Jena Property Table Implementation

Kevin Wilkinson
HP Laboratories Palo Alto
HPL-2006-140
October 4, 2006*

A common approach to providing persistent storage for RDF is to store statements in a three-column table in a relational database system. This is commonly referred to as a triple store. Each table row represents one RDF statement. For RDF graphs with frequent patterns, an alternative storage scheme is a property table. A property table comprises one column containing a statement subject plus one or more columns containing property values for that subject. In this approach, a single table row may store many RDF statements. This paper describes a property table design and implementation for Jena, an RDF Semantic Web toolkit. A design goal is to make Jena property tables look like normal relational database tables. This enables relational database tools such as loaders, report writers and query optimizers to work well with Jena property tables. This paper includes a basic performance analysis comparing a triple store with property tables for dataset load time and query response time. Depending on the application and data characteristics, Jena property tables may provide a performance advantage compared to a triple store for large RDF graphs with frequent patterns. The disadvantage is some loss in flexibility.

* Internal Accession Date Only
Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA Approved for External Publication

Jena Property Table Implementation

Kevin Wilkinson

Hewlett-Packard Laboratories
Palo Alto, California USA
kevin.wilkinson@hp.com

Abstract. A common approach to providing persistent storage for RDF is to store statements in a three-column table in a relational database system. This is commonly referred to as a triple store. Each table row represents one RDF statement. For RDF graphs with frequent patterns, an alternative storage scheme is a property table. A property table comprises one column containing a statement subject plus one or more columns containing property values for that subject. In this approach, a single table row may store many RDF statements. This paper describes a property table design and implementation for Jena, an RDF Semantic Web toolkit. A design goal is to make Jena property tables look like normal relational database tables. This enables relational database tools such as loaders, report writers and query optimizers to work well with Jena property tables. This paper includes a basic performance analysis comparing a triple store with property tables for dataset load time and query response time. Depending on the application and data characteristics, Jena property tables may provide a performance advantage compared to a triple store for large RDF graphs with frequent patterns. The disadvantage is some loss in flexibility.

1.0 Introduction

A common approach to providing persistent storage for RDF is to store statements in a three-column table in a relational database system. This is typically referred to as a triple store. There are many variations on this approach, e.g., storing literals and URI's in a separate table referenced from the triple store. But, the basic approach is that each RDF statement maps onto a single row in a database table. However, many RDF datasets have a significant amount of regularity, i.e., frequently occurring patterns of statements. For example, an employee dataset might include for each employee, an employee number, a name, location, phone, etc. This paper describes a way to leverage this regularity. It describes a design and implementation for property tables in Jena, a leading RDF Semantic Web toolkit [11].

Property tables are intended to take advantage of regularity in RDF datasets by storing a number of related properties together in a table separate from the triple store. This should result in reduced storage requirements and faster access times for many types of queries. A second motivation for property tables is the desire to access legacy data that is stored in relational databases. A general purpose facility to access RDF property tables may also be used to access legacy data in non-RDF tables. Note that D2RQ [2] and SquirrelRDF [6] already provide the capability to read legacy tables as RDF. However, making the capability native to the RDF storage layer enables Jena to not only read but also to update legacy tables.

A goal of our approach is to enable Jena property tables to look like conventional application database tables. In this way, existing relational database tools and services can operate over the Jena tables, e.g., data mining tools, report writers. Of particular significance is that the database engine itself will be able to gather meaningful statistics for property tables which will enable more effective database query optimization. Also, the ability to use a native database loader for property tables should result in drastic reductions in load time.

The existing persistent storage layer for Jena2 has limited support for property tables. In particular, reified statements are stored in a property table, separate from the triple table that stores asserted statements [1]. This paper describes extending that work to support general, user-defined property tables in Jena.

For queries, Jena property tables offer two significant advantages over a triple store. First, the database query optimizer can sometimes generate better plans over property tables than over a triple store. For example, consider a query to find all 50 year old people with an IQ of 150, i.e., *?s ex:hasAge 50 and ?s ex: hasIQ 150.* The goal of the query optimizer is to generate an efficient query plan which here means choosing the most selective predicate (age or IQ) and evaluating that predicate first, i.e., first find the 50 year olds and then the smart people or vice versa. Assuming accurate database statistics, the query optimizer can determine

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

from the property tables that there are many more 50 year old instances than there are smart instances. However, for the triple store, all property values are stored in the same column so the optimizer cannot readily distinguish IQs from age values and so cannot choose the best order of evaluation. A second advantage for property tables is that joins can be eliminated. In the above example, if we assume that age and IQ are single-valued properties, then we can store both properties as columns in the same property table and process the query as a selection over that table. However, that same query over a triple store requires a join.

