Efficient RDF Storage and Retrieval in Jena2

Kevin Wilkinson[1], Craig Sayers[1], Harumi Kuno[1], Dave Reynolds[2]
HP Laboratories
[1] 1501 Page Mill Road
Palo Alto, CA, 94304 USA
[2] Filton Road, Stoke Gifford
Bristol BS34 8QZ United Kingdom
firstName.lastName@hp.com

Abstract. RDF and related Semantic Web technologies have been the recent focus of much research activity. This work has led to new specifications for RDF and OWL. However, efficient implementations of these standards are needed to realize the vision of a world-wide semantic Web. In particular, implementations that scale to large, enterprise-class data sets are required. Jena2 is the second generation of Jena, a leading semantic web programmers' toolkit. This paper describes the persistence subsystem of Jena2 which is intended to support large datasets. This paper describes its features, the changes from Jena1, relevant details of the implementation and performance tuning issues. Query optimization for RDF is identified as a promising area for future research.

1.0 Introduction

Jena is a leading Semantic Web programmers' toolkit[1]. It is an open-source project, implemented in Java, and available for download from SourceForge. Jena offers a simple abstraction of the RDF graph as its central internal interface. This is used uniformly for graph implementations, including in-memory, database-backed, and inferred graphs.

Jena2 is the second generation of the Jena toolkit. It conforms to the revised RDF specification, has new capabilities and a new internal architecture. A design principle for Jena2 was to minimize changes to the API. This simplifies migration from Jena1 to Jena2. This paper describes Jena2 persistent storage; details on other aspects of Jena2 are available in [2].

The Jena database subsystem implements persistence for RDF graphs using an SQL database through a JDBC connection. The Jena1 implementation supported a number of database engines (e.g., Postgresql, MySQL, Oracle) and had a flexible architecture that facilitated porting to new SQL database engines and experimentation with different database layouts. Some options included the use of value-based identifiers (e.g., SHA-1) for inter-table references, use of database procedures, etc. Jena1 also worked with Berkeley DB.

Among the lessons learned from Jena1 was that database portability was valu-

able to the open source community and this was retained as a goal for Jena2. However, Jena1 users did little experimentation with schema flexibility. In general, the default layouts were used. The design focus for Jena2 was performance and scaling. Although Jena1 performance was quite good, there is room for improvement. It was felt that performance issues will become more important with the renewed interest in applying semantic web technology to real-world applications [6]. Jena2 addresses the following performance issues.

Too many joins. The use of a normalized schema requires a three-way join for find operations.

Single statement table. A single statement table doesn't scale for large data sets and cannot take advantage of locality among subjects and predicates.

Reification storage bloat. A naive implementation of the RDF specification stores four statements for each reification. A more efficient technique is required, especially since some applications reify every statement. Jena1 provided optimized storage for reified statements but a statement could only be reified once. Revisions of the RDF specification removed this restriction. The goal for Jena2 was to implement the revised specification with similar or better optimization.

Query optimization. In Jena1, joins for RDQL queries were not pushed-down to the database engine and were instead performed within the Java layer of Jena.

This paper describes how these performance issues were addressed. The next section provides an overview of Jena and RDF for readers unfamiliar with those technologies. More details on RDF are available in [3]. Section 3.0 describes the Jena database schema. Section 4.0 is a high-level overview of the database subsystem of Jena2. Section 5.0 addresses query processing. The final sections cover miscellaneous topics, the status of the implementation and related work.

2.0 Overview of Jena and RDF

2.1 Jena Overview

Jena offers a simple abstraction of the RDF graph as its central internal interface. This is used uniformly for graph implementations, including in-memory, database-backed, and inferred graphs. The main contribution of Jena is a rich API for manipulating RDF graphs. Around this, Jena provides various tools, e.g., an RDF/XML parser, a query language, I/O modules for N3, N-triple and RDF/XML output. Underneath the API the user can choose to store RDF graphs in memory or in databases. Jena provides additional functionality to support RDFS and OWL.

The two key architectural goals of Jena2 are:

- Multiple, flexible presentations of RDF graphs to the application programmer. This allows easy access to, and manipulation of, data in graphs enabling the application programmer to navigate the triple structure.

- A simple minimalist view of the RDF graph to the system programmer wishing to expose data as triples.

The first is layered on top of the second, so that essentially any triple source can back any presentation API. Triple sources may be materialized, for example database or in-memory triple stores, or virtual, for example resulting from inference processes applied to other triple sources.

An simplified overview of the Jena 2 architecture is shown in Figure 1. Applications typically interact with an abstract Model which translates higher-level operations into low-level operations on triples stored in an RDF Graph. There are several different graph implementations, but in this paper we focus on those which provide persistent storage.

