Supporting Scalable, Persistent Semantic Web Applications

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

Abstract

To realize the vision of the Semantic Web, efficient storage and retrieval of large RDF data sets is required. A common technique for persisting RDF data (graphs) is to use a single relational database table, a triple store. But, we believe a single triple store cannot scale for large-scale applications. This paper describes storing and querying persistent RDF graphs in Jena, a Semantic Web programmers' toolkit. Jena augments the triple store with property tables that cluster multiple property values in a single table row. We also describe two tools to assist in designing application-specific RDF storage schema. The first is a synthetic data generator that generates RDF graphs consistent with an underlying ontology. The second mines an RDF graph or an RDF query log for frequently occurring patterns. These patterns can be applied to schema design or caching strategies to improve performance. We also briefly describe Jena inferencing and a new approach to context in RDF which we call snippets.

1 Introduction

This paper addresses the problems of storing and retrieving Resource Description Framework (RDF) graphs, the data model used for the Semantic Web. It describes RDF persistence in Jena, a widely-used, open-source Semantic Web programmers' toolkit [9, 18]. This paper assumes no familiarity with the Semantic Web and this introduction provides an overview of RDF [15]. The next section describes how Jena implements RDF persistence. The following sections outline Jena query processing and our support for developing application-specific RDF storage schema. A final section discusses the higher-level issues of inferencing and context.

The underlying data model of the Semantic Web, RDF, is a labeled, directed graph. This graph describes properties of and relationships between Web resources. An RDF graph is encoded as a set of triples where each triple encodes one arc of the graph. A triple, also referred to as a statement, has the form <*S,P,O*>, where S, the subject, and O, the object, are nodes of the graph and P, the predicate, labels an arc from S to O. The subject and predicate are always URIs while the object may be either a literal value or a URI. RDF has the notion of an anonymous resource, also called a blank node or bnode. It denotes an existential variable which is used to indicate a resource whose identity (URI) is unknown or not of interest. A bnode can be used to model n-ary relationships and to model containment (is-part-of). For example, to model a compund name, a bnode can be used to represent an (anonymous) name resource with two arcs to literal nodes for the first name and last name.

Just as LISP builds on a simple list abstraction to create higher-level constructs, RDF builds on predicates and the statement abstraction to model higher-level constructs. RDF includes predefined predicates such as *type*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

to state that the subject is in the class referenced by the object. Other predefined predicates model bags, lists, etc. Of particular interest is the notion of *reification*. It gives identity to a statement and so enables one to make statements about a statement. It uses the predefined predicates *subject, predicate, object, type*. For example, to reify the statement *<sky,is,blue>* so that we can refer to it from other statements, we first need a URI or bnode to identify the statement, say *X*. We then encode the reified statement with four separate statements as follows: *<X,subject,sky>, <X,predicate,is>, <X,object,blue>, <X,type,Statement>*. Given this, to express "Alice said the sky is blue", we add the statement *<Alice,said,X>*. Reification enables an RDF graph to include seemingly contradictory assertions, e.g., by reifiying *<sky,is,green>*.

RDFS (RDF Schema [7]) builds on RDF by adding more predefined predicates to increase the modeling power of the graph. RDFS includes a *subclass* predicate to support class hierarchies and a *sub-property* predicate. OWL (Ontology Web Language [19]) adds another modeling layer with predicates for ontologies and reasoning, e.g., to describe transitive properties and functional predicates. It is largely derived from DAML+OIL [10] and is based on Description Logics. However, note that OWL, RDFS and RDF graphs all have the same representation, i.e., a set of triples (statements).

There are a number of toolkits that implement the RDF model. Although there is no standard API such as ODBC, most RDF systems support methods to add a statement to a graph, remove a statement from a graph and to query graphs. In addition, they usually support some graph operations such as graph creation, graph merging, graph serialization, etc. However, the RDF specifications have no explicit notion of graph. So issues like graph identity, querying across graphs, distributing graphs are handled differently (or not at all) across the toolkits.