The next section describes the types of property tables supported in Jena. Section 3 describes a benchmark designed to evaluate the benefit of property tables. Section 4 presents the benchmark results over both a triple store and property tables. The final section describes future work.

2.0 Jena Property Tables

A Jena property table is defined as a relational database table in which each table row corresponds to one or more RDF statements and the URIs of statement properties are not stored in the table. Instead, the property URIs are stored with the table metadata. A Jena property table has a single column to store the subject of the statement. The remaining columns store property values for statements, i.e., the object of the statement (see Figure 1).

In our approach, property tables augment but do not replace the triple store. The triple store is used for statements containing a predicate that has no property table. In addition, all object values for a given property are stored in a single table, either a property table or a triple store but never both. An exception to this rule is the *rdf:type* property as discussed later.

Some other systems that support property tables derive the table definitions automatically from the ontology of the dataset (e.g., Sesame [7]). However, in our approach, property tables are defined by the application, independently of any ontology. The definitions must be provided when the graph (model) is initially created. In principle, property tables could be created and deleted dynamically but this would require the system to reorganize the database, moving statements between the triple store and property tables. So, this is left as future work.

2.1 Types of Property Tables.

Jena supports three kinds of property tables as well as a triple store (Figure 1). A single-valued property table stores values for one or more properties that have a maximum cardinality of one. The subject column serves as the table key (the unique identifier for the row). Each property column may store an object value or be null. Thus, each row represents as many RDF statements as it has non-null property values.

A multi-valued property table is used to store a single property that has a maximum cardinality greater than one (or that is unknown). The subject and object value serve as the table key, a compound key in this

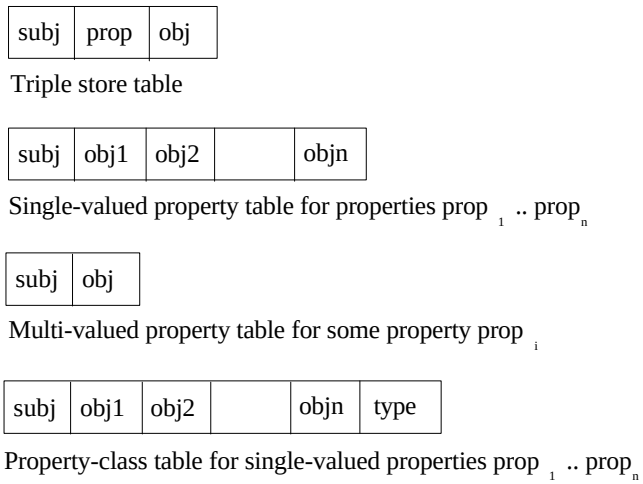

FIGURE 1. Types of Jena tables

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

case. Each row in a multi-valued property table represents a single RDF statement. The property column value may not be null. Note that, in principle, the distinction between single-valued and multi-valued tables should not be needed since various strategies could be used to store multiple values for a property in a multi-column property table. However, these strategies greatly complicate query processing so they were not seriously considered for this first implementation. Consequently, an application must know in advance which properties are single or multi-valued. This is a loss of flexibility compared to a triple store. In the absence of any knowledge, many multi-valued tables may still be used. For some applications, this can still provide performance advantages over a single triple store.

In some cases, it is useful to store all members of a class together in a single table (e.g., to easily enumerate all instances). In effect, this amounts to storing the value of *rdf:type* in a property table. Each class is stored in its own table, spreading the rdf:type property across many tables. So, *rdf:type* is an exception to the rule that all values for a property are stored in one table. This kind of table is referred to as a property-class table. Besides the *rdf:type* value, a property-class table may also store a number of single-valued properties. Any single-valued property may be stored in a class table, i.e., not just those that declare the class in their domain. A current limitation is that a property-class table only stores instances of a single class. However, this could be relaxed in the future to enable a single table to store sibling subclasses of a common class. As an example of a property-class table, consider reified statements which contain the properties *rdf:subject*, *rdf:predicate*, *rdf:object* and *rdf:type*, where the type value is *rdf:Statement*. In fact, this is how Jena stores reifications.

