Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 09-003

Data-Centric Schema Creation for RDF

Justin J. Levandoski and Mohamed F. Mokbel

January 26, 2009

Data-Centric Schema Creation for RDF

Justin J. Levandoski [1], Mohamed F. Mokbel [2]

Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA
[1]justin@cs.umn.edu, [2]mokbel@cs.umn.edu

***Abstract*— Very recently, the vision of the Semantic Web has brought about new challenges in data management. One fundamental research issue in this arena is storage of the Resource Description Framework (RDF): the data model at the core of the Semantic Web. In this paper, we study 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. Furthermore, our approach does not assume query workload statistics. 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 previous state-of-the art approaches.**

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]. Currently, many heterogeneous data sources exist in different applications and domains across the world, causing interoperability problems when these data sources need to be shared across boundaries. The Semantic Web establishes a means to solve this problem by giving data semantic *meaning*, allowing machines to consume, understand, and reason about the structure and purpose of the data. Furthermore, the Semantic Web is not *distinct* from the World Wide Web (WWW). Rather, it is designed to be complementary to the WWW, tying together data from a range of heterogeneous sources. 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.

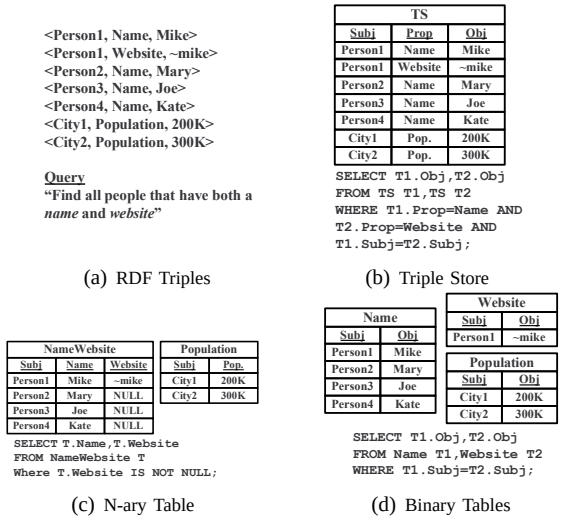
The core of the Semantic Web is built on the Resource Description Framework (RDF) data model. RDF provides a simple syntax, where each data item is broken down into a $<subject, property, object>$ triple. The *subject* represents an entity instance, identified by a Uniform Resource Identifier (URI). 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 ICDE 2009:

```
person1 hasName ''John''
confICDE09 hasTitle ''ICDE 2009''
person1 isReviewerFor confICDE09
```

While the ubiquity of the RDF data model has yet to be realized, many application areas and use-cases exist for RDF [3], such as intelligence [4], mobile search environments [5], social networking [6], and biology and life science [7], making it an emerging and challenging research domain.

An important and fundamental challenge exists in storing and querying RDF data in a scalable and efficient manner, making RDF data management a problem aptly suited for the database community. In fact, many RDF storage solutions use relational databases to achieve this scalability and efficiency, implementing a variety of storage schemas. 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 [8], [9], [10], [11], [12], [13], [14], where each RDF triple is stored directly in a three-column table (Figure 1(b)). This approach suffers during query execution due to a proliferation of self-joins, as shown in the SQL query in Figure 1(b). Another schema approach is the *property table* [9], [15], [12], [16], [17] (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 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 [18]. The *decomposed storage* schema [19] (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 query.

In this paper, we propose a new storage solution for RDF data that aims to avoid the drawbacks of these previous approaches, i.e., self joins on a triple table, a high ratio of null storage in property tables, and the proliferation of joins over binary tables. Our approach can be considered *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. The main intuition driving this *data-centric* approach is that RDF data sets across different domains require *unique* storage schemas. Furthermore, our approach does not assume a query workload for schema creation, making it useful for situations where a query workload cannot be reliably derived, likely in cases where a majority of queries on an RDF knowledge base are ad-hoc. In order to build a relational schema without a query workload and achieve efficient query



(a) RDF Triples (b) Triple Store

(c) N-ary Table (d) Binary Tables

Fig. 1. RDF Storage Example

processing, our data-centric approach defines the following trade off: (1) storing as much RDF data together, reducing, on average, the need for joins in a query plan, and (2) tuning extra storage (i.e., null storage) to fall below a given threshold.

Our data-centric schema creation approach involves two phases, namely *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 to be stored together in an n-ary table. Likewise, properties *not* in a cluster are candidates to be stored 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; specifically, the DBLP [20], DBPedia [21], and Uniprot [7] data sets. 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 shows orders of magnitude performance improvement over the triple store, and speedup factors of up to 36 over a straight binary table approach.

The rest of this paper is organized as follows. Section II highlights related work. Section III gives an overview of how our schema creation approach interacts with a DBMS. Section IV gives the details of our data-centric schema creation approach. Handling *multi-valued attributes* and *reification* in RDF is covered in Section V. Section VI gives experimental evidence that our approach outperforms previous approaches. Finally, Section VII concludes this paper.

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 [9], [12], Sesame [11], 3-Store [13], R-Star [14], RDFSuite [8], and Redland [10]. (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 [15], [16], [17]. Oracle [9], [12] also makes use of property tables as *secondary* structures, called materialized join views (MJVs). (3) The *decomposed storage model* [22] (Figure 1(d)) has recently been proposed as an RDF storage method [19], 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 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*). Furthermore, we note that 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 [12]. Our approach provides an *automated* method to place properties together in tables based on the structure of the data.

