Application-Specific Schema Design for Storing Large RDF Datasets

Luping Ding[1], Kevin Wilkinson[2], Craig Sayers[2], Harumi Kuno[2]
HP Labs, Palo Alto
lisading@wpi.edu[1], {*firstName.lastName*}@hp.com[2]

Abstract: In order to realize the vision of the Semantic Web, a semantic model for encoding content in the World Wide Web, efficient storage and retrieval of large RDF data sets is required. A common technique for storing RDF data (graphs) is to use a single relational database table, a triple store, for the graph. However, we believe a single triple store cannot scale for the needs of large-scale applications. Instead, database schemas that can be customized for a particular dataset or application are required. To enable this, some RDF systems offer the ability to store RDF graphs across multiple tables. However, tools are needed to assist users in developing application-specific schema. In this paper, we describe our approach to developing RDF storage schema and describe two tools assisting in schema development. The first is a synthetic data generator that generates large RDF graphs consistent with an underlying ontology and using data distributions and relationships specified by a user. The second tool mines an RDF graph or an RDF query log for frequently occurring patterns. Knowledge of these patterns can be applied to schema design or caching strategies to improve performance. The tools are being developed as part of the Jena Semantic Web programmers' toolkit but they are generic and can be used with other RDF stores. Preliminary results with these tools on real data sets are also presented.

Key Words: RDF, Semantic Web, Schema Design, Storage Tuning, Data Mining, Sequential Pattern Mining, Synthetic Data Generation.
Category: H.2.1, H.2.2, H.2.8

1 Introduction

In order to realize the vision of the Semantic Web, a semantic model for encoding content in the World Wide Web, efficient storage and retrieval of large RDF data sets is required. Efficient storage is also needed if Semantic Web technologies are to be applied to other application domains such as enterprise integration, as discussed in [BL03]. Storing RDF in relational and object-relational database management systems has been the focus of much research. At first glance, a relational model is a natural fit for RDF. The RDF model is a set of statements about Web resources, identified by URIs. Those statements have the form $< S, P, O >$ and are interpreted as subject S has a predicate (property) P with a value O. So, an RDF model is easily implemented as a relational table of three columns. Many RDF storage systems have used this triple-store approach [HP 03] [ACK$^+$01] [KAO] [BK01] [TAP02].

In the absence of any other knowledge about the objects being stored, there are few options for storage optimization and a triple-store approach is reasonable. However, RDF often encodes higher-level constructs, such as lists, and higher-level semantics for ontologies, such as class hierarchies and constraints. In addition, real-world data and applications often have access patterns or structure. The basic triple store approach cannot leverage such higher-level knowledge about objects or their access patterns. For scalability and high-performance, we believe that the triple-store approach must be augmented by other storage strategies optimized for the data being stored or the applications that use it.

Previous work on RDF stores has used the RDF Schema class definitions and class hierarchies to derive an application-specific database schema, [ACK$^+$01] [KAO] [BK01]. However, this requires *a priori* knowledge about the data, which may not always be obtainable. In this paper, we consider how to derive an efficient storage schema without *a priori* knowledge.

To explore various layout options as well as understand performance tuning trade-offs in RDF stores, we are developing a set of tools for RDF data sets. In this paper, we describe two tools, one to analyze RDF graphs or queries for patterns, and another that generates synthetic RDF data sets. We have used these tools to assist in developing the persistence subsystem for the second generation of Jena, a leading Semantic Web programmers' toolkit [McB02] [CDD+03]. The Jena2 persistence subsystem [WSKR03] supports property tables and has various configuration parameters and our tools can help study trade-offs among the options. These tools are implemented as Jena applications but can be used with or easily ported to other RDF stores.

The rest of the paper is organized as follows: Section 2 introduces our approach to designing application-specific schema for RDF data. Section 3 defines the RDF data analysis problem and our data analysis tool. Some preliminary work in applying this tool to real RDF data sets is described in Section 4. Section 5 presents the RDF synthetic data generation tool. Section 6 explains the related work and we conclude our work in Section 7.

2 Application-Specific Schema Design

Most RDF stores have options and tuning parameters to influence the underlying storage engine. Understanding the interaction and trade-offs among these parameters is a complicated task. We believe iterative design and experimentation with test data can simplify the process and yield a robust schema design that performs well for the intended application. We are developing a set of tools to facilitate schema design and storage tuning. Figure 1 sketches our view of the application-specific schema design process and how our tools interact.