At an abstract level, the Jena2 storage subsystem need only implement three operations: add statement, to store an RDF statement in a database; delete statement, to remove an RDF statement from the database; and the find operation. The find operation retrieves all statements that match a pattern of the form <S,P,O> where each S, P, O is either a constant or a don't-care. Jena's query language,

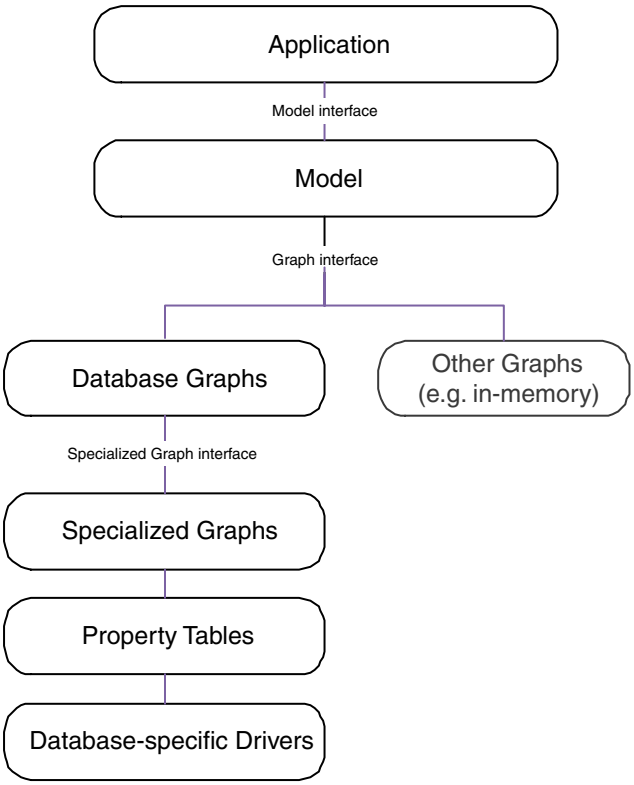

FIGURE 1. Jena2 Architectural Overview

RDQL, is converted to a set of find operations in which variables are permitted in the find patterns. The variables enable joins across the patterns.

A widely used implementation technique [4,5] is to store RDF statements in a relational database using a single statement table, often called a “triple-store.” This is a table that stores each RDF statement as a row and has columns for the subject, predicate and object. Jena1 used this approach but normalized the statement table so that literals and resources are only stored once. This is described below.

2.2 RDF Overview

The Resource Description Framework (RDF) has rapidly gained popularity a means of expressing and exchanging semantic metadata, i.e., data that specifies semantic information about data. RDF was originally designed for the representation and processing of metadata about remote information sources (referred to as resources or Web resources), and defines a model for describing relationships among resources in terms of uniquely identified properties (attributes) and values. RDF provides a simple tuple model, <S,P,O>, to express all knowledge. The interpretation of this *statement* is that subject S has property P with value O, where S and P are resource URIs and O is either a URI or a literal value. RDF and its related specifications, RDF Schema and OWL, provide some predefined basic properties such as type, class, subclass, etc.

RDF is characterized by a property-centric, extremely flexible and dynamic data model. Resources can acquire properties and types at any time, regardless of the type of the resource or property. This flexibility makes RDF an attractive technology for the specification and exchange of arbitrary metadata because resource descriptions are ``grounded'' without necessarily being bound by fixed schemas.

In object-oriented (OO) terms, we might consider RDF resources to be analogous to objects, RDF properties to represent attributes, and RDF statements to express the attribute values of objects. A key difference between the two communities is that unlike OO systems which use the concept of a type hierarchy to constrain the properties that an object may possess, RDF permits resources to be associated with arbitrary properties; statements associating a resource with new properties and values may be added to an RDF fact base at any time.

The challenge is thus how to provide persistent storage for the new RDF data model in an efficient and flexible manner. A naïve approach might be to map the RDF data to XML and simply leverage prior work on the efficient storage of XML. However, the standard RDF/XML mapping is unsuitable for this since multiple XML serializations are possible for the same RDF graph, making retrieval complex. Non-standard RDF-to-XML mappings are possible, and have been used in some implementations. However the simpler mappings are unable to support advanced features of RDF, such as the ability of RDF to treat both properties and statements as resources, which allows metadata describing these elements to be incorporated seamlessly into the data model and queried in an integrated fashion.

Statement Table

Subject	Predicate	ObjectURI	ObjectLiteral
201	202	null	101
201	203	204	null
201	205	101	null

Literals Table

Id	Value
101	Jena2
101	The description - a very long literal that might be stored as a blob.

Resources Table

Id	URI
201	mylib:doc
202	dc:title
203	dc:creator
204	hp:JenaTeam
205	dc:description

FIGURE 2. Jena1 Schema (Normalized)

Many RDF systems have used relational or object databases for persistent storage and retrieval. However, this is not always a good fit and the mapping can be challenging because the semantics of the underlying database model clash with the openness and flexibility of RDF. For example, SQL requires fixed, known column data types; object systems often have restrictions on class inheritance and type membership (changes).

3.0 Storage Schema

In this section we compare the storage of arbitrary RDF statements between Jena1 and Jena2. We then look at optimizations for common patterns of statements.

3.1 Storing Arbitrary RDF Statements

Jena1. The first version of Jena used two different database schemas. One for relational databases, and a special one for Berkeley DB. For relational databases, the schema consisted of a statement table, a literals table and a resources table (Figure 2). The statement table contained all asserted and reified statements and referenced the resources and literals tables for subjects, predicates and objects. To distinguish literal objects from resource URIs, two columns were used. The literals table contained all literal values and the resources table contained all resource URIs in the graph. There was no reference counting to reduce overhead. This schema was very efficient in space as multiple occurrences of the same resource URI or literal value were only stored once. However, every find operation required multiple joins between the statement table and the literals table or the resources table.