Other work in RDF storage has dealt with storing pre-computed *paths* in a relational database [23], 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, has been proposed [24]. 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 [25], horizontal and vertical partitioning [26], [27], and partitioning for large scientific workloads [28]. Our automated data-centric schema design method for RDF differs from these approaches in two main ways. First, our method does *not* assume a set of query workload statistics, rather, we base our method on the structure found in RDF data. Second, these previous schema creation techniques do not 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. System Overview and Problem Definition

System Overview. Figure 2 gives an overview of how RDF data is managed using a relational database system. In general, two modules (represented by dashed rectangles) 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 (represented by a shaded rectangle in Figure 2). The schema creation process takes as input an RDF data set. The output of our technique is a schema (i.e., a set

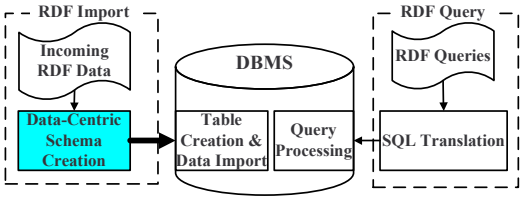


Fig. 2. RDF Query Architecture using DBMS

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 and (2) minimize the amount of extra (e.g., null) data 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 overhead that also 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

In this section, we present our data-centric schema creation algorithm for RDF data. The output of this algorithm can be considered a balanced mix of binary and n-ary tables based on the structure of the data. Unlike previous techniques that use the same schema regardless of the structure of the data, 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 subjects) may result in a few large n-ary tables used as a primary storage schema. On the other hand, a relatively *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.

The basic idea behind our approach is to implement a two-phase algorithm that: (1) finds interesting clusters of RDF properties that are candidates to be stored in the *same* n-ary table. This process relates to the first criterion in our problem definition (Section III), and (2) partition the clusters to balance the tradeoff between storing the maximum number properties *together*, while ensuring that extra (i.e., null) storage is kept to a minimum. This process relates to the second criterion in our problem definition. The output of our algorithm is a schema that achieves a balance between a set of n-ary and binary tables based on the structure of the RDF data. The n-ary tables contain a subject column with multiple RDF property columns (i.e., a property table), while the binary tables contain a subject column with a single property column (i.e., decomposed storage tables).

The rest of this section introduces our data-centric schema creation algorithm. First, an overview of our algorithm is given, followed by a presentation of its details.

A. Algorithm Overview and Data Structures

Algorithm parameters. Our schema creation algorithm takes as parameters an RDF data set, along with two numerical values, namely *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. This parameter exists to tune the null storage to an appropriate level for efficient query processing.

Data Structures. 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, since it is only defined for the subject *Person1*. Likewise, the *Name* property would have a usage count of four (defined for subjects $Person1$-$Person4$), 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: $Person1 \rightarrow \{Name, Website\}$, $Person2 \rightarrow \{Name\}$, $Person3 \rightarrow \{Name\}$, $Person4 \rightarrow \{Name\}$, $City1 \rightarrow \{Population\}$, and $City2 \rightarrow \{Population\}$.

High-level algorithm. Our schema creation algorithm involves two main phases, namely *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, specifically, maximum frequent itemset generation, to look for 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 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.

Algorithm 1 gives the pseudocode for our schema creation process. The function takes as arguments *RDFTriples*, an RDF

Algorithm 1 RDF Data-Centric Schema Creation

1: **Function** BuildRDFSchema(RDFTriples T,$Thresh_{sup}$,$Thresh_{null}$)
2: /* Preprocessing - Build Data Structures*/
3: $Baskets, PropertyUsage \leftarrow$ BuildDS(*T*)
4: /* Phase I: Clustering */
5: $TablesI$,$Clusters \leftarrow$ Cluster($Baskets$, $PropertyUsage$, $Thresh_{sup}$, $Thresh_{null}$)
6: /* Phase II: Partitioning */
7: $TablesII \leftarrow$ Partition($Clusters$,$PropertyUsage$,$Thresh_{null}$)
8: **return** $TablesI \cup TablesII$

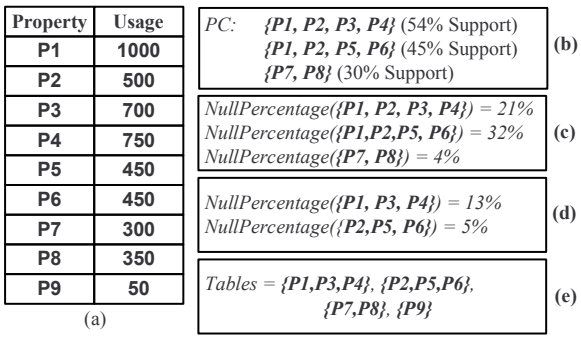


Fig. 3. RDF Data Partitioning Example

data set, and two threshold values $Thresh_{sup}$, the support threshold for clustering, and $Thresh_{null}$, the null ratio threshold for partitioning. The data structures, namely, the *property usage list* and *subject-property baskets*, are created using the *BuildDS* function (Line 3 in Algorithm 1). The first phase of the algorithm is invoked to find property clusters by calling the function *Cluster*, passing the *subject-property* baskets, *property usage* list, *support threshold*, and *null threshold* as arguments (Line 5 in Algorithm 1). Generated clusters that need to be sent to the *partitioning* phase are stored in the list *Clusters*, while the initial list of final tables is stored in list $TablesI$. Next, the second phase (*partitioning*) is started by calling the method $Partition$ (Line 7 in Algorithm 1), passing as parameters the property clusters ($Clusters$), the *property usage* list, and the *null storage threshold*. The function $Partition$ returns the second part of the final table set, $TablesII$. The union of tables lists $TablesI$ and $TablesII$ is considered the complete final schema, and is returned by the high-level algorithm (Line 8 in Algorithm 1).

Example. Figure 3 gives example data that will be used as a running example throughout the rest of this section to demonstrate how our partitioning method works. Figure 3 (a) gives an example *property usage* list with nine properties. The data given in Figure 1 will also be used in our examples. Phase I is the topic of Section IV-B, while Phase II is discussed in Section IV-C.

