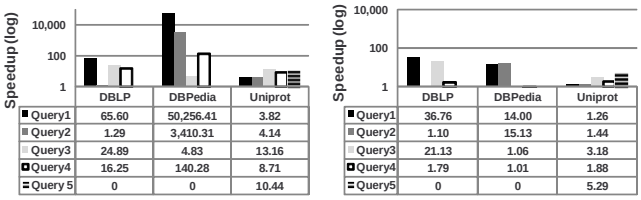
2) Query 2: Single subject/all defined properties: Query 2 (Figure 6(b)) involves a selection of all defined properties for a single RDF subject (i.e., a single record). For specific data set queries (along with SQL), the reader is encouraged to see our companion technical report [27]. For DBLP, this query accesses 13 RDF properties. The decomposed and triple-store approach involved 13 table accesses, while the data-centric approach involved nine. The performance between the decomposed and our data-centric approaches is similar in this case, due to the fact that some tables in the data-centric approach contained *extraneous* properties, meaning some stored properties were not used in the query. For DBPedia, the query accesses 23 RDF properties. The data-centric and decomposed approaches exhibit a similar relative performance to the triple-store with sub-second runtimes. However, the data-centric approach accessed a total of 17 tables, compared to the 23 needed by the decomposed and triple-store approaches. For Uniprot, this query accesses 15 RDF properties. The decomposed and triple-store approach involved fifteen table accesses along with fourteen subject-to-subject joins. Meanwhile, the data-centric approach involved 11 table accesses generating 10 subject-to-subject joins.

3) Query 3: Administrative query: Query 3 (Figure 6(c)) is an administrative query asking about date ranges for a set of recently modified RDF subjects in the data set. The general signature of this query is a range selection over dates. For specific data set queries (along with SQL), the reader is encouraged to see our companion technical report [27]. The data-centric approach shows better relative performance to that of the other schema approaches. Again, for the well-structured DBLP data, data-centric approach stored all query properties in a single table, causing a factor of 24 speedup over the triple-store, and a factor of 21 speedup over the decomposed approach. This performance is due to the data-centric approach requiring a single table access, with all five queried properties clustered to a single table. Meanwhile, both the the triple-store and decomposed approaches required separate table accesses for the range query and joins. The data-centric approach also shows good speedup for the semi-structured Uniprot data.

4) Query 4: Predetermined props/spec subjects: Query 4 (Figure 6(d))retrieves a specific set of properties for a particular set of RDF subjects. The general signature of this query is a selection of a set of RDF subjects (using the IN operator). For specific data set queries (along with SQL), the reader is encouraged to see our companion technical report [27]. Again, the data-centric approach shows better overall performance to that of the other schema approaches. For the Uniprot and DBLP, the data-centric approach shows good speedup over the triple-store, and a 1.8 speedup over the decomposed approach. The data-centric approach required only two table accesses and one join for the Uniprot data, and a single table access for the DBLP data, compared to four and five table accesses, respectively, for the other storage methods. The performance was similar for the data-centric and decomposed methods over the DBPedia data, as both queries accessed all binary tables.

5) Query 5: Reification: Query 5 (Figure 6(e)) involves a query using reification. For this query, only the Uniprot data set is tested, as it is the only experimental data set that makes use of reification. The query here is to *display the top hit count for statements made about proteins*. In the Uniprot dataset, this information is stored as reified data with the the *object* property corresponding to a protein identifier, and the hit count modeled as the *hits* property. The large difference in performance numbers here is mainly due to the table accesses needed by both the decomposed and triple-store to reconstruct the statements used for reification. Our data-centric approach involved a single table access with no joins, due to the fact that the reification structure being clustered together in n-ary tables. Our approach shows a speedup of 5.29 and 10.44 over the decomposed and triple-store approaches.

6) Relative Speedup: Figure 7(a) gives the relative speedup for the data-centric approach over the triple-store approach for each query and data set, while Figure 7(b) gives the same speedup data over the decomposed approach. The DBLP data set is well-structured, and our *data centric* approach showed superior speedup for queries 1 and 3 over the DBLP data as it clustered *all* related data to the same table. Thus, the queries were answered with a single table access, compared to multiple accesses and joins for the triple-store and decomposed



(a) Speedup over Triple Store (b) Speedup over DSM

Fig. 7. Relative Speedup

approaches. For DBPedia queries 1 and 2, the data-centric approach showed speedup over the decomposed approach due to accessing the few n-ary tables present to store this data. However, this data was mostly semi-structured, thus queries 3 and 4 showed similar performance as they involved the *same* table structure. The speedup over the triple-store for DBPedia was superior as queries using the data-centric approach involved tables (1) with smaller cardinality and (2) containing, on average, only the properties necessary to answer the queries, as opposed to the high-selectivity joins used by the large triple-store. Our data-centric approach showed a moderate speedup performance for the Uniprot queries due to two main factors: (1) some data-centric tables contained extraneous properties and multi-valued attributes that caused redundancy, and (2) the semi-structured nature of the Uniprot data set led to a similar number of relative joins and table accesses.