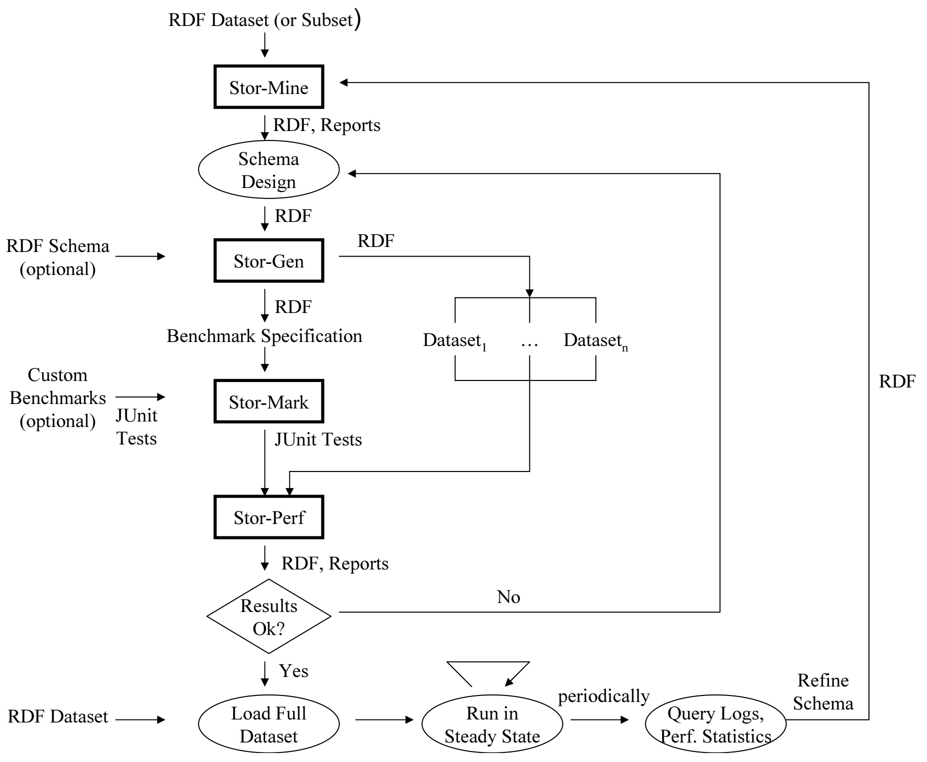

Figure 1: Iterative RDF Schema Design Process and Tools.

The process is summarized as follows. First, an initial schema design is developed based on analyzing the data to be stored. Second, synthetic data sets that model the data to be stored are generated. Also generated are benchmark programs to evaluate the performance of the schema on the synthetic data. Third, the benchmarks are run. If the performance is not acceptable, the schema design is adjusted and the process iterates.

We assume the existence of an initial data set on which we can apply our data analysis tool. If no such data set exists, the initial schema design is based solely on knowledge of the application and expected usage patterns. The analysis from *Stor-Mine* serves two purposes: (1) discovering features and characteristics of the graph to guide the data generator in generating more realistic synthetic data sets and (2) discovering patterns in graphs or queries that can be leveraged to improve performance.

For example, $person$ objects may always have properties like *family name*, *given name*, *phone* and *email*. *Stor-Mine* could reveal this regular structure. For such a graph, the property table shown in Table 1 might have a performance advantage for storing RDF statements about persons.

Subject	*FamilyName*	*GivenName*	*Phone*	*eMail*
ex:person1	Ding	Luping		lisading@WPI.EDU
ex:person2	Kuno	Harumi	123-456-7890	harumi.kuno@hp.com
ex:person3	Sayers	Craig		csayers@hpl.hp.com
ex:person4	Wilkinson	Kevin	123-555-7890	

Table 1: Property Table.

The *Stor-Gen* tool is used to generate synthetic data sets that model the actual data of an application. It is reasonable to ask why synthetic data is needed if actual data exists. Use of actual data presents two problems. First, the size of the actual data may not match the needs of the intended application. A robust schema design should perform well over a range of data set sizes. Synthetic data generation ensures we can work with whatever data set sizes are required. Moreover, simply sampling a large data set to create a smaller one is not valid because the sampling may lose relationships among the data items. Second, use of synthetic data ensures that the benchmark measures exactly the features of interest, while real world data usually has noise that makes interpreting results of the benchmarks difficult.

The *Stor-Mark* tool generates a benchmark suite that evaluates the performance of the application schema on the synthetic data sets. The benchmark specifications are user-provided as well as automatically derived from the *Stor-Gen* specifications. The *Stor-Perf* tool actually runs the benchmark and records the results. It is implemented as a JUnit [JUn] application. The entire process iterates until a schema design with acceptable performance is found.

Once a schema has been designed, the actual data is loaded and the application installed. Ideally, the application will have acceptable performance. However, if the synthetic data does not correctly characterize the actual data or if the benchmarks do not reflect the application, performance may be poor. Even if performance is good, user access patterns may change over time. Thus, it may be necessary to periodically tune or redesign the schema. Query logs and performance statistics can be fed back into the data analysis tool to further optimize the schema. To support this, *Stor-Mine* can find patterns in query logs as well.

3 RDF Data Analysis

Our goal is to analyze RDF data and queries to discover interesting patterns to help design and optimize application-specific schema. We now define four pattern discovery problems and describe our tool for addressing these problems.

3.1 RDF Pattern Discovery Problem

As mentioned earlier, an RDF graph is a set of statements of the form $< S, P, O >$, interpreted as subject S has property P with value O. See [KC02] for more details. For this work, we only consider static RDF graphs and assume that under steady-state conditions the statistical characteristics of the graph, for mining purposes, are stable (not always a valid assumption, of course). For RDF graphs, we aim to discover property co-occurrence patterns that would suggest possible properties to store together in a property table.

Problem PD1: Subject-property co-occurrence pattern discovery.

Informally, we want to find properties that are frequently defined for the same subject (e.g., name, address, etc.). We find the longest patterns with a minimum level of support (frequency of occurrence). The properties in such patterns are then candidates for grouping together in a property table. For this paper, we do not consider the (RDF) types for the subject as, in general, the types need not be defined. A formal definition of all four problem statements is given in [DWSK03] but is not included here due to space limitations.

3.2 RDF Query Pattern Discovery Problems

In this paper, we only consider mining of RDQL queries. RDQL, the query language for Jena2, is an implementation of the Squish query language [LM02]. An RDQL query can be represented as a conjunction of *triple patterns*, $TP_1 \wedge TP_2 \wedge ... \wedge TP_n$, where TP_i has the form $< S, P, O >$ and each element is either a constant, a variable or a don't-care. A triple pattern binds its unbound variables (elements) to statements that match its constant elements and its bound variables. The advantage of this triple-pattern representation is that the queries can be modeled as RDF statements (graphs).