B. Phase I: Clustering

Objective. The objective of the clustering phase is to find property clusters, or groups of *related* properties using the *subject-property basket* data structure. Properties in each cluster are candidates to be stored together 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 is also responsible for building an initial set of final tables. These tables consist of: (1) the properties that are *not* found in the generated clusters (thus being stored in binary tables), and (2) property clusters that *meet* the *null threshold* and do not contain properties that *overlap* with other clusters, thus not needed in the *partitioning* phase.

Main idea. The clustering phase involves two main 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 [29]. For our purposes, the terms *frequent itemsets* and *clusters* are used synonymously. The idea behind the *clustering phase* is to find 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 that *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. In general, we can think of the *support threshold* as the strength of the relation between properties in the data. 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. Intuitively, we are interested in these clusters because our schema creation method aims to *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. This list of tables contains (1) the 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 no 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%, and the six subject-property baskets generated from the data in Figure 1(a): $Person1 \rightarrow \{Name, Website\}$, $Person2 \rightarrow \{Name\}$, $Person3 \rightarrow \{Name\}$, $Person4 \rightarrow \{Name\}$, $City1 \rightarrow \{Population\}$, and $City2 \rightarrow \{Population\}$. 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%. The

Algorithm 2 Clustering

```
1: Function Cluster(Baskets B,Usage PU,Thresh_sup,Thresh_null)
2: Clusters ← GetClusters(B,Thresh_sup)
3: /* Initialize final table set */
4: Tables ← properties not in PC /* Binary tables */
5: for all clust_1 ∈ PC do
6:     OK ← false
7:     /* Test 1: cluster is below null threshold */
8:     if Null%(clust_1, PU) ≤ Thresh_null then
9:         OK ← true
10:        /* Test 2: cluster doesn't contain overlapping properties */
11:        for all clust_2 ∈ PC if clust_1 ∩ clust_2 ≠ φ then OK ← false
12:    end if
13:    if OK then Tables ← Tables ∪ clust_1; Clusters ← Clusters − clust_1
14: end for
15: return Tables,Clusters
```

cluster $\{Population\}$ occurs in 2 of 6 baskets (with support 33%). The clusters $\{Website\}$ and $\{Name, Website\}$ occur in 1 of the 6 property baskets, giving them a support of 16%. In this case, the $\{Name, Website\}$ is generated as a cluster, since 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. Note also that this arrangement corresponds to the tables in Figure 1(c), and that the $\{Name, Website\}$ table contains 25% null values. With the null threshold of 20%, the initial final table list would contain $\{Population\}$, while the cluster list would be set to $\{Name, Website\}$, as it does not meet the null threshold.

As a second example, Figure 3(b) gives three example clusters along with their support values, while Figure 3 (c) gives their null storage values (null storage calculation will be covered in the algorithm discussion). 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 2 gives the pseudocode for the clustering phase. The algorithm takes as parameters the subject-property baskets (B), the property usage list (PU), the support threshold ($Thresh_{sup}$), and the null threshold parameter ($Thresh_{null}$). The algorithm begins by generating clusters and storing them in a list $Clusters$, sorted in descending order by support value. (Line 2 in Algorithm 2). This is a direct call to a maximum frequent itemset algorithm [29], [30]. Next, we initialize $Tables$, an initial list of final tables, to the properties that do *not* qualify for clusters (i.e., stored in binary tables). Next, the algorithm filters out the set of clusters that qualify for final tables (Lines 5 to 14 in Algorithm 2). The algorithm first checks that each cluster's null storage falls below the given threshold (Line 9 in Algorithm 2). In general, the null storage for a cluster c can be calculated from a property usage list PU as follows. 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,

Subj	P1	P2	P3	P4
Usage: 1000	Usage: 1000	Usage: 500	Usage: 700	Usage: 750
		Null: 500	Null: 300	Null: 250

Null({P1,P2,P3,P4}) = 1050/5000 = 21%

Fig. 4. Null Calculation

in Figure 3 (b) if $c = \{P1, P2, P3, P4\}$), $|c| = 4$ and $PU.maxcount(c) = 1000$ (corresponding to $P1$). Figure 4 gives a graphical representation for a sample null calculation using cluster $\{P1, P2, P3, P4\}$. If $PU.count(c_i)$ is the usage count for the ith property in c, the *null storage percentage* for c is:

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

The algorithm also checks that a cluster does not contain properties that overlap with other clusters (Line 11 in Algorithm 2). For clusters that pass this test, they are *removed* from the cluster list $Clusters$ and added to the final table list $Tables$ (i.e., as n-ary tables) (Line 13 in Algorithm 2). Finally, the algorithm returns the initial final table list and clusters, assumed to be *sorted* in decreasing order by their support value (Line 15 in Figure 2).

C. Phase II: Partitioning

Objective. 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 academic publications: $TitleConf = \{subj, title, conference\}$ and $TitleJourn = \{subj, title, journal\}$. In this case, an RDF query asking for all published titles would involve two table accesses and a union, due to the fact that publications can exist in a conference or a journal, but not both. (2) Ensuring that each partitioned cluster, when populated with data as an n-ary table, falls below the null storage threshold. This objective is based on a main requirement of our algorithm, stated in the problem definition given in Section III, and tunes our schema for efficient query processing.