The Jena1 schema for BerkeleyDB was quite different. It stored all parts of a

statement in a single row and each statement was stored three times: once indexed by subject, once by predicate and once by object.

Comparing the two approaches, it was observed that Jena graphs stored using Berkeley DB were often accessed significantly faster than graphs stored in relational databases [8]. While part of this could be attributed to the absence of transactional overheads in BerkeleyDB, our intuition was that most of the speed difference was because the Berkeley DB schema used a single access method to store statements and that use of a denormalized relational schema might reduce response times.

Jena2. The Jena2 schema trades-off space for time. Drawing on experience with Jena1, it uses a denormalized schema in which resource URIs and simple literal values are stored directly in the statement table (Figure 3).

In order to distinguish database references from literals and URIs, column values are encoded with a prefix that indicates which the kind of value (codes are not shown). A separate literals table is only used to store literal values whose length exceeds a threshold, such as blobs. Similarly, a separate resources table is used to store long URIs. By storing values directly in the statement table it is possible to perform many find operations without a join. However, a denormalized schema uses more database space because the same value (literal or URI) is stored repeatedly.

The increase in database space consumption is addressed in several ways. First, common prefixes in URIs, such as namespaces, are compressed. They are stored in a separate table (not shown) and the prefix in the URI is replaced by a database reference. It is expected that the number of common prefixes will be small

Statement Table

Subject	Predicate	Object
mylib:doc1	dc:title	Jena2
mylib:doc1	dc:creator	HP Labs - Bristol
mylib:doc1	dc:creator	Hewlett-Packard
mylib:doc1	dc:description	101
201	dc:title	Jena2 Persistence
201	dc:publisher	com.hp/HPLaboratories

Literals Table

Id	Value
101	The description - a very long literal that might be stored as a blob.

Resources Table

Id	URI
201	hp:aResource-WithAnExtreme-lyLongURI

FIGURE 3. Jena2 Schema (Denormalized)

and cacheable in memory. Expanding the prefixes will be done in memory and will not require a database join.

Second, as mentioned earlier, long values are stored only once. The length threshold for determining when to store a value in the literals or resources table is configurable. Applications may trade-off time for space by lowering the threshold. Third, Jena2 supports property tables as described below. Property tables offer a modest reduction in space consumption in that the property URI is not stored.

Both Jena1 and Jena2 permit multiple graphs to be stored in a single database instance. In Jena1, all graphs were stored in a single statement table[1]. However, Jena2 supports the use of multiple statement tables in a single database so that applications can flexibly map graphs to different tables. In this way, graphs that are often accessed together may be stored together while graphs that are never accessed together may be stored separately (Figure 6). For example, as described below, the system metadata is stored as RDF statements in its own statement table separate from user tables. The use of multiple statement tables may improve performance through better locality and caching. It may also simplify database administration since the separate tables can be separately managed and tuned.

3.2 Optimizing for Common Statement Patterns

An RDF graph will typically have a number of common statement patterns. One source of those patterns is the RDF specification itself which defines some types and properties for modeling higher-level constructs such as bags, sequences and reification. For example, if object *x* is a sequence, it will have a type property with value *rdf:Seq* and one or more element properties, *_1*, *_2*, *_3*, etc. that specify elements of the sequence. A reified statement (i.e., a statement about some other statement) has a type property with value *rdf:Statement* and three properties, *rdf:subject*, *rdf:predicate*, *rdf:object* for values of the statement's subject, predicate and object.

The other source of common patterns is the user data. Applications typically have access patterns in which certain subjects and/or properties are accessed together. For example, a graph of data about persons might have many occurrences of objects with properties name, address, phone, gender that are referenced together. Using knowledge of these access patterns to influence the underlying database storage structures can provide a performance benefit. Techniques for detecting patterns in user data and in RDF query logs are reported in [16].

Jena1. In Jena1, the commonly-occurring case of reified statements was handled as a special-case. Rather than storing four separate triples for each reified statement, it stored the reified subject, predicate and object in the regular statements table, with two additional columns to indicate its reified state and to store a statement identifier. Since a statement had only one identifier, it could only be reified once. For

1. In Jena1 and Jena2, tables include a column for the graph identifier; this is not shown.

Jena2, changes were required to conform with the revised RDF specification that allows multiple reified instances of any statement.

Jena2 Property Tables. Jena2 will provide a general facility for clustering properties that are commonly accessed together. A Jena2 property table is a separate table that stores the subject-value pairs related by a particular property. A property table stores *all* instances of the property in the graph, i.e., that property does not appear in any other table used for the graph. For properties that have a maximum cardinality of one, it is possible to efficiently cluster multiple properties together in a single table. A single row of the table would store those property values for a common subject. For example, in Dublin Core, it may be beneficial to create a property table for the three properties dc:title, dc:publisher, dc:description if these properties are frequently accessed together. The use of such a property table for the data in Figure 3 is presented in Figure 4.