A query log is then a time-ordered sequence of RDQL queries applied to an RDF graph. Given this log, we have identified three problems of interest.

Problem PD2: Single query pattern discovery.

Informally, we are given a log of queries $Q_1, Q_2, ..., Q_k$ where each Q_i is a conjunction of triple patterns as above. We assume that some queries are repeated (or are simple variants) and we want to find those commonly occurring queries.

To compare and match queries, they must be transformed because variable names are arbitrarily assigned (see [DWSK03] for details). For example, the two queries below have same pattern but different representations.

```
Query1: (Var1, VCARD:Name, -) (Var1, VCARD:Title, "professor")
Query2: (Var4, VCARD:Name, -) (Var4, VCARD:Title, "instructor")
```

In general, we look for the largest queries with a minimal level of support. Note that larger queries may subsume smaller queries, e.g., the query *(Var3, VCARD:Name, -)* is subsumed by those above. The frequently-occurring queries are then candidates for materialized views or indexes.

Problem PD3: Subject-property query pattern discovery.

Informally, we are looking for common triple patterns among all the queries. In other words, we decompose all queries into their individual triple patterns and look for properties that are commonly queried for the same subject. In the above example, we see a simple pattern of *VCARD:Name* and *VCARD:Title* being queried for a common subject. As before, we look for the largest set of properties with a minimum level of support. The properties within a pattern are then candidates for property tables or caching.

Problem PD4: Subject-property query sequence discovery.

Informally, we are looking for repeated sequences of queries (triple patterns) for a common subject. As in PD3, the queries are decomposed into their individual triple patterns, tagged by the query time (or the sequence number of the query). The triple patterns are then partitioned by subject. Only constant subjects are considered: partitions with variable and don't-care subjects are ignored. We then look for common sequences of properties that are queried for subjects.

The common sequences of properties found provide insight into the control flow of the program and assist in the design of cache prefetching strategies.

3.3 RDF Mining

Market-basket analysis.

We apply data mining techniques for *Market-Basket Analysis* [AIS93] to solve the above pattern discovery problems. These techniques are developed to find the *association rules* [AIS93] or *frequent item set sequences* [AS95] in a database of customer transactions to answer the question of which items are frequently purchased together. A savvy marketer can then use the knowledge of frequent item sets for marketing actions to achieve some business objective such as increase revenue, reduce inventory, etc.

The input data for association rule mining is a collection of $baskets$, each of which is a set of items purchased by a customer in one transaction (quantities of items are not considered). Two input parameters are the $support\ s$, which specifies the fraction of baskets that must contain the items in an association rule and $confidence\ c$ for rules (of the form "A implies B") which specifies the fraction of baskets that contain B, from among the baskets that contain A. See [AIS93] for further details on association rule mining.

The input data for sequential pattern mining is a collection of *customer-sequences*. Each customer-sequence is an ordered sequence of transactions made by a single customer over a period of time, where each transaction is, as before, an item set. The algorithm also takes a $support\ s$ parameter and looks for sequences supported by at least $(numCustomers * s)$ customers. A customer $supports$ a sequence seq if seq is contained in the customer-sequence for this customer. The readers are referred to [AS95] for further details.

Problem Mapping.

We can easily map our problems into a Market-Basket Analysis (MBA) framework. *Problems PD1*, *PD2* and *PD3* are mapped into *finding frequent item sets* in MBA, and *Problem PD4* is mapped into *finding sequential patterns* in MBA. In *Problem PD1*, each subject-property co-occurrence set acts as a basket, and in *Problem PD3*, each subject-property query set acts as a basket. The properties become the items contained in the basket.

In *Problem PD2*, the triple patterns within a query form a basket. But, as mentioned earlier, the triple patterns are transformed to unify the variable names (see [DWSK03]). In *Problem PD4*, the subject-property query sequence acts as customer-sequence. However, unlike customer-sequence, each item set in property query sequence only contains one item, which is a property.

3.4 Stor-Mine: RDF Data Analysis Tool

Our RDF data analysis tool *Stor-Mine* can search an RDF graph or RDQL query log for patterns that could be used to derive an optimized storage schema for the data. *Stor-Mine* (Figure 2) consists of three main modules: *Data pre-processing*, *basket derivation* and *pattern discovery*. *Data pre-processing* cleans the original RDF graph or query log by filtering out data that cannot currently be exploited by our algorithms (e.g., properties for *rdf:Bag* such as *_1*, *_2*, *etc.*, see [DWSK03]). *Basket derivation* takes these RDF statements and transforms them into suitable ”baskets”, thus preparing the input for mining. The *pattern discovery* module includes the largely application-independent use of generic data mining techniques such as discovery of frequent item sets, association rules and sequential patterns.

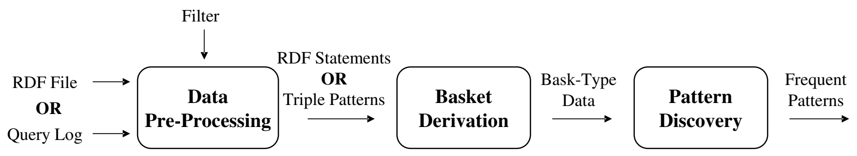

Figure 2: Architecture of Stor-Mine.