Main idea. To achieve our objectives, we propose a greedy algorithm that continually attempts to keep the cluster with *highest* support intact, while *pruning* lower-support clusters containing overlapping properties (i.e., ensuring that each property exists in a *single* table). The reason for this greedy approach is that, intuitively, the clusters with *highest* support contain properties that occur together *most often* in the data set. Recall that support is the percentage of RDF subjects that have *all* of the cluster's properties. Thus, keeping high support clusters intact implies that the most RDF subjects (with the cluster's properties defined) will be stored in this table. Our greedy approach iterates through the given cluster list (sorted in decreasing order by support value), takes the *highest* support

Algorithm 3 Partition Clusters

```
1:  Function Partition(PropClust C,PropUsage PU,Thresh_null)
2:  Tables ← φ
3:  /* Traverse list from highest support to lowest */
4:  for all clust_1 ∈ C do
5:      C ← (C − clust_1)
6:      if Null%(clust_1, PU) > NullThresh then
7:          /* Case 2: cluster needs partitioning */
8:          repeat
9:              p ← property causing most null storage
10:             clust_1 ← (clust_1 − p)
11:             /* Case 2a: partitioned property in other cluster */
12:             if p exists in lower-support cluster do continue
13:             /* Case 2b: partitioned property not in other cluster */
14:             else Tables ← Tables ∪ p /* Binary table */
15:         until Null%(clust_1, PU) ≤ NullThresh
16:     end if
17:     Tables ← Tables ∪ clust_1
18:     forall clust_2 ∈ C do clust_2 ← clust_2 − (clust_2 ∩ clust_1)
19:     Merge cluster fragments
20: end for
21: return Tables
```

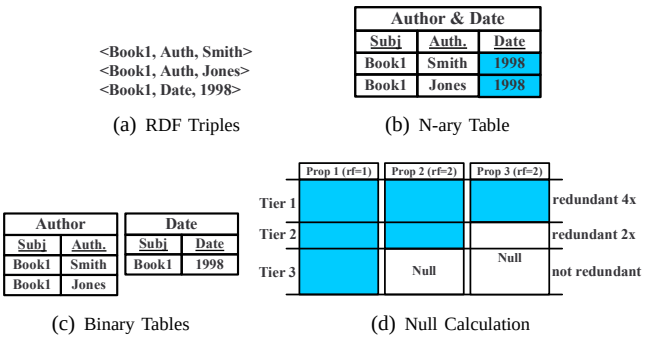
cluster, and handles two main cases based on its null storage computation (null computation is discussed in Section IV-B). *Case 1*: the cluster meets the null storage threshold. This case handles the given cluster from Phase I that meets the null threshold but contains overlapping properties. In this 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 high-support 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. The reason for removing p is to remove the maximum null storage from the cluster possible in one iteration. Also, we note that support for clusters is *monotonic*. That is, given two clusters c_1 and c_2, $c_1 \subseteq c_2 \Leftarrow support(c_1) \geq support(c_2)$. With this property, the cluster will still meet the given support threshold. After removing p, the *partitioning* process handles two cases. *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 3, 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 3 (b)), thus it is handled first. Since this cluster does not meet the null threshold (as given in Figure 3 (c)) the cluster is partitioned (*Case 2*) by removing the property that causes the most null storage, $P2$, corresponding to the property with minimum usage in the *property usage* list in Figure 3 (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 3 (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 3 (d)), it would be added to the final table list in the next iteration. With the two final tables created in this example, and the initial final table list created by the *clustering* phase, Figure 3 (e) gives the combined final table list.

Algorithm. Algorithm 3 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 3). Next, it traverses each property cluster $clust_1$ in list C, starting at the cluster with highest support (Line 4 in Algorithm 3). Next, $clust_1$ is removed from the cluster list C (Line 5 in Algorithm 3). The algorithm then checks that $clust_1$ meets the null storage threshold (Line 6 in Algorithm 3). 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 18 to 19 in Algorithm 3). If $clust_1$ does not meet the null theshold, 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 9 and 10 in Algorithm 3). 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 12 and 14 in Algorithm 3). Partitioning continues until $clust_1$ meets the null storage threshold (Line 8 in Algorithm 3). 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 18 to 19 in Algorithm 3).

V. Important RDF Cases

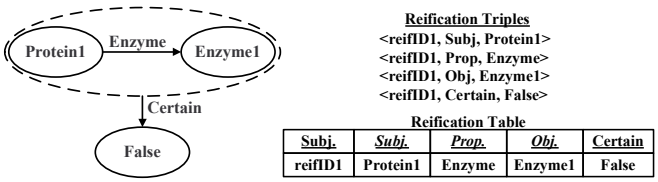
In this section, we highlight two cases for RDF that are important to our schema creation technique. The first case deals with multi-valued properties (i.e., properties defined *multiple* times for the same subject). The second case covers with *reification*; an RDF data model structure that allows statements to be made about other whole RDF statements.



(a) RDF Triples (b) N-ary Table

(c) Binary Tables (d) Null Calculation

Fig. 5. Multi-Valued Attribute Example



(a) Reification Graph (b) Reification Table

Fig. 6. Reification Example

A. Multi-Valued Properties

We assumed thus far that candidate properties for storage in n-ary tables are *single-valued* (i.e., defined *once* for each subject). However, *multi-valued* properties exist in RDF data that would cause redundancy if stored in n-ary tables. For example, given the data in Figure 5(a), an n-ary table (Figure 5(b)) stores the *date* property redundantly due to the multi-valued attribute *auth*. We now outline a method to deal with multi-valued properties in our schema creation framework.

If a certain amount of redundant data storage is tolerated, we propose the following method to handle it in our framework. 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 $rf = \frac{PU.count(p)}{support(p) \times N_b}$. Intuitively, 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 5(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 5(a) the basket is $book1 \rightarrow \{auth, date\}$). For the data in Figure 5(a), the rf value for *auth* is 2 ($\frac{2}{1 \times 1}$), while for *date* it is 1 ($\frac{1}{1 \times 1}$). To control redundancy, a *redundancy threshold* can be defined that sets the maximum rf value a property can have in order to qualify for storage in an n-ary table. We note that rf values multiply each other, that is, 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. For example, the policy in Figure 5(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 allowed in n-ary tables, thus they are all stored in binary tables.