2.2 Column Encoding

There are many ways to encode RDF terms (subjects, predicates and objects) in a table. A commonly used approach is to encode each term as a number and to store that number in the triple store table. A separate *symbols* table is used to map the number to the value for the term [3]. This is an elegant and simple approach and saves space since each term value is stored only once in the symbols table at the expense of an additional join between the symbols table and the triple store. An alternative approach, used in Jena, is to store the term values directly in the statement table. However, some additional encoding is needed to distinguish, for example, URIs from Bnodes from literal values since are all stored as strings. This *denormalized* approach has the benefit of reducing the number of joins and indexes required but it complicates query processing.

Recall that a design goal is to enable property tables to look like normal database tables. Thus, it should be possible to store property table columns as native database datatypes, e.g., integer, float, string, date-time, etc. with no additional encoding. This is also needed to enable access to legacy database tables where Jena has no control over the choice of the column datatype. However, it is also useful to store property tables using the Jena encoding in order to enable joins between property tables and the triple store.

Consequently, support is needed for both encoded column values and native column values. The Jena triple store uses an encoded representation. A Jena property table column may be either a native database type or use Jena encoding. However, all values in a particular column must be the same type. The use of a native database type for a column reduces flexibility but enables performance optimizations (e.g., range queries).

2.3 Property Table Definition

Property tables for a graph (model) must be specified at graph creation time. This is done by providing a meta-graph to the graph constructor. The meta-graph contains metadata in the form of RDF statements defining property tables (e.g., type of property table, name) and property table columns (e.g., name, property, encoding, indexes, etc.). This metadata is stored in a system graph separately from the user graph. Note that graphs may share property tables. For large graphs where the frequently occurring patterns are unknown, tools may be used to discover patterns and suggest candidate tables [9].

2.4 Graphs Operations on Property Tables

Property tables present issues for the basic graph operations of add, delete, find, query. This is because the graph operations operate over statements while property tables operate over rows, i.e., sets of statements. At a high-level, each property table is modeled as a disjoint subgraph of the parent graph, i.e., it contains a set of

statements not stored in any other property table or triple store. So, each add, delete or find operation can be processed by applying the operation to each subgraph (property table or triple store) and concatenating the results. Below, the processing of each operation is briefly described.

Add statement

Assuming that the statement to be added is not a duplicate of an existing stored statement, an add operation on a multi-valued property table creates a new row in the table. An add on a single-valued or property-class table creates a new row if the statement subject does not exist in the table. If the subject already exists, the row for that subject is updated with the property value.

An optimization for single-valued property tables is possible when inserting batches of statements. Assume the statements are sorted so that all statements for a common subject are grouped. Then, rather than individual database operations for each statement (i.e., insert row, update, update, update), Jena maintains state between each add operation and performs a single insert statement for the entire row. This batch add operation can significantly reduce load times for single-valued and property-class tables.

Delete statement

Delete processing is similar to add processing. On a multi-valued table, a delete removes a row. On a single-valued or property-class table, delete changes a column value to null. If all property columns are null as a result, the row could be removed (garbage-collected). As with add, an optimization is possible for batches of statements.

Find statement

The find operation takes a triple pattern and returns all statements that match the pattern. A triple pattern has the form *[s,p,o]* where each term is either an RDF resource or literal or a don't-care. A triple pattern can be easily processed over a triple store with a single SQL statement that matches each term in the pattern to the corresponding table column. There are eight possible pattern types (combinations of terms and don't-care) so Jena predefines and caches SQL statements for all possible triple patterns over a triple store.

For a property table, the number of possible triple patterns (and associated SQL statements) is `4*(p+1)` where `p` is this number of property columns in the table. So, Jena does not predefine SQL statements for all possible triple patterns over all the property tables. For these patterns, it generates SQL dynamically and caches the SQL statement for reuse. Note that if the predicate term in a triple pattern is a don't-care, then all property tables and the triple store must be searched for matching statements. The results are concatenated.

Query processing

An in-depth discussion of query processing is beyond the scope of this paper. However, it is worth pointing out an advantage and disadvantage of query processing over property tables. A simple query is just a conjunction of triple patterns in which variables may appear as terms. The goal is to convert this query into a single SQL statement. As before, if a predicate term in some triple pattern in the query is a don't-care or a variable, then that pattern could match statements in any property table as well as the triple store. A single SQL statement for that query would involve a large SQL union. Consequently, for this case the query processor avoids the union and queries each table separately.