Multi-valued properties may be clustered or may be stored in a separate table. For example, dc:creator might be stored in a multi-valued property table containing two columns, one for the subject and one for dc:creator. Alternatively, it might be stored with the same property table as title, publisher and description although this may be less efficient if it results in many null-valued columns for a row. At first glance, it may seem that multi-valued property tables offer little benefit. However, there may be benefits to clustering values if they are frequently accessed together, e.g. a set of values that is searched as a lookup table. Note that property tables offer a small storage savings because the property URI is not stored in the table, and for clustered property tables, the subject is only stored once.

For some properties, the datatype of the object value will be fixed and known. It may be specified as a property range constraint. Property tables can leverage this knowledge by, when possible, making the underlying database column for the property value match the property type[1]. This may enable the database to better optimize the storage and searching of the column.

Jena 2 Property-Class Tables. A property-class table is a special kind of property

Statement Table

Subject	Predicate	Object
mylib:doc1	dc:creator	HP Labs - Bristol
mylib:doc1	dc:creator	Hewlett-Packard

DC Properties Table

Subject	Title	Publisher	Description
mylib:doc1	Jena2	-	101
201	Jena2 Persistence	com.hp/HPLaboratories	-

FIGURE 4. Dublin Core Property Table

table that serves two purposes. It records all instances of a specified class, i.e., resources that have that class. It also stores properties of that class, i.e., each property in the table must have the class as its domain. Thus, a property-class table has two or more columns: one for the subject resource, a second boolean column indicating if the subject has been explicitly asserted as a class member (as opposed to inferred as a member), and zero or more columns for property values.

It is worth noting that Jena2 implements reification as a property-class table (Figure 5). The properties are rdf:subject, rdf:predicate, rdf:object and the class is constrained to be *rdf:Statement*. The subject of the property-class table is the URI that reifies the statement. Storing a reification this way saves space compared to the alternative of explicitly storing the four statements of a reification. Note that partially reified statements are easily supported.

Applications specify the schema for a graph, i.e., the property, property-class and statement tables at graph creation time through the configuration meta-graph. To simplify the implementation, once defined, the table configuration cannot be altered. However indexes may be added or removed. In the future, some limited changes to the table configuration may be enabled.

4.0 Jena2 Persistence Architecture

An overview of the Jena2 persistence architecture was presented in Figure 1. In this section, we describe the implementation of that architecture, including the specialized graph interface that implements RDF sub-graphs and the database drivers that access the database on their behalf.

4.1 Specialized Graph interface

The Jena2 persistence layer presents a Graph interface to the higher levels of Jena, supporting the usual Graph operations of add, delete and find (Figure 6). Each logical graph is implemented using an ordered list of specialized graphs; each of which is optimized for storing a particular style of statement. For example, in the figure the first logical graph is implemented using three specialized graphs: one optimized for reified statements, another optimized for ontology triples and a third which han-

Reified Statement Table

StmtURI	Subject	Predicate	Object	Type
mydir:alice	mylib:doc1	dc:title	Jena2	rdf:Statement
mydir:bob	mylib:doc2	-	Jena2	-