We have incorporated implementations of many commonly used data mining algorithms such as Apriori, FPgrowth, etc. and also our own implementation of some mining algorithms, for example, AprioriAll. Currently, the discovery of frequent property co-occurrence pattern, frequent property query pattern and frequent single query pattern uses the FPgrowth algorithm [HPY00], a very fast algorithm for finding the large frequent item sets. Sequential property query pattern discovery uses the AprioriAll algorithm introduced in [AS95].

4 Experiments

4.1 Initial Experiment: Mining MusicBrainz

A preliminary experiment with *Stor-Mine* was performed on a subset of MusicBrainz [Mus], an open-source database about music compact discs. MusicBrainz is stored in a relational database but is exported as an RDF graph. Consequently, it is highly structured.

MusicBrainz consists of 3 classes: Artist, Album, Track. Artist identifies an artist and includes a list of references to the albums by that artist. Album identifies an album and has a reference to the artist and a list of references to the tracks on the album. Track identifies one track in an album and references the artist. *Stor-Mine* should be able to recover this class structure from the RDF graph.

Our sampled subset contains about 200,000 statements which reduces to 50,000 after filtering. It contained roughly equal numbers of Artist, Album and Track class instances. The complete data set has more than an order of magnitude more Track instances than Artists. So, ours was an artificial experiment but a good baseline comparison.

On current technology PCs, *Stor-Mine* takes about 420 milliseconds on this dataset. Different values of support were tried and results for two levels are shown in Table 2 and Table 3.

	$Property_1$	$Property_2$	$Property_3$	$Property_4$	$Comment$
$Pattern_1$	rdf:type	title	creator	duration	Track instance
$Pattern_2$	rdf:type	title	sortName	albumList	Artist instance
$Pattern_3$	trackNum				Track number

Table 2: MusicBrainz Property Co-occurrence Patterns (Min Support: 0.05).

	$Property_1$	$Property_2$	$Property_3$	$Property_4$	$Property_5$	$Comment$
$Pattern_1$	rdf:type	title	creator	duration	trmidList	Track instance
$Pattern_2$	rdf:type	title	sortName	albumList		Artist instance
$Pattern_3$	trackNum					Track number
$Pattern_4$	rdf:type	title	creator	release	trackList	Album instance

Table 3: MusicBrainz Property Co-occurrence Patterns (Min Support: 0.01).

Stor-Mine run with a higher support level (5%) uncovered the Track and Artist classes but not the Album class. The trackNum property is actually a property of a Track class. The reason it appears separately rather than with the other Track properties (Pattern1) is because the data was not correctly sampled (i.e., the Track instance for these trackNum properties were not in the sample so they appear as separate classes).

Note that there are many more trackNum properties than Album instances which is why the trackNum frequent item set is found before the Album frequent item set. *Stor-Mine* run with a lower support level (1%) does uncover the Album class. It also finds an additional property for the Track class, trmidList, that does not appear at higher support levels.

Stor-Mine is able to distinguish overlapping uses of the same property in different classes (title and creator). However, a low support level was required to find all the classes. And this was with an artificial sample with roughly equal numbers of class instances. The complete data set has more than ten times as many Track instances as Artists, requiring an even lower support level to find all three classes. Consequently, a more sophisticated approach is needed.

Remaining Problems.

In mining the MusicBrainz data, we found the following issues that need to be addressed in the future. Our current heuristic is to look for the longest frequent property sets that have a minimal level of support. However, we cannot simply use the longest frequent item set because the dataset may have groups of properties that overlap but are actually distinct clusters. For example, the properties $name$, $address$ and $phone$ may be used both for a Person class and a Business class. Yet properties like *spouse, height, weight, number-of-employees, revenue* are particular to one class or the other. *Stor-Mine* should be able to distinguish such clusters and not generate a large frequent property set which is the union of them.

On the other hand, we cannot simply use the frequent property set with the highest support. This is because optional values may result in overlapping item sets. Consider a dataset of properties about people containing a number of gender-specific properties. The mining algorithm would produce two frequent items sets, one for males and one for females, where the support of each depends on the proportion of males and females in the dataset. But, what is really wanted is just a single property table for person.

Other RDF metadata such as *rdf:type* can also be useful in mining an RDF graph, e.g., varying the support level per type. However, we cannot depend on the existence of type information in an RDF graph. One goal of our initial experiment was to determine the effectiveness of *Stor-Mine* without special treatment for *rdf:type*. As an aside, we note that an RDF graph in which every statement is reified will have just one pattern. In the future we will also study special treatment for reification.

4.2 Mining Haystack Trace

In order to explore our analysis techniques over less highly structured graphs, we ran *Stor-Mine* over a Haystack trace [Qua]. Haystack [QHK03] is a personal knowledge management system. It uses RDF for all its internal objects, including the GUI. The Haystack trace is an RDF graph of 109962 statements (triples) and a query log containing 51844 queries [1]. Before doing each kind of pattern discovery, we pre-process the data into appropriate format to feed into the mining algorithm.

Mining RDF Graph

The RDF graph was analyzed to look for frequent subject-property co-occurrence patterns as described in section 3. Due to space limitations, these results are not shown here (see [DWSK03] for details). However, unlike the MusicBrainz results, there were few clear-cut candidates for property tables and many patterns with only two properties, one of which was *rdf:type*. This suggests it may be beneficial to mine this data by class (rd:type), which is future work.

Mining Query Log

The Haystack query log has 51844 queries which were decomposed into 53500 triple patterns. This implies that most queries only contain a single triple pattern (few joins).

Discover single query patterns.

As the first step, we search for the query patterns that are frequently asked (problem PD2).