There is currently no standard query language for RDF. We describe here the Jena query capabilities which are typical of many RDF toolkits. Jena supports two forms of querying: triple match and RDQL [16]. A *triple match* takes a match template of the form *(s,p,o)*, where each term is either a constant or a don't-care. It returns all statements that match the template. For complex queries Jena supports RDQL, a declarative RDF query language that has been widely adopted. An RDQL query may be expressed as a conjunction of triple match templates in which a match term may also be a variable. The variables enable joins across match templates. Additional constraint clauses can be used to limit the value range of variable bindings using arithmetic and string processing functions. The query returns all bindings of variables that satisfy the triple matches.

2 Persistent RDF Storage in Jena

A common approach to making RDF graphs persistent is to use a relational database and store the graph in a three column table, one column each for the subject, predicate and object of a statement. Many RDF storage systems use this *triple store* approach [4, 8, 14]. Typically, a normalized schema is used in the triple store approach which adds two additional tables. A literals table is used to store all literal values and a resources table is used to store all URI and bnodes. The columns of the triple store then reference values in the literals and resources tables. The normalized triple store approach is very efficient in space since a literal or resource is stored only once. However, retrieving a statement requires a three way join.

There are challenging problems in using relational databases for storing RDF. For example, the character lengths of the literals and URIs in RDF graphs may vary widely and may even be blobs. The object of an RDF statement is not strongly typed and may either be a literal or a URI. An additional (boolean) column is sometimes used to indicate which it is. An RDF literal may optionally have an XSD datatype [5] or even a user-defined type. Consequently, there is no database column type that is suitable for all literal values. Typically, implementations simply store the value as a character string. Some systems optionally store the literal redundantly in a separate column in a native format, e.g., integer. This is usually only done for a subset of the XSD types.

Jena Denormalized Schema. The initial Jena implementation used a normalized triple store [20]. It also supported drivers for a number of different database engines, including MySQL, PostgreSQL, Oracle and Berkeley

DB (BDB). The BDB driver actually used a different, denormalized, schema and it was observed that the BDB implementation was significantly faster than the relational engines. Consequently, for the second generation Jena, we implemented a denormalized triple store for the relational engines [22]. In this schema, *short* literal values and *short* resource URIs are stored directly in the triple store table. Only when the length of a literal or URI exceeds a threshold is it stored separately in the literals or resources table. To tag its location, a literal value is stored with a prefix to indicate if it is in-line or in the literals table. Similarly, for URIs.

The denormalized approach uses more space than the normalized approach. However it saves time in that short values in statements can be retrieved without a join. And, the length threshold for short values is configurable so that applications may control the degree of denormalization, trading off space for time. Preliminary performance results for simple retrievals show the denormalized approach to be twice as fast as the normalized approach but approximately twice the space [22].

Property Tables. In the absence knowledge about the graph being stored, there are few options for storage optimization and a triple store approach is reasonable. But, as mentioned above, RDF encodes higher-level constructs such as lists. And, real-world data often have repeating patterns or structures. A simple triple store cannot leverage knowledge of patterns. For scalability and high-performance, we believe the triple store must be augmented by other storage strategies better tuned for the graph being stored or the applications using it.

To leverage patterns in an RDF graph, in Jena we have started to experiment with *property tables* to augment the triple store. A property table clusters multiple property values for the same subject in one table row. It uses one column for the subject and one column for each predicate value. In general, a property table stores all statements for its set of predicates. For example, a Person property table might store all statements about names, addresses and phones. There would be no statements with those predicates in the triple store.

A property table saves space over a triple store in that a property table does not explicitly store the predicate URIs. Instead the column (name) identifies the predicates. Also, for all statements in the row of a property table, the subject is only stored once. A property table can also save time over a triple store. When there is access locality to its statements, the entire row of a property table can be cached to avoid repeated calls to the database engine. Note that, unlike the triple store, the columns of a property table can be null. So the sparsity of property values is a factor in choosing which predicates to cluster in a property table. In general, the choice of the property tables is application-specific. Later, we describe tools to help users choose property tables.