FIGURE 5. Reification as a Property-Class Table

1. Not all XSD types correspond to an SQL datatype.

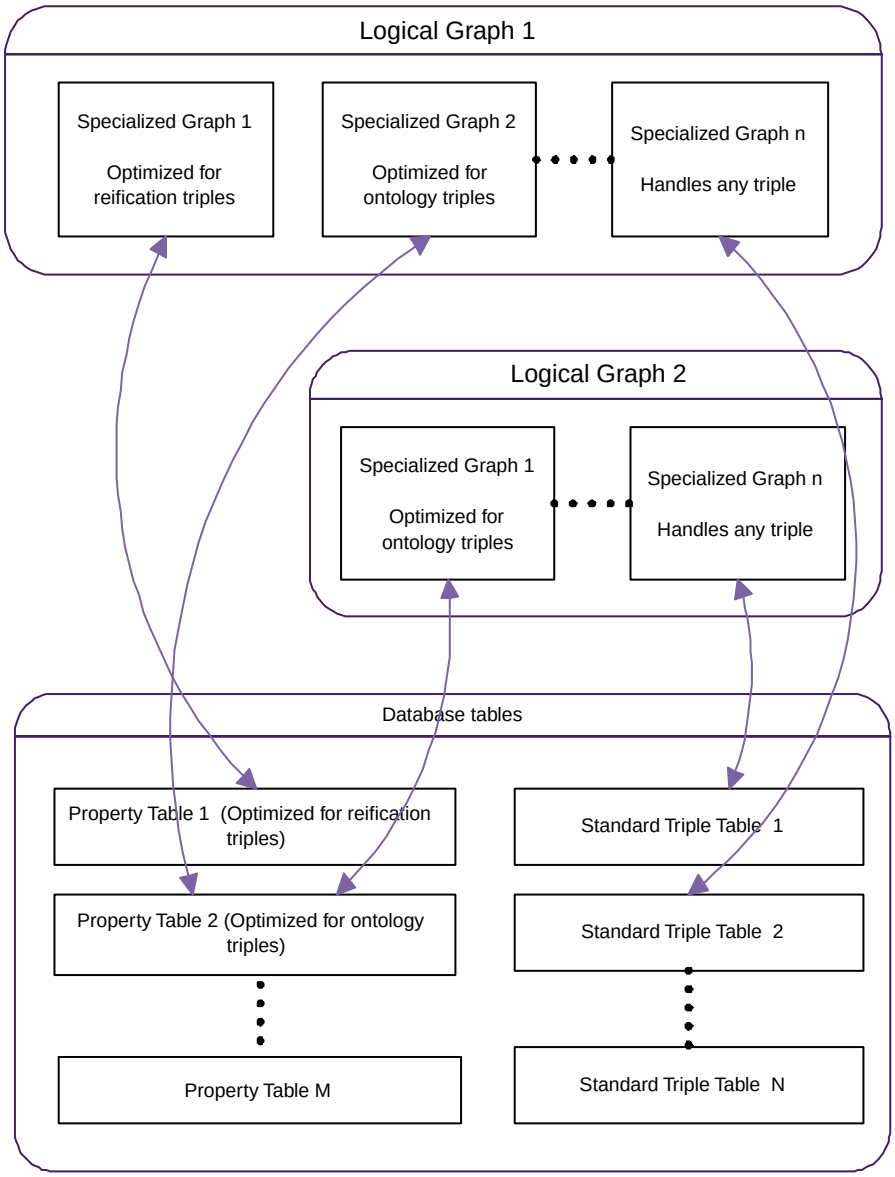

FIGURE 6. Graphs Comprise Specialized Graphs Over Tables

dles any remaining triples.

An operation on the entire logical graph, such as add statement, delete statement or find, is processed by invoking add, delete, find on each specialized graph, in turn. The results of the individual operations are combined and returned as the

result for the entire graph.

Note that this process can be optimized because, in certain cases, an operation can be completely processed for the entire graph by one specialized graph. Thus, the operation need not be invoked on the remaining specialized graphs. For example, a specialized graph that stores every statement with a property of dc:title can process all add and delete operations that reference dc:title and can fully satisfy any request to find all such properties. To support this optimization, each specialized graph operation returns a completion flag to indicate if the operation has been completely processed and the ordering of the specialized graphs is kept constant.

In the case of a find operation, an additional optimization, which the specialized graphs permit, is to evaluate the find on each graph in a lazy fashion; using resources from later specialized graphs only if the application is still hungry after consuming results from earlier graphs.

Each specialized graph maps the graph operations onto appropriate tables in the database. In the present implementation, there is a many-to-one mapping between specialized graphs and database tables. In some cases, this allows the overhead of each database table to be amortized across several graphs.

4.2 Database Driver

The database driver provides an abstract storage interface that insulates the specialized graphs from differences in how database engines support blobs, nulls, expressions, table and index creation, etc. There is a generic driver implementation for SQL databases and engine-specific drivers for Postgresql, MySQL, Oracle, etc. The engine-specific drivers override the generic methods as necessary, e.g., for different quoting conventions or treatment of blobs.

The driver is responsible for data definition operations such as database initialization, table creation and deletion, allocating database identifiers. It is also responsible for mapping graph objects between their Java representation and their database encoding. For data manipulation, the drivers use a combination of static and dynamically generated SQL. The static SQL is used for fixed, predefined operations such as inserting a triple in a graph or the various forms of the find operation. For access to property-class tables and for RDQL queries, the drivers dynamically generate SQL select statements. To reduce the overhead of query compilation, the driver layer maintains a cache of prepared SQL statements.

The driver uses a storage abstraction that is designed to be mapped to other persistent stores. Non-SQL drivers are also possible. In the future, we plan to support a Berkeley DB driver and a native-Java persistent store.

4.3 Configuration and Meta-graphs

In Jena1, database configuration parameters and options were specified in a configuration file of property-value pairs that was read when initially connecting to the database. In Jena2, the files are not used. Instead, configuration parameters are

specified as RDF statements. This is analogous to storing metadata for relational databases in tables. A graph containing configuration parameters may be passed as an argument when creating a new persistent graph. Jena2 provides default graphs containing the default configuration parameters for all supported databases.

Associated with each Jena2 persistent store is a meta-graph, a separate, auxiliary RDF graph containing metadata about each logical graph. This auxiliary graph includes the configuration parameters and options mentioned above as well as other metadata such as the date the database was formatted, the version of the driver, a list of graphs stored in the database, the mapping of graphs to tables, etc. The meta-graph may be queried just as any other Jena graph but, unlike other graphs, it may not be modified and it does not support reification.

The default schema for a graph is a statement table and a reified statement table, implemented as a property-class table. The user-provided meta-graph may specify that graphs share tables. The meta-graph may also specify additional property, property-class tables and indexes. Parameters such as the threshold size for long literals and resources are also specified as statements within the meta-graph.

5.0 Jena2 Query Processing

There are two forms of Jena querying. The find operation returns all statements satisfying a pattern. A find pattern has the form (S,P,O) where each element is either a constant or a don't-care. An RDQL query [17] is compiled into a conjunction of find patterns that may include variables to specify joins. It returns all possible valid bindings of the variables over statements in the graph[1].