The results are shown in Table 4. All the frequent query patterns along with their support and the number of triples contained are listed. "C" means constant, "V" along with a number means some specific variable and "-" means don't-care. The same variable used in multiple triples within one query implies a join. From the table, we observe the following:

1. Most queries are single property value retrieval, that is, retrieve the value of a specific property of the given subject, represented as (C, $property$, –), e.g., pattern Q01.
2. A few queries search for subjects with a given property value, represented as (–, $property$, C), for example, the triple (–, ozone:dataDomain, C) in pattern Q11.
3. Few queries contain a join pattern, e.g., pattern Q47.

[1] The Haystack queries had to be transformed into an RDQL triple-pattern representation.

ID	*Support*	*Pattern*
Q01	0.134	[(C, config:singleton, -)]
Q02	0.113	[(C, config:hostsService, C)]
Q03	0.048	[(C, ozone:registerService, -)]
Q04	0.043	[(C, ozone:registerToolbar, -)]
Q05	0.040	[(C, rdf:type, C)]
Q06	0.038	[(C, rdf:type, -)]
Q07	0.032	[(C, dc:title, -)]
Q08	0.031	[(C, rdfs:label, -)]
Q09	0.024	[(C, haystack:javaImplementation, -)]
Q10	0.024	[(C, haystack:className, -)]
Q11	0.021	[(-, ozone:dataDomain, C)]
Q12	0.014	[(C, ozoneslide:bgcolor, -)]
Q13	0.014	[(C, ozone:onEnterPressed, -)]
Q14	0.014	[(C, ozone:tooltip, -)]
Q15	0.014	[(C, ozone:onClick, -)]
Q16	0.014	[(C, ozone:putProperty, -)]
Q17	0.014	[(C, ozone:putLocalProperty, -)]
Q18	0.013	[(C, content:path, -)]
Q19	0.013	[(C, daml+oil:rest, -)]
Q20	0.013	[(C, daml+oil:first, -)]
Q21	0.011	[(C, ozoneslide:fontSize, -)]
Q22	0.011	[(C, ozoneslide:alignX, -)]
Q23	0.011	[(C, ozoneslide:linkColor, -)]
Q24	0.011	[(C, ozoneslide:fontBold, -)]
Q25	0.011	[(C, ozoneslide:color, -)]
Q26	0.011	[(C, ozoneslide:alignY, -)]
Q27	0.011	[(C, ozoneslide:style, -)]
Q28	0.011	[(C, ozoneslide:fontFamily, -)]
Q29	0.008	[(C, ozone:dataSource, -)]
Q30	0.007	[(V1, rdf:type, C), (V1, ozone:viewDomain, V6), (C, haystack:classView, V6)]
Q31	0.006	[(C, rdfs:subClassOf, -)]
Q32	0.005	[(C, haystack:JavaClass, -)]
Q33	0.005	[(C, adenine:debug, -)]
Q34	0.005	[(C, adenine:preload, -)]
Q35	0.005	[(V1, rdf:type, C), (V1, ozone:viewDomain, C)]
Q036	0.005	[(C, ozone:connector, -)]
Q37	0.004	[(C, ozoneslide:textAlign, -)]
Q38	0.004	[(C, summaryView:titleSlide, -)]
Q39	0.004	[(-, information:knowsAbout, C)]
Q40	0.004	[(C, ozoneslide:defaultText, -)]
Q41	0.004	[(C, ozoneslide:text, -)]
Q42	0.004	[(C, ozoneslide:wrap, -)]
Q43	0.004	[(C, config:dependsOn, -)]
Q44	0.003	[(C, config:includes, -)]
Q45	0.003	[(C, dc:description, -)]
Q46	0.003	[(C, haystack:md5, -)]
Q47	0.003	[(C, rdf:type, V1), (V1, summaryView:titleSourcePredicate, -)]
Q48	0.003	[(C, ozoneslide:borderLeftWidth, -)]
Q49	0.003	[(C, ozoneslide:marginY, -)]
Q50	0.003	[(C, ozoneslide:marginX, -)]
Q51	0.003	[(C, adenine:main, -)]
Q52	0.003	[(C, ozoneslide:marginBottom, -)]
Q53	0.003	[(C, ozoneslide:borderWidth, -)]
Q54	0.003	[(C, ozoneslide:borderColor, -)]
Q55	0.003	[(C, ozoneslide:marginLeft, -)]
Q56	0.003	[(C, ozoneslide:margin, -)]
Q57	0.003	[(C, ozoneslide:marginTop, -)]
Q58	0.003	[(C, ozoneslide:marginRight, -)]
Q59	0.003	[(C, ozoneslide:borderBottomWidth, -)]
Q60	0.003	[(C, ozoneslide:borderTopWidth, -)]
Q61	0.003	[(C, ozoneslide:borderRightWidth, -)]
Q62	0.003	[(C, dataProvider:predicate, -)]
Q63	0.003	[(C, ozoneslide:child, -)]
Q64	0.003	[(C, haystack:view, -)]

Table 4: Frequent Query Patterns (Min Support: 0.003).

The frequency of many simple queries suggests a design pattern in which the program flow is determined by simple queries on some subject. In effect, the sequence of queries is a trace of program flow. This motivated a second set of experiments on discovering frequent subject-property query patterns. If frequent access patterns to common properties are discovered, this suggests a caching strategy that eagerly retrieves the common property values may be beneficial.