The null calculation (as discussed in Section IV-B) for clusters are affected if multi-valued properties are allowed. Due to space restrictions, we do not list new calculations for this case. However, we outline how the calculation changes using the example in Figure 5(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 value for a table can be calculated as discussed in Section IV-B. However, the final calculation must account for redundancy. In Figure 5(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). In general, the null calculation must be aware of the redundancy distribution in the tables containing multi-valued properties.

B. Reification

Reification is a special RDF data model property that allows *statements* to be made about other RDF *statements*. An example of reification is given in Figure 6, taken from the Uniprot protein annotation data set [7]. The graph form of reification is given in Figure 6(a), while the RDF triple format is given at the top of Figure 6(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 for the graph representation in Figure 6(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 6(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 6(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 6(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 6), Section VI will experimentally test this claim over the real-world Uniprot [7] data set.

Statistics	DBLP	DBPedia	Uniprot
Triples	13.5M	10M	11M
No. Properties	30	19K	86
Mult-Val. Properties	14%	32%	41%
Reified Triples	0	0	232K
% null for wide table	95%	99%	91%

TABLE I
DATA SET STATISTICS

VI. EXPERIMENTS

This section provides experimental evidence that our data-centric schema creation approach outperforms the *triple-store* and the *decomposed storage* approaches for query processing on a relational database. Three real-world data sets are used from three different domains. We test the DBLP [20], DB-Pedia [21], and Uniprot [7] RDF data sets with five queries based on previous benchmarks on this data [31], [12]. All experiments are performed using PostgreSQL.

The rest of this section is organized as follows. Section VI-A provides an overview of our real-world RDF data sets. Section VI-B gives the output of our data-centric schema creation algorithm for each RDF data set. Section VI-C describes the system setup for our experiments. Section VI-D studies performance for a set of benchmark queries that run over three real-world data sets.

A. RDF Data Sets

In this section, we give an overview of three publicly available real-world RDF datasets. Specifically, the datasets we use in our experiments are the DBLP [20], DBPedia [21], and Uniprot [7] protein annotation data sets. Table I gives a handful of overview statistics for each of the data sets.

DBLP. The Digital Bibliography and Library Project (DBLP) is a well-known database tracking bibliographical information for major computer science journals and conference proceedings. The DBLP server indexes more than 955K computer science articles. For our experiments, we use the SwetoDBLP [20] data set, an RDF version of the DBLP database with approximately 13M triples. In total, the DBLP dataset contains 30 properties, of which only 14% are multi-valued (i.e., appearing more than once for a given subject). Also, the DBLP data set does *not* make use of reification, and if stored in a single wide property table the data would cause 95% null storage.

DBPedia. The DBPedia [21] data set encodes Wikipedia data in RDF. For our experiments, we use a subset of the data that encodes infoboxes found on the English version of Wikipedia. In total, this data set contains 10M triples. DBPedia uses 19K unique RDF properties in encoding; a high value relative to the other data sets. In total, 32% of these properties are multi-valued. DBPedia does *not* make use of reification. If stored in a wide property table, the DBPedia data shows the highest percentage of null storage at 99%.

Uniprot. The Uniprot [7] dataset is a large-scale database of protein sequence and annotation data. This dataset joins three of the largest protein annotation databases in the world (Swiss-Prot, TrEMBL, and PIR) into one comprehensive data set open to the research community at large. Uniprot stores a wide array of data, ranging from cellular components, proteins, enzymes, and citations for journal publications about each protein. In total, the Uniprot data we use is 11M triples. A total of 86 properties exist in this dataset where 41% are multi-valued. The Uniprot dataset also contains roughly 23K reified statements. If this dataset were to be stored using a single wide property table, the total null storage would be 91%.

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. 7. Data Centric Schema Tables

B. Data-Centric Schema Tables

This section gives an overview of the tables created by our data-centric schema approach for the three data sets discussed in Section VI-A. For this purpose, the *support* parameter was set to 1% (a generally accepted default support value [32]), the null threshold value was set to 30%, and the redundancy threshold was set to 1.5. Figure 7(a) gives the breakdown of the percentage of all properties for each data set that are stored in either n-ary tables or binary tables (rows 1-3). Also, 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 (given in row 6).

For the tables created for each data set, Figure 7(b) gives the table type (i.e., binary or n-ary tables) and the distribution of properties stored in each table type for each data set. We note that the numbers given for each dataset sum to the total number of properties given in the first row in Figure 7(a). For example, the DBLP dataset contains 30 properties (Figure 7(a)), and the sum of all properties in Figure 7(b) for DBLP sum to 30. Also, the number of properties in binary tables given in Figure 7(b) correspond to the percentages given in the second row in Figure 7(a) (e.g., for DBLP .40*30 = 12), while the number of properties in n-ary tables given in Figure 7(b) correspond to the percentages in the third row Figure 7(a) (e.g., for DBLP .60*30 = 18).

For the 60% of the properties stored in property tables for the DBLP dataset, large tables sizes (i.e., with three and greater properties) are favored. This number implies that larger

clusters of properties were found to exist *often* in the data. For the Uniprot data a range of table sizes are favored, while the DBPedia data set favors both smaller and larger property tables.

C. Experimental Setup

This section gives the details of our experimental setup. The experimental machine used in our experiments is a 64-bit single-processor 3.0 GHz Pentium IV, running Feisty Ubuntu Linux with 4Gbytes of memory. The hard-disk is a standard SCSI setup with a 80GB volume.