In the current implementation, Jena supports one type of property table. It is used to store *reified* statements. Reification is an important special case because some RDF applications reify every statement. It would be inefficient to store each reified statement as four separate triples. For reification, Jena uses a property table of five columns, one for the statement identifier and one each for the subject, predicate, object and type values. Some preliminary performance results show that simple retrievals from the reification property table to be four times faster than retrieving the same statements from a triple store [22]. In some cases, the speed-up is as high as ten times faster which we believe is due to caching effects since the property table makes more effective use of the cache (e.g., the predicate URI is not repeatedly stored).

In the future, Jena will support arbitrary property tables for applications. One issue is that property tables complicate query processing, as described in the next section. However, once general property tables are supported, it should be an easy next step to directly connect Jena to arbitrary (legacy) relational databases to extract that data as RDF statements, e.g., using techniques similar to those in [6].

3 RDF Query Processing in Jena

There are two forms of Jena querying: triple match and RDQL querying. A triple match returns all statements that match a template of the form *(s,p,o)* where each element is either a constant or a don't-care. An RDQL query [16] can be expressed as a conjunction of triple match templates, where each term is either a constant, a don't-care or a variable. Variables enable joins across the triple matches. A query returns all valid bindings of its variables over statements in the graph.

The addition of property tables significantly complicates query processing. Consider iterators. Unlike a triple store table where each row corresponds to a single RDF statement, an iterator over a property table must expand a row into multiple statements, one for each predicate value. However, the major complexity occurs when a template includes an unspecified property, i.e., where the predicate is a don't-care or a variable that will be bound when the query is processed. For example, a common template is to return all known property values for a specified subject, e.g., *("Alice",-,-)*.

For a triple match where the predicate is unknown, in addition to the scanning the triple store, all property tables must be scanned and the results concatenated. In principle, it is possible to generate a single SQL union query for this match but such queries can quickly become unwieldy so this optimization has not been pursued.

In Jena, an RDQL query can be processed in a naive fashion as a nested-loops pipeline of triple match templates. The variable bindings of one triple match are used to as input to the next triple match which then binds variables for subsequent triple matches until all matches are processed. However, it would be more efficient if joins could be pushed into the database engine for evaluation. This is a difficult problem in the presence of property tables and unspecified predicates. However, in the case that all triple match templates reference only the triple store, it is possible to generate a single SQL query to resolve all match templates. Similarly, if all match templates reference only property tables, it is also possible to generate a single SQL query.

However, when the match templates span both the triple store and property tables or the predicates are unspecified (variable or don't-care), then it is difficult to generate a single SQL query. The current approach in Jena is to partition the match templates for a query into groups, where each group contains matches that can be completely evaluated by the same table and that are connected by join variables. There is one additional group containing templates that span tables. SQL queries are then generated for the single table groups and the final group is processed using the (naive) nested-loops approach. More efficient algorithms remain future work.

4 Developing Application-Specific Schema for RDF

By application-specific schema, we mean an RDF storage schema that augments the triple store with additional tables tailored for the application. Application-specific schema for storing RDF are supported in other RDF stores [4, 8, 14]. However, those systems use the RDF Schema class hierarchies to derive the application-specific database schema. In Jena, we want to support arbitrary property tables independent of the class hierarchies.

To faciliate developing an application-specific schema, we have created two tools [11]. The first is a data mining tool that discovers patterns within RDF graphs. The second is a synthetic data generator that generates realistic RDF graphs according to user-supplied specifications. These synthetic data sets can be used to populate and benchmark candidate application-specific schema. This section briefly describes these tools (see also [11]).

The RDF mining tool uses *Market-Basket Analysis* [2] techniques to discover patterns in RDF graphs and RDF query logs. If an application has an initial or sample data set (an RDF graph), our mining tool can be applied to this graph to discover property co-occurrence patterns to suggest possible properties to store together. For example, if the tool found that the predicates *Name, Address, Phone* were a frequent pattern for subjects, they would be a candidate property table.