ID	*Support*	*Pattern*
PQ01	0.246	[config:singleton]
PQ02	0.064	[haystack:javaImplementation]
PQ03	0.063	[content:path, rdf:type, dc:description, config:includes, config:dependsOn, haystack:md5, adenine:main]
PQ04	0.043	[rdf:type, ozone:onClick, ozone:onEnterPressed, ozone:tooltip, ozone:registerService, ozone:putProperty, ozone:putLocalProperty, ozone:registerToolbar, ozoneslide:style, ozoneslide:fontFamily, ozoneslide:fontSize, ozoneslide:fontBold, ozoneslide:color, ozoneslide:bgcolor, ozoneslide:linkColor, ozoneslide:alignX, ozoneslide:alignY, ozoneslide:children]
PQ05	0.043	[haystack:className]
PQ06	0.041	[rdf:type, ozone:onClick, ozone:onEnterPressed, ozone:tooltip, ozone:registerService, ozone:putProperty, ozone:putLocalProperty, ozone:registerToolbar, ozone:dataSource]
PQ07	0.040	[dataProvider:predicate]
PQ08	0.037	[rdf:type, ozone:onClick, ozone:onEnterPressed, ozone:tooltip, ozone:registerService, ozone:putProperty, ozone:putLocalProperty, ozone:registerToolbar, ozone:connector]
PQ09	0.037	[adenine:preload, adenine:debug, haystack:JavaClass]
PQ10	0.034	[rdf:type, haystack:view, rdfs:label, dc:title, summaryView:titleSlide]
PQ11	0.031	[ozone:icon]
PQ12	0.030	[ozone:putProperty, ozone:putLocalProperty, ozone:registerToolbar, ozone:viewPartClass]

Table 5: Frequent Property Querying Patterns (Min Support: 0.030).

Discover subject-property query patterns.

The previous result shows that Haystack generates many simple queries for a subject. In this step, we search for frequently occurring triple patterns for a common subject (problem PD3).

To conserve memory, we filter out baskets whose size exceeds some threshold. In particular, a few subjects have a very large number of properties. We also filter out triples with a *don't-care* as a subject. Hence the subject is either a bound variable or a constant. Table 5 shows the result of this experiment, in which the minimal support is set to 0.03 and all the baskets larger than 40 are ignored. A total of 7599 triple patterns are ignored.

In this experiment, we get a number of long property querying patterns. However, a significant portion of them overlap, e.g., patterns PQ04, PQ06 and PQ08. And the 12 frequent patterns cover a large fraction of the total queries, approximately 70%. This suggests that a property table may have a big benefit because it can save resources by reducing the number of stored records and by optimizing queryies by enabling a property value pre-fetching strategy.

From this experiment, we can derive the frequently queried properties which belong to the same subject. But the order of the queries is ignored. Also by looking at the long patterns such as PQ04, PQ06 and PQ08, we guess that the queries included in these patterns are issued in some fixed order. To verify this, we need to run a sequential pattern mining algorithm to search for frequent sequential query patterns. This remains as future work.

5 RDF Data Generation

5.1 Stor-Gen: Synthetic RDF Data Generator

Stor-Gen is an RDF data generator that generates large volumes of synthetic RDF data. The generated data is useful in experiments to measure the effectiveness of different storage schema, to evaluate different processing algorithms or to run comparative benchmarks. *Stor-Gen* allows a high degree of control over the characteristics of the generated data. Moreover, unlike many other synthetic data generators [ANZ01], *Stor-Gen* is capable of modeling relationships among class instances.

In its current implementation, Stor-Gen generates class instances for specified classes. Each class instance contains a set of property values of the properties specified for that class. This was our immediate need. In the future, we aim to extend *Stor-Gen* to generate arbitrary RDF data sets, i.e., statements about resources with arbitrary class memberships.

For each property of a class, the cardinality, i.e., the number of property instances, can be either a fixed value or a range which conforms to a random distribution function, called a *property cardinality distribution*. The range may include zero which makes the property optional for the class. Therefore, different instances of a class may have identical, overlapping or disjoint sets of properties.

The values of a property for a class conform to a random distribution function called a *property value distribution*. The value can be either a literal (e.g., integers, strings) or a resource (URIs). Currently, the values for a given property must be homogeneous, that is, either all literal values or all URIs. The distribution functions supported for both property cardinality and property value distributions include constant, uniform, Gauss (Normal), Poisson and Zipf. For generating random values from a Zipfian distribution, we use the technique described in [GSE$^+$94].

Currently, string values are either fixed-length and generated by choosing random letters from an alphabet, or generated by choosing random words from an existing vocabulary. String literals may include a language tag and English and Chinese[2] literals are supported. More languages may easily be supported.

Stor-Gen can also generate property values of composite type, for example, blank nodes. A blank node is considered as an anonymous class instance. This is useful for complex object values that have sub-components, such as address. The container structures such as Bag, Seq and Alt are also supported and they can be repeated and nested.

$Resource$ type values may reference other generated class instances or external resources. Properties of a class may also be self-referencing. This way trees of references can be generated to model relationships such as taxonomies, ancestors, sub-parts. More complicated chains of references such as the Markov chains in [ANZ01] remain future work.

Stor-Gen can generate interesting reference patterns by varying the order in which the tree properties are generated. We illustrate this by an example. For simplicity, assume that each class instance appears exactly once in the tree and that the tree is symmetric. Therefore, we have a tree representing some property relationship (e.g., ancestor) and a set of class instances that must be assigned to some node on the tree. The instances could be assigned randomly. Or the instances could be assigned in order by either a depth-first or breadth-first traversal of the tree. Consider the trees in Figure 3 where the numbers might refer to instances of a Person class and the links might represent the Child-of property. Now consider a benchmark query that finds all descendants of a randomly chosen resource (tree node). If the resource is chosen uniformly among all class instances, we do not expect much difference between the depth-first and breadth-first trees. But, if the resource is chosen by a Zipf distribution, depth-first and breadth-first have different performance.