The addition of property tables significantly complicates query processing. Consider iterators. Unlike a statement table where each row corresponds to a single RDF statement, an iterator over a property table may need to expand a row into multiple statements and add URIs for properties that are not explicitly stored. In addition, columns in a property table can be null.

However, the major complexity occurs when a query references an unknown property, i.e., where the property is a don't-care or a variable that will be bound when the query is processed. These cases are discussed below.

5.1 Find Processing

In Jena1, a find pattern was evaluated with a single SQL select query over the statement table. In Jena2, this has to be generalized because there can be multiple statement tables for a graph. To evaluate a pattern in Jena2, the pattern is passed, in turn, to each specialized graph handler for evaluation, stopping when the completion flag is set. The results are concatenated and returned to the application. This handles the

1. Querying over inferred graphs is not addressed here.

case when the pattern contains an unspecified property, i.e., a don't-care (note that find operations do not have variables).

Currently, each specialized graph issues a separate database query for the pattern. We plan to investigate if a single database query over all specialized graphs would be more efficient. For example, suppose the pattern is (*,*,*), which retrieves all statements, and suppose the graph has two tables, a statement table and a reified statement table. Rather than two separate queries, the following single SQL query could be used to process the pattern.

```
Select t.subject, t.predicate, t.object from stmt_table t
Union
Select r.URI, "rdf:subject", r.subject from reif_table r
   where r.subject is not null
Union
Select r.URI, "rdf:predicate", r.predicate from reif_table
   r where r.predicate is not null
Union
Select r.URI, "rdf:object", r.object from reif_table r
   where r.object is not null
Union
Select r.URI, "rdf:type", r.type from reif_table r where
   r.type is not null
```

Such queries can quickly become unwieldy for complicated patterns and several statement tables and may cause challenges for query optimizers. In addition, it is not clear that a single, large union query is more efficient that the alternative of issuing two separate queries. With the single, union query, rows in the reification table are read four times.

5.2 RDQL Processing

In Jena1, an RDQL query is converted into a pipeline of find patterns connected by join variables. The query is then be evaluated in a nested-loops fashion in Jena by using the result of a find operation over one pattern to bind values to variables and then generating patterns for new find operations. It would be more efficient if the join could be pushed into the database engine for evaluation.

This is a goal of Jena2 query processing, i.e. converting multiple triple patterns into a single query to be evaluated by the database engine. A full discussion of query processing is beyond the scope of the paper. In this section, we discuss two simple cases and mention the difficulties for the general case.

For the first simple case, assume that the find patterns for a query reference only the statement table, i.e., it can be determined a priori that statements in the property tables match none of the patterns. As mentioned above, a single pattern can be completely evaluated by a single SQL query over the statement table. To evaluate multiple patterns, it is sufficient to associate a table alias with each pattern and perform a join across the aliases for linking variables in the find patterns. For

example, consider an RDQL query to get the authors of a paper. It requires two patterns, each of which has an associated SQL evaluation expression.

```
Pattern 1: (Var1, dc:title, "Jena2")
Pattern 2: (Var1, dc:creator, Var2)
Select p1.subject, p2.object
from stmt_table p1, stmt_table p2
where p1.predicate = "dc:title" and p1.object="Jena2" and
   p2.predicate="dc.creator" and p1.subject = p2.subject
```

The second simple case occurs when all patterns can be completely evaluated by a single property table. The interesting thing about this case is that it is possible to eliminate joins if the patterns reference single-valued properties clustered together. For example, suppose there is a clustered property table for dc:title and dc:creator (assume creator is single-valued here). Then, the two patterns in the previous example require only a single table alias and can be evaluated without a join.

```
Select p1.subject, p1.creator from dcPropertyTable p1
where p1.title="Jena"
```

When the find patterns for a query apply to multiple tables, it is more difficult to construct a single SQL query to satisfy all patterns. This presents the same issues as generating a single SQL query for a find operation. The current approach in Jena2 is to partition the patterns into groups, where each group contains patterns that can be completely evaluated by a single table, plus one additional group containing patterns that span tables. A SQL query is then generated for the former groups and the latter group is processed using the nested loops approach as in Jena1.

In Jena2, there are three cases in which a pattern may span tables. First, the property may be a don't-care in which case all tables must be searched. Second, the property may refer to an unspecified class, i.e., the property is *rdf:type* but the object value (the class) is not specified. In this case, it is impossible to know which property table may contain values for the pattern. Third, the property may be a variable. This is the most interesting case as it corresponds to table variables in relational database querying processing, i.e., the table name is unknown until the query is processed. This is a difficult query processing problem.

Finally, a feature of Jena2 is that queries may span graphs. This is done by specifying the graph to which a pattern applies. If the graphs reside in the same database instance, it is possible to optimize those query patterns as if they were all part of the same graph. If the graphs reside in different instances or different database engines, no attempt is made to optimize the query and the basic nested-loop approach is applied.