An important optimization for property tables is that joins can be eliminated. Consider a query consisting of two triple patterns: *[?var,p1,-] and [?var,p2,-]*. Processing this query over a triple store requires a join. But if properties *p1* and *p2* are both stored in the same property table, then the join can be replaced by a simple select operation since it is known that a subject may only have one value for its *p1* and *p2* properties.

2.5 Legacy Database Tables

Given the capability to read and write Jena property tables, it might seem a small stretch to extend this capability to access *legacy* database tables, i.e., non-Jena tables created and managed by other applications. D2RQ [2] and SquirrelRDF [6] already provide this capability for read-only access. Our goal is to support modification of legacy tables. This section presents an overview of our mapping between legacy tables and RDF. While it is not as flexible as D2RQ, it is a starting point and should support access to an interesting subset of legacy tables. For more details, see [4].

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

Given the flexibility of the relational model, there are countless different data modeling strategies. But, at a high level, we can categorize a table as one of three types: object table[1], relationship table or mapping table. A mapping table is used to encode or transform data from one representation to another, e.g., from an integer code to a character string. For now, mapping tables are ignored.

We can assume that an object table will have a single key to identify the object instance and a relationship table will have multiple keys to identify all objects that participate in the relationship. If we assume that each key can be stored in a single column (no compound keys), then it is relatively straightforward to access an object table as a property table. See Figure 2, where the property values are stored as columns *vi*. Note that the key, by definition, uniquely identifies a row. So, duplicates are not an issue. Access to tables with compound keys is discussed next.

Virtual bnodes for compound keys

The key of a relationship table (e.g., key1..keyn in Figure 2), is a compound key that represents a relationship among two or more objects. RDF does not directly support compound keys, i.e., each subject of an RDF statement is a single URI. So, given a compound key, we infer the existence of an anonymous object that represents an instance of a relationship among the objects in the compound key. To do this, we create a new type of anonymous object, a *virtual bnode.* It is a surrogate for a compound key and identifies a row in a table. It is virtual because it is not stored in a table but is constructed on demand when a row is extracted. More details about virtual bnodes are available in [4]. This approach can also be used for object tables with compound keys.

3.0 Benchmark

To study the performance tradeoffs of property tables, a synthetic dataset was created and queries were run against that dataset under two different storage schemas. The first storage schema is the (default) Jena triple store. The second schema augments the triple store with the set of property tables described below. This section describes the dataset, the schema and the queries. A more detailed description of the dataset and queries is presented in [5]. We chose to develop our own dataset and benchmark that was tailored to this implementation in order to better understand the cost and benefit of different aspects of property tables. In the future, we may run a general-purpose RDF benchmark (e.g., [10]) to better evaluate the overall impact of property tables on applications.

3.1 Dataset.

The dataset consists of two object classes and a number of properties on the instances. Most properties are single-valued, datatype properties (literal-valued). The main object class, S10K, has 10,000 instances. A second object class, S100, has 100 instances but no properties. It is only used as the range of an object property for S10K.

key	v1	v2		vn

Object table becomes a single or multi-valued property table

key1	key2		keyn	v1	v2		vk

Relationship table becomes single or multi-valued property table

FIGURE 2. Tables serve different purposes in a relational database

[1] Unfortunately, the word object is overloaded. It is used here in the sense of object-oriented modeling and should not be confused with RDF statement objects.

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

There are three groups of datatype properties. A set of integer-valued properties, two short (5 character) string-valued properties and a long (50 character) string-valued property. The properties differ in the cardinality of the range. Some properties map to unique values, others have a fixed number of values ranging from 2 values (for a binary valued property, like gender), 10 values, 100 and 1000. The actual integer and string values are unimportant but they are stored as typed literals. In this experiment, the object values are chosen uniformly random, typically with replacement (non-unique values), but sometimes without replacement (unique values). Also, there are no missing values, i.e., each instance has at least one value for each property.

The datatype properties for class S10K are listed in Table 1. The datatype properties should be self-explanatory. The object properties for S10K are listed in Table 2. The *Uniq* property links an instance of S10K to a randomly chosen, unique instance of S10K. The Nonuniq also links to an S10K instance but that instance may be linked to many times. The S100C5 property is a multi-valued property that links an S10K instance to five randomly chosen instances of class S100. The tree properties link the S10K instances into a tree structure with a fan-out of five. The instances are enumerated either in a depth-first or breadth-first order.