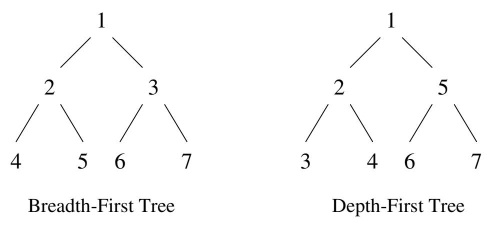

Figure 3: Referencing a Tree Property.

This is because a $Zipf$ distribution chooses higher-ranked instances more frequently. For the trees in Figure 3, assuming highest rank means 1 and lowest rank is 7, then the breadth-first tree would tend to return instances at the top of the tree. The depth-first tree would tend to return instances on the left side and bottom of the tree. Consequently, the benchmark query, on average, would run longer (traverse more nodes) when run on the breadth-first tree than on the depth-first tree. The ability to control such interactions is very useful in benchmark studies, e.g., consider studies of caching.

Stor-Gen can generate useful sized data sets in a reasonable amount of time. For example, 100,000 Dublin Core documents of 10 properties can be generated in less than a minute on current generation PCs when the dataset can fit in physical memory. In the future, *Stor-Gen* will be modified to optimize the creation of large data sets that do not fit in physical memory.

The architecture of *Stor-Gen* is shown in Figure 4. First, the user provides a *Stor-Gen specifi-*

[2] Currently, only a subset of all Chinese characters are supported.

cation file (in RDF) describing the characteristics of the generated data, in particular, the classes and properties and distributions desired. *Stor-Gen Configurator* takes this specification, referencing other existing ontologies if necessary, and then generates a generator hierarchy containing configured class generators, each one corresponding to a class. Then the *RDF Graph Generator* use the generator hierarchy to generate the RDF graph. The RDF graph can be directly pipelined to the application that uses it or be fed into *RDF Model Writer* to be serialized into an RDF file.

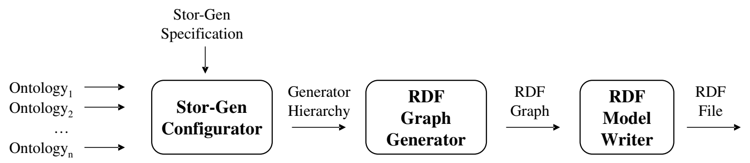

Figure 4: Architecture of Stor-Gen.

5.2 Stor-Gen Specification

To simplify the specification, *Stor-Gen* can use properties from existing ontologies. This reduces the amount of information a user must provide in the specification. The ontologies may be simple vocabularies or rich OWL ontologies. For example, $domain$ and $range$ properties in *RDF Schema* ontology determine which properties to generate for a given class. Also, the existing OWL $cardinality$ restrictions determine the number of property values to generate for a class instance. Statements in the *Stor-Gen* specification may augment or override the existing ontology specification, e.g., despite the ontology, someone may really want their document class instances to have multiple titles. However, the user is responsible for ensuring that property specifications among the ontologies do not conflict with each other.

We acknowledge that our interpretation of OWL cardinality constraints in this way is not consistent with OWL semantics. For example, suppose property X is declared as single-valued. In OWL, the following two RDF statements are not necessarily contradictory:

```
Subject1    PropertyX    Object1
Subject1    PropertyX    Object2
```

In OWL, these statements are interpreted to mean Object1 and Object2 are equivalent. However, for our purposes, we interpret a cardinality constraint to mean the number of occurrences of a particular property for a common subject. We believe this is a common case, especially for RDF data sets derived from existing relational data sources.

The specification, which is also an RDF file, specifies which classes, among all the classes in the ontologies, to generate and how many instances of each class to generate. For each class, a user needs to specify which properties to generate as well as the property cardinality, property value type and property value distribution of each property. If the property value is blank node, it is specified in a similar way to specifying a class. Note that blank node and container structures may be nested.

To generate a resource reference, the specification must include the cardinality of the referenced class. The property value distribution then generates a number in that range. This number is then appended to the class URI to form the resource reference. When referencing a resource related by a tree property, we use a Zipf distribution to choose the resource.

6 Related Work

A good introduction to RDF storage subsystems and a comparative review of implementations is available in [Bec01] [BG01]. Jena1 experimented with property tables [Rey03]. Other RDF stores create application-specific storage schema from knowledge of the RDF Schema class hierarchies (Forth [ACK+01], Sesame [BK01], KAON [KAO]). However, for arbitrary graphs without RDF Schema, they simply use a triple store.

Many pattern discovery problems for semi-structured data have been defined and addressed in the literature. [TSA+01] defines the problem of *mining path expression patterns* in semi-structured text. These patterns can be useful for deciding which portion of a given data is important. However, it only considers tree-structured HTML/XML. Mobasher et al proposed WEB-MINER [MJHS96], a web mining system which applies data mining techniques to analyze web server access logs. They also defined transaction models for various web mining tasks. Different from our work, they are only interested in discovering the relationships among the accessed files instead of document structure or query patterns.

To validate new ideas and evaluate system performance in database research, many benchmark systems have been developed for generating synthetic data for a broad spectrum of queries. Techniques for efficient generation of very large relational data sets are described in [GSE+94]. This work also includes an algorithm for generating Zipfian distributions which *Stor-Gen* uses. In a semi-structured data context, [ANZ01] describes a synthetic XML data generator that generates tree-structured XML data and allows for a high level of control over the characteristics of the generated data such as element frequency distribution, tag name recursion and repetition, etc. Another XML benchmark, XMark [SWK+01], generates XML documents and XQueries that model real-world applications. The generated XML data conforms to some fixed schema. Our work is different in that *Stor-Gen* generates graph-structured RDF statements that model class instances and relationships among class instances.