1) Implementation: All experiments were evaluated using the open-source PostgreSQL 8.0.3 database. Our schema creation module was built using C++, and integrated with PostgreSQL database. Specifically, our module reads any RDF dataset (locally or remotely) in any standard transport format. After the schema creation process, SQL scripts are created for table creation and forwarded to PostgreSQL.

2) Storage Details: For all of the approaches, a dictionary-encoding scheme is used, meaning that each string in the RDF dataset is mapped to a unique 32-bit integer. Thus, each table stores 32-bit integers, while the integer-to-string dictionary is stored in a separate table. For the dictionary table, two B+ trees exist: one clustered on the integer (i.e., encoding) column, while an unclustered index is built over the string column. As the bulk of query processing is performed on integers instead of strings, the dictionary-encoding scheme was shown to provide an oder-of-magnitude performance improvement for *all* storage approaches in our experiments.

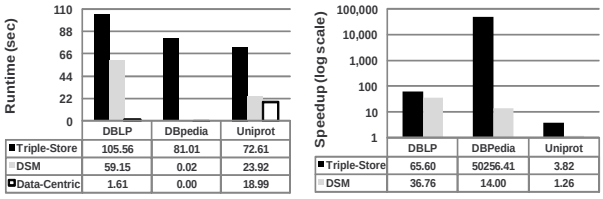
Triple-Store. We implement the triple-store similar to many RDF storage applications using triple-stores as their primary storage approach (e.g., see [8], [11], [13], [14]), which is a single table containing three columns corresponding to an RDF subject, property, and object. The table has three B+ tree indices built over it. The first index is clustered on (subject, property, object), second index is unclustered on (property, object, subject), and the third index is unclustered on (object, subject, property).

Decomposed Storage. We implement the decomposed RDF storage method as follows: each table corresponds to a unique property in the RDF dataset. A clustered B+ tree index is built over the subject column, while an unclustered B+ tree index is built over the object column.

Our Data-Centric Approach. Our data-centric approach results in both n-ary and binary tables. For n-ary tables, a clustered B+ tree index is built over the subject column, while an unclustered B+ tree index is built over all subsequent columns (representing properties). For binary tables, indices were built according to the decomposed model described above.

D. Experimental Evaluation

This section provides performance numbers for a set of benchmark queries on the data sets introduced in Section VI-A. The queries used in these experiments are based on previous benchmark queries for Uniprot [12] and DBPedia [31]. We



(a) Query 1 Runtime (b) Query 1 Speedup

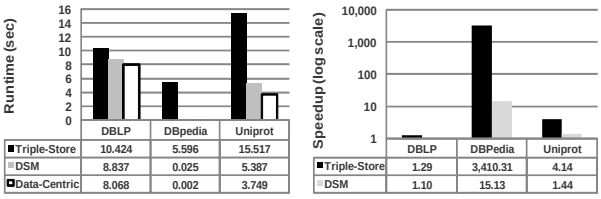
Fig. 8. Query 1

note that 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. In total, we use five queries with the following signatures: *predetermined properties retrieving all subjects*, *single subject retrieving all defined properties*, *administrative query*, *predetermined properties retrieving specific subjects*, and *reification retrieval*. For each query, we plot the query runtime for each of the three storage approaches: triple-store, decomposed storage model (DSM), and our proposed data-centric schema creation approach. All times given are the average of several runs, with the cache cleared between each run. In general, our data-centric schema creation approach shows superior performance over all queries. This performance improvement is mainly due to the reduction of joins in the query execution, due to common properties in the queries existing in the same table.

1) Query 1: Predetermined props/all subjects: Query 1 asks about a predetermined set of RDF properties. The general signature of this query is to select *all* records for which certain properties are defined. Figure 8(a) gives the runtime for this query across all three data sets, while Figure 8(b) gives the speedup of the data-centric approach over the triple-store and decomposed method.

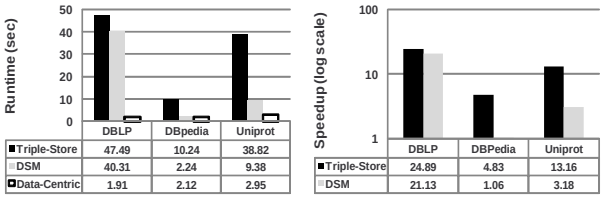
DBLP. For DBLP, this query accesses six RDF properties, and translates to *Display the author, conference, title, publisher, date, and year for all conference articles*. The data-centric approach stores all relevant properties in a single relation, thus producing a single table access (table distributions for our data-centric approach are discussed in Section VI-B). Meanwhile, both the decomposed and triple-store approaches involve six table accesses including subject-to-subject joins, with the triple-store involving five self-joins, respectively. Due to the relative number of table accesses and joins, the data-centric approach shows superior performance with a runtime of 1.61 seconds, compared to 59.15 and 105.56 seconds for the decomposed and triple-store approaches. This performance translates to a relative speedup of a factor of 36 and 65, respectively (Figure 8(b)).

DBPedia. For DBPedia, this query accesses five RDF properties, and translates to *Display population information for all cities*. The triple-store approach involved five table accesses and four self-joins, with the decomposed approach using five table accesses and four joins. The data-centric approach used three table accesses and two joins. The runtimes for the decomposed and data-centric approaches are similar and showed sub-second performance. While the query times for the



(a) Query 2 Runtime (b) Query 2 Speedup

Fig. 9. Query 2



(a) Query 3 Runtime (b) Query 3 Speedup

Fig. 10. Query 3

decomposed and data-centric approaches were sub-second, the relative speedup of 14 in this case is large.

Uniprot. For Uniprot, this query accesses six RDF properties, and translates to *Show all ranges of transmembrane regions*. The data-centric approach required a total of five table accesses, where the majority of the joins were subject-to-subject, making extensive use of the clustered indices. Meanwhile, the decomposed approach required a total of six table accesses. The triple-store approach also used six table accesses, with five self-joins being generated over the table. The data-centric approach showed better relative performance, with a runtime of 18.9 seconds, compared to 23.92 seconds and 72.61 seconds over the decomposed and triple-store approaches, translating to a relative speedup of factors of 1.2 and 3.8, respectively.