TABLE 1. Datatype Properties for class S10K

Property	Domain	Range	Description
intR2	S10K	int 0..1	single-valued, int, cardinality 2
intR10	S10K	int 0..9	single-valued, int, cardinality 10
intR100	S10K	int 0..99	single-valued, int, cardinality 100
intR1K	S10K	int 0..9999	single-valued, int, cardinality 1000
str5R10	S10K	5 char string	single-valued, 5 char string, cardinality 10 strings
str5R100	S10K	5 char string	single-valued, 5 char string, cardinality 100 strings
str50	S10K	50 char string	single-valued, 50 char random string

TABLE 2. Object Properties for class S10K

Property	Domain	Range	Description
S10Kuniq	S10K	S10K	single-valued, random unique instance of S10K
S10Knniq	S10K	S10K	single-valued, random instance of S10K
S100C5	S10K	S100	multi-valued, 5 random instances of S100
S10KtreeC5DF	S10K	S10K	tree of S10K instances, fan-out 5, depth-first enum.
S10KtreeC5BF	S10K	S10K	tree of S10K instances, fan-out 5, breadth-first enum.

3.2 Schema

A set of property tables was created to store the S10K properties (Table 3). For the datatype properties, the table columns are created as either SQL integer or character columns. For object properties, the table columns are created as character columns using the same URI encoding as in the Jena triple store. This simplifies joins between the triple store and a property table.

Each property table includes a primary key column that contains the URI of an S10K instance (i.e., the subject of triples stored in the rows of the property table). For the single-valued property table, indexes were created for the intR1K, S10Kuniq and S10Knniq properties. There seemed little value in creating an index on low selectivity properties, intR2, intR10, intR100. For each multi-valued table, an index was created on the property value column in addition to an index on the subject column. The triple store includes an index on the object value column and a multi-column index on the subject and predicate columns.

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

TABLE 3. Property Tables for class S10K

Table Name	Table Description
S10K_SV	Stores all single-valued properties of class S10K (9 properties total)
S10K_S100_MV	Stores the S100C5 property
S10K_DF_MV	Stores the S10KtreeC5DF property (depth-first tree)
S10K_BF_MV	Stores the S10KtreeC5BF property (breadth-first tree)

3.3 Data Loading

In order to determine the impact of property tables on the time to load the dataset, a number of different datasets were loaded. These are listed in Table 4. Unless otherwise stated, all datasets are in RDF/XML format and duplicate checking is disabled for loading since the dataset has no duplicate triples. The first configuration is the complete dataset. Note that in RDF/XML format, the property values for an instance are often listed consecutively. For property tables, this means that successive statement add operations can be optimized as a single table row insert.

To measure the effect of this optimization, the second dataset contains only the single-valued properties while the third dataset contains only the multi-valued properties. A final dataset includes all single-valued property values, as in the S10K_SV dataset, but this time encoded in N-Triple format and the statements are randomized. In this way, property values for the same instance are scattered throughout the load file.

TABLE 4. Configurations for Dataset Load

Dataset	Description
S10K_All	Contains triples for all S10K property values
S10K_SV	Contains triples for S10K single-valued properties
S10K_MV	Contains triples for S10K multi-valued properties
S10K_SV_Rand (NT)	Contains triples for S10K single-valued properties but in random order (Ntriples)

3.4 Queries

The queries over the dataset are summarized in Table 5. The SPARQL syntax for each query is included in the Appendix[2].

TABLE 5. Queries on class S10K

Query	Description
Query1	Retrieve many properties for all S10K instances with a given intR1K value (indexed)
Query2	Retrieve many properties for all S10K instances with a given intR100 value (unindexed)
Query5	A path expression using the S10Kuniq and S10Knniq properties
Query6	A cross-product that returns 10,000 results
Query8	A simple selection on the intR2 property and the intR1K property
Query9	Same as Query8 but the order of the selection predicates is reversed
Query10	Retrieve the S100C5 values for one S10K instance (multi-valued property retrieval)
Query11	Retrieve all property values for one S10K instance (query with an unspecified predicate)

[2] Note that the queries were actually run using RDQL. The integration of property tables and SPARQL is not yet complete but is not expected to significantly change the results here.