7 Conclusion

In summary, we believe that application-specific storage schema are required to realize efficient stores for large-scale Semantic Web applications. Also needed are tools and utilities to help derive and evaluate these storage schemes. We sketch a process for deriving application-specific storage schema for RDF and a set of tools used for this process. For RDF data analysis, we propose four types of analysis, provide algorithms for the analysis and implement these algorithms in our *Stor-Mine* tool. We also describe a synthetic RDF data generator tool, *Stor-Gen*, that can be used to generate RDF data sets to model real application data.

In the future, we plan to address the remaining problems of RDF mining as described in Section 3. We will enhance the RDF data generator to generate resource references with arbitrary class memberships. We also plan to implement other tools for use in the schema design process, i.e. *Stor-Mark* and *Stor-Perf*, and to explore the effectiveness of our application-specific schema design methodology.

Acknowledgements

The authors wish to thank Jeremy Carroll, Ian Dickinson, Dave Reynolds and Andy Seaborne for their helpful comments on an earlier draft of this paper. Umesh Dayal provided valuable feedback on the design of Stor-Mine. Bernard Burg and Kevin Smathers provided helpful suggestions on the design of Stor-Gen.

References

[ACK+01] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, and K. Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *2nd Intl Workshop on the Semantic Web (SemWeb'01, with WWW10)*, pages 1–13, Hong Kong, May 2001.

[AIS93] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993. http://citeseer.nj.nec.com/agrawal93mining.html.

[ANZ01] A. Aboulnaga, J. Naughton, and C. Zhang. Generating synthetic complex structured XML data. In *Proc. 4th Int. Workshop on the Web and Databases (WebDB'2001)*, May 2001.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In Philip S. Yu and Arbee S. P. Chen, editors, *Eleventh International Conference on Data Engineering*, pages 3–14, Taipei, Taiwan, 1995. IEEE Computer Society Press. http://citeseer.nj.nec.com/agrawal95mining.html.

[Bec01] D. Beckett. SWAD-Europe: Scalability and storage: Survey of free software / open source RDF storage systems, 2001. http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/.

[BG01] D. Beckett and J. Grant. SWAD-Europe: Mapping semantic web data with RDBMSes, 2001. http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/.

[BK01] J. Broekstra and A. Kampman. Sesame: A generic architecture for storing and querying RDF and RDF schema, October 2001. http://sesame.aidministrator.nl/publications/del10.pdf.

[BL03] T. Berners-Lee. Web Services–Semantic Web. Keynote Speech at World Wide Web Conference, 2003. http://www.w3.org/2003/Talks/0521-www-keynote-tbl/.

[CDD+03] J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. The Jena Semantic Web Platform: Architecture and design. Technical report, Hewlett Packard Laboratories, Palo Alto, California, 2003.

[DWSK03] L. Ding, K. Wilkinson, C. Sayers, and H. Kuno. Application-specific schema design for storing large RDF datasets. Technical Report HPL-2003-170, Hewlett Packard Laboratories, Palo Alto, California, 2003.

[GSE+94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billionrecord synthetic databases. *SIGMOD Record*, 23(2):243–252, June 1994.

[HP 03] HP Labs Semantic Web Research, 2003. http://www.hpl.hp.com/semweb/.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, May 2000. http://citeseer.nj.nec.com/han99mining.html.

[JUn] JUnit. http://www.junit.org/.

[KAO] KAON - the Karlsruhe Ontology and Semantic Web Tool Suite. http://kaon.semanticweb.org/.

[KC02] G. Klyne and J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts, November 2002.

[LM02] A. Reggiori L. Miller, A. Seaborne. Three implementations of SquishQL, a simple RDF query language. Technical Report HPL-2002-110, Hewlett Packard Laboratories, Palo Alto, California, 2002.

[McB02] B. McBride. Jena. In *IEEE Internet Computing*, July/August 2002.

[MJHS96] B. Mobasher, N. Jain, E. Han, and J. Srivastava. Web mining: Pattern discovery from World Wide Web transactions. Technical Report TR-96050, University of Minnesota, 1996. http://citeseer.nj.nec.com/mobasher96web.html.

[Mus] MusicBrainz. http://www.musicbrainz.org/.

[QHK03] D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user Semantic Web applications. In *The Twelfth International World Wide Web Conference*, 2003.

[Qua] D. Quan. SQL-Encoded trace of Haystack, by personal communication. http://www.ai.mit.edu/people/dquan/.

[Rey03] D. Reynolds. Jena Relational Database interface - performance notes (in Jena 1.6.1), 2003. http://www.hpl.hp.com/semweb/download.htm/.

[SWK+01] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, CWI, Amsterdam, the Netherlands, April 2001.

[TAP02] TAP project, 2002. http://tap.stanford.edu/.

[TSA+01] K. Taniguchi, H. Sakamoto, H. Arimura, S. Shimozono, and S. Arikawa. Mining semi-structured data by path expressions. *Lecture Notes in Computer Science*, 2226:378, 2001. http://citeseer.nj.nec.com/taniguchi01mining.html.

[WSKR03] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF storage and retrieval in Jena2. In *First Intl Workshop on Semantic Web and Databases (SWDB'03, with VLDB03)*, Berlin, September 2003.