The mining tool finds *frequent item sets* [3] in a collection of transactions of items purchased together (a market basket). Applying this to a set of RDF statements, we use the statement subject as the transaction identifier and its item set is the set of all predicates that appear with that subject in some statement. The mining algorithm then finds the longest frequently occurring item sets (for a common subject) that exceed a support threshold, i.e., a percentage of all statements.

In addition to statement patterns, the set of queries applied to a graph may also exhibit patterns and these patterns should be considered when devising a storage schema. Our mining tool can find several types of patterns in a log of RDQL queries. By varying the definition of transaction identifier and item set, our mining tool can uncover both frequently occurring queries (e.g., common join queries) or frequently queried predicates for a common subject. By using a sequence mining algorithm, it can also find commonly occurring query sequences.

Our data generator tool quickly produces large volumes of synthetic RDF data. The user-provided specifications allow a high degree of control over the characteristics of the generated data. Unlike many other synthetic data generators, it is capable of modeling relationships among class instances. Currently, the tool generates some number of class instances where each class instance comprises values for predicates of that class. Class definitions may come from an existing ontology or from a user specification. The number of occurrences of a predicate (its cardinality) in an instance may be fixed or varied. The values may be literals or resources and may randomly generated or selected from a fixed population.

Distribution functions may be specified for both the predicate cardinalities and the values themselves. The distributions supported include constant, uniform, Normal, Poisson and Zipf. The tool can also generate composite values (e.g. a name comprised of first and last names) by using bnodes and the RDF container structures *Bag, Seq* and *Alt*. Resource values may reference other generated class instances or external resources. Properties of a class may also be self-referencing to model hierarchical relationships such as taxonomies, ancestors, sub-parts. More complicated chains of references such as the Markov chains in [1] remain future work.

5 Support for Inferencing and Contexts

Inferencing. In addition to a basic graph API, Jena supports an ontology API that can generate entailments, i.e., statements that are inferred by the class and property relationships but that are not explicitly present in a base graph. For example, given the two statements *<employee,subclass,person>, <"Alice",type,employee>*, then, in a list of all persons, the RDFS entailments would include the statement *<"Alice",type,person>* even though it is not explicitly stated.

In Jena, inferencing is implemented through inference graphs. An inference graph implements the standard graph interface but also supports additional methods for consistency checking, access to deduction traces and query with posits. Inference graphs are layered on top of a base graph implementation, i.e., in-memory graphs or persistent graphs. The inference graph API can be used to access a range of different inference engines. Jena ships with a pair of generic rule engines - a forward chaining engine based on the standard RETE algorithm and backward chaining engine which uses SLD style backtracking with tabling. These engines can run in a hybrid mode (forward-backward chaining). Jena includes built-in rule sets which enable these rule engines to provide RDFS and a useful subset of OWL inferences. The architecture is extensible to allow external rule engines to be used and there has been some early experimental work on connecting to the Racer[12] description logic engine.

The current rule engines will work over persistent graphs but all inferencing is done in memory, outside the database engine. There is clearly scope for further optimization that might leverage support in the database engine for transitive closure or recursion. However, at the time of writing, there has been no specific investigation into direct deductive database implementations in Jena. Some RDF systems support inferencing directly in the storage engine by precomputing and storing some or all of the entailments [8, 13]. The hybrid rule engine approach of Jena lends itself to this - the partial closure generated by the forward rules could be persisted in the database with the backward rules being used to rewrite queries dynamically.

Contexts. Some Semantic Web applications require the ability to represent the context of a statement. For example, one might want to represent that a set of statements all refer to a particular place or occurred at a certain time or were uttered by a given speaker. We expect that a collection of statements made about a particular subject and in a particular context will be a commonly-occuring pattern. We have invented a name for that case, we call it a *Snippet* [21]. Each snippet consists of a set of statements $\{<S,P_i,O_i>\}$, $0<i<N$, made about a subject S in a context C. A snippet may be stored in a standard form as a bag of reified statements as described in [21].