Query1 and Query2 are similar. They retrieve a large number of single-valued properties for selected S10K instances. However, Query1 selects instances using the intR1K property which has approximately 10 matches. Query2 selects the same properties but using the intR100 property which has approximately 100 matches. So the result sizes differ by an order of magnitude. Note that for the property table schema, Query1 has an index available for intR1K while no index is available for intR100 in Query2. Also, for the triple-store schema, both queries require a 7-way self-join. But, for the property table schema, the query can be optimized to a simple selection (no join required).

Query6 is a cross-product that retrieves a large number of matches (10,000). The response time should be dominated by data transfer costs. Query8 and Query9 are identical except the order of the selection conditions is reversed. These queries are intended to show the benefit of statistics in query optimization. The intR1K predicate is much more selective than the intR2 predicate (one returns 100 instances, the other returns 5000 instances). A good query optimizer should detect this and process the intR1K predicate first. However, for the triple store schema, such detailed statistics are typically not stored[3]. So the optimizer has no way of knowing which predicate is more selective and will typically process the predicates in the order given. Consequently, for the triple store we should see large differences in response time for these two queries while for the property tables we should see little difference.

Query10 retrieves the value of a multi-valued property for an S10K instance. We should not expect a huge difference between the two storage schemes for this query. However, Query11 retrieves all properties for the same S10K instance. The query contains an unspecified predicate. It can be processed as a simple select on the triple store. However, for the property tables, several queries are required, one for each table. So, the triple store should out-perform the property table for this query.

4.0 Performance Results

This section presents the response time measurements for the load and query operations discussed above. The times were measured at the application level (Jena API). Each operation was executed multiple times, outliers ignored and the results averaged. The table indexes were created prior to loading a dataset. After each operation, the database buffer cache was flushed. Table statistics were computed over all tables before running the queries so that the optimizer had accurate information. The measurements were performed using Jena 2.4 with a widely-used relational database engine, running on 2.8 GHz Pentium Xeon Windows XP system with SCSI disks, 1.5 GB RAM. In the results below, the absolute response times should be viewed as less interesting than the relative times.

4.1 Dataset Loading

The times for loading the various datasets under the triple store and property table schemas are listed in Table 6. As should be expected, the load time for the complete dataset (S10K_All) is approximately equal to the sum of the load times for the single valued and multi-valued properties (S10K_SV and S10K_MV) for each schema. Recall that there are nine single-valued properties for S10K. One might expect the load time for the single-valued property table to be nearly an order of magnitude lower than the load time for the triple store since the property table inserts 10,000 rows compared to 90,000 rows for the triple store. But this is not the case. The property table load time is roughly one-quarter the triple store load time (14.4 vs. 60.9). However, considering that the time spent in the Jena storage subsystem is only a fraction of the total path length for adding a triple, this is a significant improvement.

Note that there is not much difference between the two schemas in the load times for the multi-valued properties. Both schemas store the same number of rows. Still the property table has a slight advantage. This may be because the property table only stores two columns and so transfers less data.

[3] Database statistics on the triple store table would indicate the number of triples with a particular object value, e.g., 1 or 9, or the number of triples with a particular predicate value, e.g., property intR2 or int1K. But, what is needed are multi-variate statistics, e.g., the number of triples with property intR1K and the value 9. Only sophisticated query optimizers maintain such statistics and only upon request because they consume much space.

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

The results for loading the randomized triples are very interesting. In this case we see that the triple store performs better than the property table. This is likely due to thrashing. The triple store can simply append rows. The property table must update table rows out of order and so performs more random disk I/O.

Recall that duplicate checking was disabled for these loads. An experiment was performed to determine the impact of duplicate checking. This test was run using the single-valued dataset and the results are labeled *S10K_SV (Dup Chk)* in Table 6. Here, response times for both schemas increase by a similar percentage.

As a final comment, we ran several experiments to determine the impact of indexes on load times. Specifically, if a property table has several indexes, each row inserted must update those indexes. So, a possible performance optimization is to drop indexes before loading a property table and recreate them afterward. In fact, we found no significant benefit for doing this. Response times for loading the property table were similar regardless of whether or not indexes existed.

TABLE 6. Dataset Load Times (sec)

Dataset	Triple Store	Property Table
S10K_All	114.9	58.4
S10K_SV	60.9	14.4
S10K_MV	47.9	42.5
S10K_SV_Rand (NT)	66.7	108.7
S10K_SV (Dup Chk)	152.6	23.5

4.2 Query Response Times

The query response times for both schemas are shown in Table 7. The first two queries are similar. They retrieve a number of single-valued properties for a subset of the S10K instances. Query1 selects the instances with an equality condition on the intR1K property. Query2 uses an equality condition on the intR100 property. For query1, we see the property table performs nearly five times better than the triple store. This is somewhat surprising since an index was available for the triple store. However, the triple store must perform a 7-way join to process this query. The property table can process this query as a simple row selection.