6.0 Miscellaneous Topics

Jena2 Performance Toolkit. To explore various layout options and understand

performance trade-offs, a set of Jena utility programs are under development. The first is an RDF synthetic data generator that generates statements for a specified number of classes and instances. Uniform and skewed data distributions can be generated as well as predefined reference patterns for properties, such as trees (for taxonomies, ancestor relations, etc.).

The second tool is a benchmark suite to measure the effectiveness of Jena2 enhancements and to compare different database layouts. It is designed as a general framework that can be used to make comparative runs of an arbitrary set of Jena programs. The third tool is an RDF data analysis tool that, when applied to a set of RDF statements, suggests potentially beneficial property and property-class tables to store the statements [16].

Jena Transaction Management. In Jena1, the underlying database may or may not support transactions. Consequently, all Jena API methods were atomic to ensure database consistency. In addition, transaction *begin*, *commit* and *abort* methods were available to declare explicit transactions when desired. Jena2 provides the same capabilities. However, there is an interesting case in which Jena cannot ensure database consistency. The Jena2 query handler supports queries across graphs. If the graphs are stored in separate databases, then a consistent read-set for the query cannot be guaranteed because a Jena2 transaction applies to a single database connection.

In principle, it should be possible to open an XOpen/XA distributed transaction connection to the other data source to ensure consistency. However, in the open world of the semantic web, the common case is that data sources do not support transactions, let alone the XA protocol. This suggests that a richer transaction interface for Jena2 is needed and it remains future work.

Bulk Load. A goal of Jena2 was to significantly reduce the time to load persistent graphs compared to Jena1. This is a critical issue if RDF is to be applied to very large datasets. The use of a denormalized schema helps address this problem since a typical Jena2 add operation updates fewer tables than Jena1. Jena2 also includes support for JDBC2 batch operations which enable multiple JDBC statements to be passed in one call to the database engine. The value of batching depends on the level of optimization within the database engine but in any event it reduces the number of database calls significantly.

7.0 Status and Future Work

Performance Notes. Preliminary performance measurements indicate that the denormalized schema of Jena2 is faster than the normalized schema of Jena1, twice as fast for many operations. The results of one simple retrieval experiment are presented in Table 1. The test retrieved a single, 200-byte property value for 1000 randomly selected objects. The test was run under two configurations. The denormalized configuration stored the property value directly in the statement

table. The normalized configuration reduced the long literal threshold (see Section 3.1) to 100 bytes which caused the property value to be stored in the literals table.

Thus, retrieving the property value in the denormalized configuration requires two retrievals, one for the statement table and a second for the literals table while the denormalized case requires only one. Each configuration was run multiple times with different random seeds and the result of the first and final run are presented. The times are in milliseconds and the tests were run using MySQL under WinXP on a recent generation PC workstation with 1.5GB RAM.

The large reduction in run time for the initial run compared to the final run we attribute to hardware cache effects. For the warm run, as expected, the denormalized retrieval is twice as fast as the normalized retrieval. If the schema were completely normalized so that the subject and predicate were also stored in separate tables as was done in Jena1, we would see an even greater speed-up for the denormalized schema. A more systematic study will be done upon completion of a Jena performance toolkit. Similarly, the database size increase due to the denormalized schema has not been studied pending impletion of URI prefix compression.

TABLE 1. Retrieval Times for Normalized vs. Denormalized Literal

Number of Retrievals	Normalized	Denormalized	Speed-up
1000 (initial run)	3270	2850	1.2
1000 (final run)	840	420	2.0

Next we provide some preliminary results that show the value of property-class tables for reification. A synthetic database of 10,000 reified RDF statements was generated and stored in two different formats. In the first case, the reified statement was stored in an optimized form as a property-class table. In the second case, the reified statement was stored unoptimized as RDF triples, i.e., each reified statement was stored as four RDF statements. Consequently, the first table contained 10,000 rows while the second table contained 40,000 rows.

Then a simple test program randomly selected a reified statement and retrieved the four reification triples for that statement (recall that on retrieval, the property-class table converts each table row to a set of triples). Each test was run four times with different random number seeds and three different test sizes were run of 200, 1000, 5000 retrievals. The results are presented in Table 2. As before, the times are in milliseconds and the tests were run using MySQL under WinXP on a recent generation PC workstation with 1.5GB RAM.

Our expectation was that the optimized format would perform anywhere between one and four times faster than the unoptimized form since it only needs to invoke the database engine once to get all four triples whereas the unoptimized format makes four calls. For a small number of retrievals, the optimized format shows a large improvement between the first and fourth run. We attribute this to caching effects that decrease with larger numbers of retrievals. It is interesting to see that the speed-up for large numbers of retrievals exceeds our expectations. This may be

TABLE 2. Retrieval Times for Four Triples of a Reified Statement

Number of Retrievals	Optimized	Unoptimized	Speed-up
200 (initial run)	1000	1860	1.8
200 (final run)	270	1470	5.4
1000 (initial run)	1330	7380	5.5
1000 (final run)	700	6970	10.0
5000 (initial run)	4220	34380	8.1
5000 (final run)	3470	34270	9.9

due to database caching effects. Since the optimized table is smaller, it is possible to cache a larger percentage of the entire table which reduces the number of relatively slow disk seek operations.