VII. Conclusion

This paper proposed a data-centric schema creation approach for storing RDF data in relational databases. Our approach derives a basic *structure* from RDF data and achieves a good balance between using n-ary tables (i.e., *property tables*) and binary tables (i.e., *decomposed storage*) to tune RDF storage for efficient query processing. First, a *clustering* phase finds all related properties in the data set that are candidates to be stored together. Second, the clusters are sent to a *partitioning* phase to optimize for storage of extra data in the underlying database. We compared our data-centric approach with state-of-the art approaches for RDF storage, namely the *triple store* and *decomposed storage*, using queries over three real-world data sets. The data-centric approach shows large orders of magnitude performance improvement over the triple store, and speedup factors of up to 36 over the decomposed approach.

References

[1] "World Wide Web Consortium (W3C): http://www.w3c.org."
[2] "W3C Semantic Web Activity: http://www.w3.org/2001/sw/."
[3] D. Kotzinos, S. Pediaditaki, A. Apostolidis, N. Athanasis, and V. Christophides, "Online curriculum on the semantic Web: the CSD-UoC portal for peer-to-peer e-learning," in *WWW*, 2005.
[4] J. S. Jeon and G. J. Lee, "Development of a Semantic Web Based Mobile Local Search System," in *WWW*, 2007.
[5] X. Wu, L. Zhang, and Y. Yu, "Exploring social annotations for the semantic web," in *WWW*, 2006.
[6] "Uniprot RDF Data Set: http://dev.isb-sib.ch/projects/uniprot-rdf/."
[7] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle, "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases," in *SemWeb*, 2001.
[8] D. Beckett, "The Design and Implementation of the Redland RDF Application Framework," in *WWW*, 2001.
[9] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *ISWC*, 2002.
[10] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An Efficient SQL-based RDF Querying Scheme," in *VLDB*, 2005.
[11] S. Harris and N. Gibbins, "3store: Efficient bulk rdf storage," in *PSSS*, 2003.
[12] L. Ma, Z. Su, Y. Pan, L. Zhang, and T. Liu, "Rstar: an rdf storage and query system for enterprise resource management," in *CIKM*, 2004.
[13] K. Wilkinson, "Jena Property Table Implementation," in *SSWS*, 2006.
[14] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton, "Extending rdbmss to support sparse datasets using an interpreted attribute storage format," in *ICDE*, 2006.
[15] D. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *VLDB*, 2007.
[16] B. Aleman-Meza, F. Hakimpour, I. B. Arpinar, and A. P. Sheth, "Swetodblp ontology of computer science publications," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 3, pp. 151–155, 2007.
[17] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "DBpedia: A Nucleus for a Web of Open Data," in *ISWC*, 2007.
[18] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model," in *SIGMOD*, 1985.
[19] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura, "A Path-Based Relational RDF Database," in *ADC*, 2005.
[20] R. Angles and C. Gutierrez, "Querying rdf data from a graph database perspective," in *ESWC*, 2005.
[21] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in sql databases," in *VLDB*, 2000.
[22] S. Agrawal, V. R. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD*, 2004.
[23] S. B. Navathe and M. Ra, "Vertical partitioning for database design: A graphical algorithm," in *SIGMOD*, 1989.
[24] S. Papadomanolakis and A. Ailamaki, "Autopart: Automating schema design for large scientific databases using data partitioning," in *SSDBM*, 2004.
[25] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *VLDB*, 1994.
[26] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," in *ICDE*, 2001.
[27] J. J. Levandoski and M. F. Mokbel, "RDF Data-Centric Storage," University of Minnesota, Tech. Rep. UM-CS-TR-9999, 2009.
[28] R. Agrawal and J. Kiernan, "An Access Structure for Generalized Transitive Closure Queries," in *ICDE*, 1993.
[29] "RDF Store Benchmarks with DBpedia: http://www4.wiwiss.fu-berlin.de/benchmarks-200801/."