Discussion. 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. For this query, the number of table-accesses and joins decreased significantly due to this storage scheme.

2) Query 2: Single subject/all defined properties: Query 2 involves a selection of all defined properties for a single RDF subject (i.e., a single record). Figure 9(a) gives the runtime for this query across all three data sets, while Figure 9(b) gives the relative speedup of the data-centric approach over the triple-store and decomposed method.

DBLP. For DBLP, this query accesses 13 RDF properties, and translates to *show all information about a particular conference publication*. 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, with run times of 8.84 seconds and 8.06 seconds, respectively. This similarity is 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. Thus, the reduction of joins in the data-centric method was offset by the overhead of storing extra property data stored in each tuple.

DBPedia. For DBPedia, this query accesses 23 RDF properties, and translates to *show all information about a particular cricket player*. Both 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. Thus, the speedup measures for the data-centric approach show a superior performance at 15.13 and 3K over and decomposed and triple-store approaches, respectively.

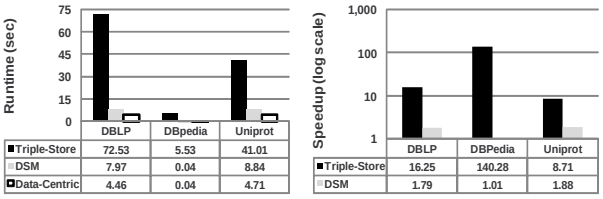
Uniprot. For Uniprot, this query accesses 15 RDF properties, and translates to *show all information about a particular protein*. The decomposed and triple-store approach involved fifteen table accesses along with fourteen subject-to-subject joins. For the triple-store, this fact is due to the need to select each property from the triple-store, plus the self join over the table to answer the query. For the decomposed approach, these accesses and joins were due to each property being stored in a separate table. Meanwhile, the data-centric approach involved 11 table accesses generating 10 subject-to-subject joins. Due to this fact, the data-centric approach shows better relative performance as given in Figure 9(a), with a runtime of 3.75 seconds, compared to 5.38 seconds and 15.51 seconds for the decomposed and triple-store approaches, respectively. This translates to a speedup of 4.14 and 1.44, respectively (Figure 9(b)).

Discussion. Overall, the data-centric approach shows better relative runtime performance to that of the other schema approaches for Query 2. This is mainly due to properties stored in the same n-ary table also being accessed together.

3) Query 3: Administrative query: Query 3 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. Figure 10(a) gives the runtime for this query across all three data sets, while Figure 10(b) gives the relative speedup of the data-centric approach over the triple-store and decomposed method.

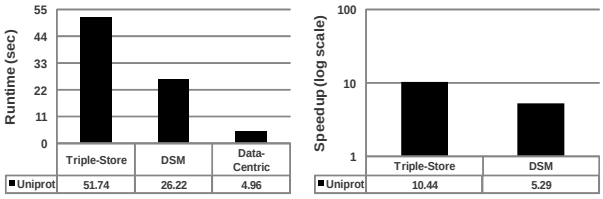
DBLP. This query accesses three RDF properties, and translates to *List title, author, and date of recently modified entries (recent is* $\geq$ *2005)*. The data-centric approach required a single table access, with all properties clustered to a single table. Both the decomposed and triple-store approaches required separate table accesses for the range selection and joins to retrieve all RDF properties. Thus, the data-centric approach shows a speedup over the decomposed and triple-store approaches of 24.8 and 21, respectively.

DBPedia. For DBPedia, this query accesses 23 RDF properties, and translates to *Show information for recently updated sports information (recent* $>$ *2006)*. The data-centric approach shows similar performance to the decomposed approach due to all data being stored in binary tables for both approaches.



(a) Query 4 Runtime (b) Query 4 Speedup

Fig. 11. Query 4



(a) Query 5 Runtime (b) Query 5 Speedup

Fig. 12. Query 5 - Reification

No discernible speedup is present between the data-centric and decomposed approaches, while the data-centric approach saw a speedup of 4.83 over the triple-store.

Uniprot. For Uniprot, this query accesses four RDF properties, and translates to *List version and creation date of recently modified entries (recent is $>$ 2002)*. One property (i.e., modified date) is used in the range selection. The data-centric approach required only two table accesses and one join. The decomposed approach used four table accesses, causing three joins, while the triple-store approach used four table accesses with three self-joins. The, the data-centric approach shows a speedup over the decomposed and triple store of 3.18 and 13.16, respectively.

Discussion. 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. The data-centric approach also showed good speedup for the semi-structured Uniprot data due to reduction of joins and table accesses.

4) Query 4: Predetermined props/spec subjects: Query 4 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). Figure 11(a) gives the runtime for this query across all three data sets, while Figure 11(b) gives the relative speedup of the data-centric approach over the triple-store and decomposed method.

DBLP. For DBLP, this query accesses five RDF properties, and translates to *Show author, conference, title, abstract, and year for all papers in SIGMOD,VLDB,ICDE (1999-2003)*. The data-centric approach stores all relevant properties in a single relation, thus producing a single table access and a selection using the IN operator. Meanwhile, both the decomposed and triple-store approaches involve five table accesses including three subject-to-subject joins and one subject-to-object joins, with the triple-store involving four self-joins. The data-centric approach had a runtime of 4.46 seconds, compared to 7.9 for the decomposed approach and 72.5 seconds for the triple-store approach. This performance translates to a speedup of 1.79 and 16.24, respectively.

