Jena Property Table Design

Kevin Wilkinson

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

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 lot 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.

Property tables are intended to take advantage of regularity in RDF datasets by storing a number of related properties together in a separate table. 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] already provides this capability. However, making the capability native to the RDF storage layer enables the system to also update those 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., report writers. In particular, the database engine itself should be able to gather meaningful statistics for the table content which will enable more effective database query optimization. The ability to leverage the 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]. However, general, user-defined property tables are not supported. This paper presents a design for user-defined property tables for Jena2 and discusses some implementation issues. The implementation is underway but not complete.

2.0 Property Table Definition

In this paper, 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 property URIs for the statements are not stored in the table. 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 statement object). The property URI for a column is stored with the table metadata.

In our approach, property tables augment but do not replace the triple store. The triple store is used for statements that have no property table. In addition, all instances 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.

In the current approach, property tables are defined by the user. The definitions must be provided when the graph is 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 a property table. So, this is left as future work.

Table types. Jena supports three kinds of property tables as illustrated in Figure 1. A sin-

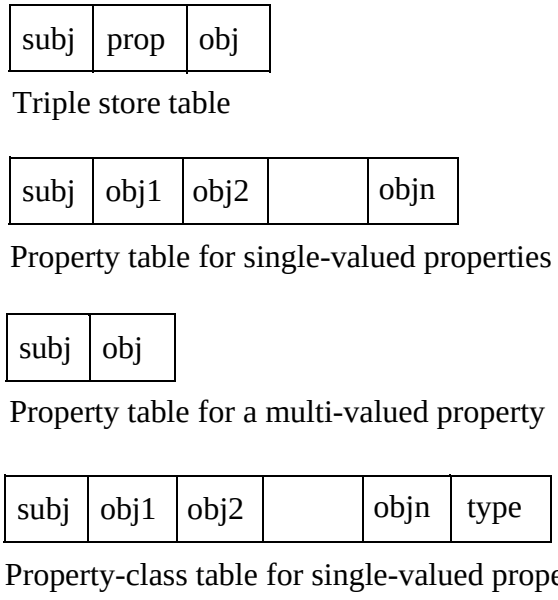

FIGURE 1. Types of Jena tables

gle-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 case. Each row in a multi-valued property table represents a single RDF statement. The property column value may not be null.

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, this is an exception to the rule that a property is stored in one table. This kind of table is referred to as a property-class table. Beside the *rdf:type* value, a property-class table may also store a number of single-valued properties. A current limitation is that a property-class table only stores instances of a single class. However, this can probably be relaxed in the future which would enable a single table to store sibling subclasses of a common class.

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 the 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. An alternative approach 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 they may all be representing as strings. This 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. This means supporting columns whose values are encoded as native database datatypes, e.g., integer, float, string, dateTime, 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.

Consequently, support is required for both encoded column values and native column values. The triple store uses an encoded representation. However, property tables can be defined using a mixture of representations (as described below). In particular, the datatype of a property column must be specified when the table is created. A property table column may store encoded values or native database values. The use of a native database type for a column reduces flexibility but enables certain performance optimizations (e.g., range queries).

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, etc.). This metadata is stored in a system graph separately from the user graph. Note that a graph may reuse/share existing property tables.

3.0 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 dis-

joint 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 described in more detail.

Add statement. For the sake of discussion, we assume the statement is not a duplicate of an existing 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.

A useful 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. Consequently, it is not feasible for Jena to predefine SQL statements for all possible triple patterns over all the property tables. So, for these patterns, it generates SQL dynamically and caches the SQL statement for reuse.

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. However, 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 falls back to the default nested-loops query processing strategy.

An advantage of property tables is that joins can be eliminated. Consider this 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 a single-valued or property-class 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.

4.0 Legacy Database Tables

Once Jena has the ability to create and access 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 already provides this capability for read-only access. Our goal is to support modification of legacy tables. This section describes 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.

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, 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.

An object table contains data for object instances[1]. Its columns contain values describing an object instance. A relationship table describes a relationship among a number of objects. It has columns to identify the objects in the relationship and, possibly, additional columns describing the relationship instance. For example, a parts-supplier database might have one object table describing stores, a second object table describing parts and an inventory relationship table containing a store id, a part id and a quantity.

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 a class 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 below.

Note that each key or key component is considered a resource identifier, regardless of its encoding. To support this, the column description of a key (in the table metadata) may specify a URI prefix to prepend to the key value. In this way, an integer-valued key component, for example, can be easily converted to a URI.

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.

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

key	v1	v2		vn

Class 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

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.