For query2, we expected the property table to perform worse since it has no index on the intR100 property and must perform a full table scan to process the query. In contrast, the triple store does have an index. However the performance difference grows and is now a factor of seven. In fact, the triple store index does little good here since it is not very selective. A match on the S100 column returns 100 rows and for each such row there is a 7-way join.

Query5 implements a path expression as a 3-way join. We expected the triple store and property table performance to be similar since they both perform the same number of joins. In fact, the property table is a factor of two better than the triple store. Query6 implements a large cross product. Once again, we expect the property table and triple store to perform similarly. In fact, that is the case here. However, response time is surely dominated by data transfer time because so many tuples are returned. We note that the property table is slightly faster, perhaps because it moves less data (the property URIs are not stored in the property table and so are not transferred from the database engine).

Queries 8 and 9 demonstrate a key advantage of property tables, i.e., their ability to leverage the query optimizer. The queries are identical except for the order of the equality conditions. One equality is on the intR2 property. This is not very selective and will match one half of the S10K instances. The second equality is on the intR1K property. This is more selective and will match one tenth of the S10K instances. Clearly, in doing a join the best strategy is to evaluate the most selective condition first.

For the property table schema, the optimizer has good statistics on the intR2 and intR1K columns and so can determine the most selective condition. However, for the triple store, statistics on the object column are useless. What is needed are multi-variate statistics on the predicate and object columns which this database engine does not use (as least with the default statistics). We see that the property table response times are roughly equal for queries 8 and 9. However, the triple store times for these queries differ by a factor of two. This indicates that the optimizer chose different execution plans for what are semantically identical queries.

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

The purpose of query10 is to detect any performance difference between a property table and a triple store when retrieving a multi-valued property. The response times are essentially the same as expected. Query 11 illustrates a fundamental weakness of property tables compared to a triple store. It retrieves all property values for an instance and so the query contains an unspecified predicate. This query can be processed as a selection over the triple store. However, for the property tables, each table must be queried separately.

TABLE 7. Query Response Times (msec)

Query	Triple Store	Property Table
Query1	1136	239
Query2	2850	406
Query5	1331	450
Query6	93907	84029
Query8	3465	246
Query9	7156	234
Query10	218	228
Query11	230	309

5.0 Summary and Next Steps

Property tables augment the Jena triple store by providing efficient storage for frequently occurring patterns of statements. They are space efficient in that the predicate URI of a statement is not stored. They are time efficient in that a single database operation can store or retrieve a set of RDF statements, encoded as a single table row. Property tables are less flexible than a triple store and the basic graph operations over property tables are more complicated. However, the core functionality enables Jena to access and update legacy relational database tables and so helps to bridge the gap between structured and semi-structured information.

This paper describes the Jena implementation of property tables and presents a preliminary analysis using a small query set. The results look promising in that the performance of property tables is generally as good or better than a triple store. And, since property tables make good use of the query optimizer, their performance is more robust with respect to the way a query is expressed. And some property tables enable some optimizations that offer big performance improvements for loading and eliminating joins in queries. However, the benefits come at the cost of reduced flexibility.

The next steps are to finish the implementation, including access to legacy database tables. A more comprehensive set of queries is also needed. Along these lines, queries over the schema itself should be added (see [8] for examples) as these are common in RDF applications. Also of particular interest are queries that span property tables and the triple store, queries over multi-valued property tables and queries in which the predicate is unspecified (requiring a scan of all tables).

Another interesting area to study is the effect of sparsity on performance. For example, in the dataset used here the property tables are dense, i.e., every instance has a value for every column. In real-word datasets, the data may not be so regular and so the properties tables will be less dense. As the density decreases, it will be interesting to see if property tables retain any performance advantage over a triple store. Queries over the tree structures should be studied as examples of queries over taxonomies. And, more complicated graph navigational queries should be considered in addition to the simple query, Query6. Finally, note that property tables perform poorly for queries with an unknown predicate and this effect should be studied.

Acknowledgements. Katie Portwin provided helpful comments on an earlier draft of this paper.