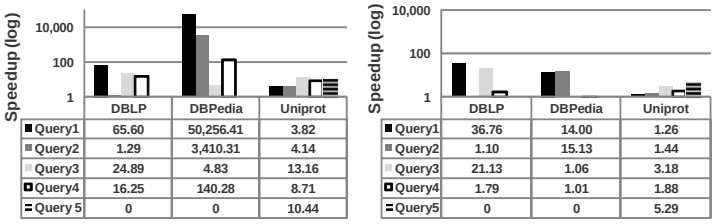
DBPedia. For DBPedia, this query translates to *Two degrees of separation from Kevin Bacon*, and involves one RDF property accessed a total of three times in order to find degrees of separation from a particular RDF subject. The data-centric and decomposed methods show similar performance due to the fact that both methods access a similar number of tables. Thus the speedup over the decomposed approach is minimal. However, the decomposed approach showed a factor of 140 speedup over the triple-store due to the high selectivity of the two self-joins.

Uniprot. For Uniprot, this query accesses four RDF properties, and translates to *Show all proteins associated with specific organisms and species of that organism*. The data-centric approach requires a total of two table accesses using one subject-to-object join, while the decomposed and triple-store methods involved four table accesses and three joins. For this query, the tables used in the data-centric approach contained extra properties. However, even with the extra storage the reduction in joins led to better a query runtime of 4.71 seconds, compared to 8.84 and 41 seconds for the decomposed and triple-store approaches. This performance translates to a speedup factor of 1.88 and 8.71, respectively.

Discussion. Again, the data-centric approach shows better overall performance to that of the other schema approaches. For the Uniprot and DBLP queries, the data-centric approach shows good speedup over the triple-store, with a factor of 1.8 speedup over the decomposed approach, as given in Figure 11(b). This query again shows the need for a data-centric schema creation approach that finds and stores related properties in the same table.

5) Query 5: Reification: Query 5 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, hit counts are stored on a subset of the statements using reification (much like the *certain* attribute discussed in Section V-B). Thus, all reification statements need to be found with the *object* property corresponding to a protein, along with the hit count (modeled as the *hits* property) for each statement. The results for this query are given in Figure 12, with Figure 12(a) giving the runtime, while Figure 12(b) displays the relative speedup over the decomposed and triple-store approaches. 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. Thus, the data-centric approach shows a speedup of 5.29 and 10.44 over the decomposed and triple-store approaches, respectively.

6) Relative Speedup: Figure 13(a) gives the relative speedup for the data-centric approach over the triple-store



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

Fig. 13. Relative Speedup

approach for each query and data set, while Figure 13(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 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. However, the speedup for Uniprot was still modest across the board.

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. Furthermore, our approach handles multi-valued properties and RDF *reification* effectively. 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. Results show that our data-centric approach shows large orders of magnitude performance improvement over the triple store, and speedup factors of up to 36 over the decomposed storage approach.

References

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

[2] "W3C Semantic Web Activity: http://www.w3.org/2001/sw/."

[3] W3C, "Semantic Web Education and Outreach Interest Group: Case Studies and Use Cases. http://www.w3.org/2001/sw/sweo/public/UseCases/."

[4] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Commun. ACM*, vol. 47, no. 3, pp. 45–47, 2004.

[5] J. S. Jeon and G. J. Lee, "Development of a Semantic Web Based Mobile Local Search System," in *WWW*, 2007.

[6] "FOAF Vocabulary Specification: http://xmlns.com/foaf/spec/."

[7] "Uniprot RDF Data Set: http://dev.isb-sib.ch/projects/uniprot-rdf/."

[8] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle, "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases," in *SemWeb*, 2001.

[9] N. Alexander and S. Ravada, "RDF Object Type and Reification in the Database," in *ICDE*, 2006.

[10] D. Beckett, "The Design and Implementation of the Redland RDF Application Framework," in *WWW*, 2001.

[11] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," in *ISWC*, 2002.

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

[13] S. Harris and N. Gibbins, "3store: Efficient bulk rdf storage," in *PSSS*, 2003.

[14] 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.

[15] J. J. Carroll, D. Reynolds, I. Dickinson, A. Seaborne, C. Dollin, and K. Wilkinson, "Jena: Implementing the semantic web recommendations," in *WWW*, 2004.

[16] K. Wilkinson, "Jena Property Table Implementation," in *SSWS*, 2006.

[17] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds, "Efficient RDF Storage and Retrieval in Jena2," in *SWDB*, 2003.

[18] 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.

[19] D. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "Scalable Semantic Web Data Management Using Vertical Partitioning," in *VLDB*, 2007.

[20] 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.

[21] 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.

[22] G. P. Copeland and S. N. Khoshafian, "A Decomposition Storage Model," in *SIGMOD*, 1985.

[23] A. Matono, T. Amagasa, M. Yoshikawa, and S. Uemura, "A Path-Based Relational RDF Database," in *ADC*, 2005.

[24] R. Angles and C. Gutierrez, "Querying rdf data from a graph database perspective," in *ESWC*, 2005.

[25] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in sql databases," in *VLDB*, 2000.

[26] S. Agrawal, V. R. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *SIGMOD*, 2004.

[27] S. B. Navathe and M. Ra, "Vertical partitioning for database design: A graphical algorithm," in *SIGMOD*, 1989.

[28] S. Papadomanolakis and A. Ailamaki, "Autopart: Automating schema design for large scientific databases using data partitioning," in *SSDBM*, 2004.

[29] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *VLDB*, 1994.

[30] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," in *ICDE*, 2001.

[31] "RDF Store Benchmarks with DBpedia: http://www4.wiwiss.fu-berlin.de/benchmarks-200801/."

[32] R. Agrawal and J. Kiernan, "An Access Structure for Generalized Transitive Closure Queries," in *ICDE*, 1993.