Additionally, we define *compound key properties*. These are single-valued properties from the virtual bnode to the components of the key. We also define inverse compound key properties. These are multi-valued properties that map from a key component to the anonymous object for the row. Compound key properties are object properties (in the OWL sense). The key properties are useful in querying since they link a relationship instance to the objects that are related.

Figure 3 shows an example table for the inventory relationship mentioned previously. It also shows the RDF graph that is generated when the table rows are retrieved. Each instance of the relationship, i.e., each row of the table, is identified by a virtual bnode. The relationship property is quantity (abbreviated qty). The compound key properties are storeid and partid. The inverse compound key properties are not shown and are not stored. They may be used as a convenience in querying.

Note that in Figure 3, the RDF graph includes objects that are not stored in the table. First, an *rdf:type* property appears in the graph. This is because we assume this inventory table is a property-class table for the *ex:inventory* class. It would not be in the graph if the relationship table had no associated class[1]. Second, the bnodes themselves (e.g., _i1) are not stored in the table. This raises the questions of how the virtual bnode identifiers are generated and how they can be used to reference rows of the table.

Accessing compound keys. When retrieving properties from a relationship table, the Jena driver will generate a virtual bnode with an identifier that can be used to uniquely identify the row. This may be done one of two ways and the choice depends on the capabilities of the underlying database engine. First, the components of a compound key may be encoded as strings (if not already) and concatenated using some suitable field separator. The resulting string can then be used within a virtual bnode identifier. Note that this approach may fail if the compound key is too long to be string-encoded. Alternatively, if the database

1. A legacy table may be viewed as a property-class table where the type is implied but not stored.

TABLE 1. Inventory relationship table, key [storeid, partid]

storeid	partid	quantity
5	widget	100
5	button	30
6	widget	250

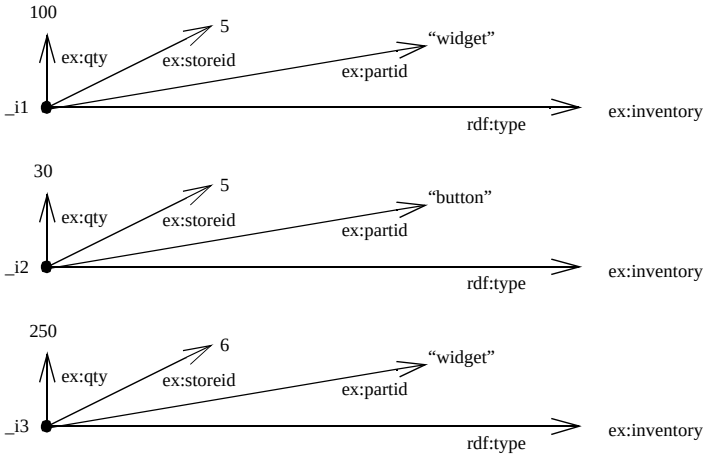

FIGURE 3. Mapping Relationship Table to RDF

engine supports tuple (or row) identifiers, that identifier can be encoded as a string and used with a virtual bnode identifier.

Adding a row to a relationship table, i.e., adding a new relationship instance, presents a challenge. A row in a relationship table cannot be created through a series Jena *add statement* operations. First, there is the question of which virtual bnode identifier to use for the relationship instance, i.e., for the statement subject. Second, there is the possibility that the relationship table may have non-null constraints that prohibit a null value in any key component. Consequently, for relationship tables the batch add operation mentioned previously must be used to ensure that all key components for the row are defined prior to row creation. A normal bnode can be used in the batch to identify a relationship instance. It will be ignored but it should be identical for all statements in the batch for that instance.

Note that updates to a relationship table are restricted to non-key properties. The compound key properties cannot be modified (i.e., deleted) once a relationship instance is created. For example, in Figure 3, only the *quantity* property could be modified. It may be possible to relax this limitation in the future.

Referencing a relationship instance. Given a relationship instance, it should be possible to directly reference that instance from other RDF statements. This is done simply by including the virtual bnode for the instance in some other statement. However, another approach is to reference a relationship instance indirectly, by querying on the compound key properties and inverse compound key properties. This ensures referential integrity.

The issue is that legacy tables are managed by non-Jena applications. Storing a direct reference to a relationship table row carries the risk that the non-Jena application may modify the row or delete it. This would leave an invalid reference in any Jena statement that referenced the row. As an example of an indirect reference, suppose we want to reference relationship instance *_i2* in Figure 3. Rather than use *_i2* directly as the object in an RDF statement, instead create a new (normal) bnode as the object. Then, to this new bnode, add a ex:storeid property with value 5 and a ex:partid property with value "button". Then, the *_i2* relationship instance is found by querying for all instances that match on the storeid and partid properties.

Summary

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 divide between structured and semi-structured information.

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, 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