Many RDF stores optimize the storage of reified statements internally by storing the four triples as a single *quad*, i.e., a statement with a fourth element. Some stores also use a quad representation for context, with a contextual node as the fourth element. While quads are efficient, they are non-standard and so may limit interoperability. For snippets, we created a compact representation with the form:

<X, type, Snippet> <X, about, S> <X, context, C> <X, P_1, O_1*> ... <X,* P_N, O_N*>*

Others have termed a similar construct a *Snapshot* [17]. Since this compact representation uses triples, it may be stored, transported and queried in the compact form using any existing RDF triple store. The application and reasoning layers need to know that a compact snippet is an abreviation but the underlying storage, transport, and querying layers do not.

Snippets neatly encapsulate a unit of knowledge about some (subject) resource within some context. Consequently, it is a natural unit of distribution and sharing. Snippets may be directly accessed by their identifier (*X* above) or associately accessed by their subject and context (*S* and *C* above). In addition, we believe a common case is that a unit of knowledge is immutable, or perhaps versioned. For this case, we have developed techniques to efficiently generate snippet identifiers using a content-based hash [21].

The use of content-based identifiers is particularly convenient for distributed environments. The identifier is short-hand for the content so it is easy to check that two snippets have the same content. Snippets can easily be cached and replicated with no risk of data inconsistency. Finally, we believe that snippets can be stored efficiently using a property table, allowing a set of N statements to reside in a single table row. Of course, this assumes the set of properties in the snippet is relatively static but this may be a common case.

6 Conclusions

An advantage of RDF is that it provides a flexible and extensible model for representing associations. Because the model does not have strong type requirements, this flexible representation raises challenges as to how to store and query RDF data efficiently. By combining the standard triple store with arbitrary property tables, we gain the advantage of being able to leverage a structured schema while keeping the flexibility of a generic triple store. We also described snippets, a new construct for the efficient expression and storage of the context associated with statements. These techniques provides efficient support for both reified and non-reified data.

References

[1] 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.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1993.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th Intl Conf on Data Engineering*, Taipei, Taiwan, 1995.

[4] 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*, May 2001.

[5] P. Biron and A. Malhotra. XML schema part 2: Datatypes. http://www.w3.org/TR/xmlschema-2, May 2001.

[6] C. Bizer. D2R map - a database to RDF mapping language. In *12th Intl World Wide Web Conf*, Budapest, May 2003.

[7] D. Brickley and R. Guha. RDF vocabulary description language 1.0: RDF schema. http://www.w3.org/TR/rdf-schema, October 2003.

[8] 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.

[9] 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, 2003.

[10] D. Connolly, F. van Harmelen, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. A. Stein. Daml+oil (march 2001) reference description. http://www.w3.org/TR/daml+oil-reference, December 2001.

[11] L. Ding, K. Wilkinson, C. Sayers, and H. Kuno. Application-specific schema design for storing large RDF datasets. In *First Intl Workshop on Practical and Scalable Semantic Systems*, Sanibel Island, Florida, October 2003.

[12] V. Haarsley and R. Moller. Description of the racer system and its applications. In *Intl Workshop on Description Logics*, Stanford, California, August 2001.

[13] S. Harris and N. Gibbins. 3store: Efficient bulf rdf storage. In *First Intl Workshop on Practical and Scalable Semantic Systems*, Sanibel Island, Florida, October 2003.

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

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

[16] A. R. 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.

[17] B. MacGregor and I.-Y. Ko. Representing contextualized data using semantic web tools. In *First Intl Workshop on Practical and Scalable Semantic Systems*, Sanibel Island, Florida, October 2003.

[18] B. McBride. Jena: Implementing the RDF model and syntax specification. In S. Decker et al., editors, *Second International Workshop on the Semantic Web*, Hong Kong, May 2001.

[19] D. McGuinness and F. van Harmelen. Owl web ontology language overview. http://www.w3.org/TR/owl-features, August 2003.

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

[21] C. Sayers and K. Wilkinson. A pragmatic approach to storing and distributing rdf context using snippets. Technical Report HPL-2003-231, Hewlett Packard Laboratories, Palo Alto, California, 2003.

[22] 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.