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

References

1. Kevin Wilkinson, Craig Sayers, Harumi Kuno, Dave Reynolds. Efficient Storage and Retrieval in Jena2, VLDB 2003 Workshop on Semantic Web and Databases, Berlin, Germany, 2003.
2. Chris Bizer, Andy Seaborne, D2RQ -Treating Non-RDF Databases as Virtual RDF Graphs, Proceedings of the Third International Semantic Web Conference (ISWC2004), Hiroshima, Japan, November 2004.
3. Steve Harris, SPARQL Query Processing with Conventional Relational Database Systems, International Workshop on Scalable Semantic Web Knowledge Base Systems, New York City, November, 2005
4. Kevin Wilkinson, Jena Property Table Design, Proceedings of the Jena Users Conference, Bristol, England, May, 2006.
5. Kevin Wilkinson, Luping Ding, Benchmark Tools for RDF Engines, Hewlett-Packard Laboratories Technical Report, to appear, 2006.
6. Damian Steer, SquirrelRDF, http://jena.sourceforge.net/SquirrelRDF/.
7. Jeen Broekstra, Arjohn Kampman, Frank van Harmelen, Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, Proceedings of the First International Semantic Web Conference, 2002.
8. Yannis Theoharis, Vassilis Chrisophides, Grigoris Karvounarakis, Benchmarking Database Representations of RDF/S Stores, Fourth International Semantic Web Conference, November, 2005.
9. Luping Ding, Kevin Wilkinson, Craig Sayers, Harumi Kuno, Application-Specific Schema Design for Storing Large RDF Datasets, First International Workshop on Practical and Scalable Semantic Systems, October 2003, co-located with ISWC 2003.
10. Y. Guo, Z. Pan, J. Heflin. An Evaluation of Knowledge base Systems for Large OWL Datasets, Proceedings of the Third International Semantic Web Conference, (ISWC2004), Hiroshima, Japan, November 2004.
11. Jena. http://jena.sourceforge.net/

Appendix - Queries

Query 1

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?s1 ?vi2 ?vi10 ?vi100 ?v5s100 ?v5s10 ?v50s
WHERE
{ ?s1  s:intR2     ?vi2 ;
       s:intR10    ?vi10 ;
       s:intR100   ?vi100 ;
       s:str5R10   ?v5s10 ;
       s:str5R100  ?v5s100 ;
       s:str50     ?v50s ;
       s:intR1K    "99"^^xsd:int .
}
```

Query2

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?s1 ?vi2 ?vi10 ?vi100 ?v5s100 ?v5s10 ?v50s
WHERE
{ ?s1  s:intR2     ?vi2 ;
       s:intR10    ?vi10 ;
       s:intR100   ?vi100 ;
       s:str5R10   ?v5s10 ;
       s:str5R100  ?v5s100 ;
       s:str50     ?v50s ;
       s:intR100   "99"^^xsd:int .
}
```

Published in and presented at the Second International Workshop on Scalable Semantic Web Knowledge Base Systems, 5 November 2006, Athens, Georgia, USA

Query5

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?s1 ?s2 ?s3 ?vs50
WHERE
{ ?s1  s:intR1K    "99"^^xsd:int ;
       s:S10Kuniq  ?s2 .
  ?s2  s:S10Knniq  ?s3 .
  ?s3  s:str50     ?v50s .
}
```

Query6

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?s1 ?s2 ?s1s50 ?s2s50
WHERE
{ ?s1  s:intR100  ?v1 .
  ?s2  s:intR100  ?v1 .
  ?s1  s:str50    ?s1s50 .
  ?s2  s:str50    ?s2s50 .
  FILTER ( ?v1 = "9"^^<http://www.w3.org/2001/XMLSchema#int> )
}
```

Query8

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?s1
WHERE
{ ?s1 s:intR1K "10"^^xsd:int ; s:intR2 "1"^^xsd:int . }
```

Query9

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?s1
WHERE
{ ?s1 s:intR2 "1"^^xsd:int ;  s:intR1K "10"^^xsd:int . }
```

Query10

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?o1
WHERE
{ <http://invent.hpl.hp.com/rdfgen/S10K/10>  s:S100C5  ?o1 . }
```

Query11

```
PREFIX  xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX  s: <http://invent.hpl.hp.com/rdfgen/>

SELECT  ?p1, ?o1
WHERE
{ <http://invent.hpl.hp.com/rdfgen/S10K/10>  ?p1   ?o1 . }
```