A comprehensive study of RDQL query processing has not been done. Some preliminary analysis indicates that the Jena2 algorithm is a modest improvement over the Jena1 nested-loops approach. The Jena1 algorithm works quite well for queries with high selectivity since such queries require few nested find operations. For such queries, Jena1 and Jena2 perform about the same. Jena2 performs better than Jena1 on queries which join a large number of tuples.

Future Work. Currently, Jena2 stores all literals as strings. An important enhancement for typed literals will be to store them as native SQL types. This will enable inequality comparisons and range queries to be processed within the database engine. This is future work.

A major goal of Jena2 is support for OWL and reasoning. Now that this is available, it will be interesting to explore how the persistence layer can better support these capabilities, e.g. performing transitive closure within the database.

We are presently investigating caching strategies to improve performance. One approach is to implement the caching inside the persistence layer using the specialized graph interface. An alternative is to implement caching for arbitrary logical graphs. The latter provides a convenient general-purpose solution, while the former may make use of intimate knowledge of the database to improve performance. Our initial caching algorithm is to implement a write-through cache which holds statements with commonly-used subjects. If the cache holds one statement with subject=X then it has every statement with subject=X. Currently, the cache assumes exclusive access to that subject to avoid cache consistency issues due to conflicting updates from other Jena applications. However, such exclusive access appears to be a common case. This style of cache has previously been suggested by others with experience in using RDF with Jena1 [11] and we hope will prove to be a good match for common application usage patterns. Testing and analysis is underway.

8.0 Related Work

A good introduction to RDF storage subsystems and a comparative review of

implementation is available in [4,5]. We do not attempt to duplicate such a survey here. However, if we compare the Jena2 persistent store to some of these systems along the dimensions of database schemas, architecture, and system functionality, then we can better characterize the strengths and limitations of our approach.

The Jena2 schema design is unique in that it supports two basic schema types: both a denormalized schema used for storing generic triple statements as well as property tables to store subject-value pairs related by arbitrarily specified properties. To the best of our knowledge, no other system supports the generation of property tables based on arbitrary properties; other systems are strictly schema-specific. Jena2 uses the arbitrary property tables to implement a novel architecture where the statements associated with a given graph are stored in multiple specialized subgraphs. This architecture enables the Jena2 query processor to effectively treat the subgraphs as data partitions and provides an efficient implementation for reification.

Most systems (including KAON [9,], Parka Database[13], and rdfDB[14]), support only a fixed set of underlying tables that implement a (non-schema-specific) generic store. This means that the storage mechanism cannot adapt to the data characteristics, impacting scalability.

ICS-FORTH's RDF Suite [10] supports both generic stores as well as automatically-generated schema-specific Object-Relational (SQL3) schema definitions. However, unlike Jena2, RDF Suite relies on schema specifications to create the specialized tables; it doesn't support arbitrary property tables. Similarly, the Sesame [15] system creates one specialized table per class. Tightly coupling the table layout to schema structure can facilitate inferencing by allowing the systems to exploit the explicit schema relationships, but it also means that the tables must be rebuilt whenever the schema structure changes. This forces the storage system to forfeit RDF's unique support for flexible dynamic schema restructuring; Jena2 is not subject to this limitation.

Insofar as the schema-specific tables partition the stored data, such schema-specific storage resembles the Jena2 notion of specialized subgraphs. However, because these systems tightly couple the subgraphs with the schemas, they can only partition data according to its syntactic structure; they cannot create subgraphs based on other factors. The Storage and Inference Layer (SAIL) [15] provides layered interfaces to Sesame modules that stack and allow actions to be passed between them until handled. However, because it based upon Sesame, the SAIL database schema is class-specific, and thus subject to the limitations listed above.

To the best of our knowledge, no other RDF system optimizes storage for reification in the style of Jena2. The notion of property-class tables appears to be new in RDF stores although it is commonly used in object and functional database systems.

9.0 Conclusions

The Jena2 persistence layer supports application-specific schema while retaining the flexibility to store arbitrary graphs. The notion of property-class tables appears to be new and should be beneficial for query languages that expose higher-level abstractions to applications. However, the mixing of property tables and statement tables in a graph database complicates query processing and optimization. More work is needed on efficient algorithms for this case.

Acknowledgements

The first three authors are new to the Jena effort and wish to thank the rest of the Jena team, which includes the forth author, for their help and for allowing us to participate in its development. The rest of the Jena team includes Jeremy Carroll, Ian Dickinson, Chris Dollin, Brian McBride and Andy Seaborne. For the work in this paper, we particularly thank Chris Dollin and Andy Seaborne for being so generous with their time.

References

1. B. McBride Jena IEEE Internet Computing, July/August, 2002.
2. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, The Jena Semantic Web Platform: Architecture and Design, HP Laboratories Technical Report HPL-2003-146.
3. T. Berners-Lee et al. Primer: Getting into RDF & Semantic Web using N3, http://www.w3.org/2000/10/swap/Primer.html
4. D. Beckett, SWAD-Europe: Scalability and Storage: Survey of Free Software / Open Source RDF storage systems, http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/
5. D. Beckett, J. Grant, SWAD-Europe: Mapping Semantic Web Data with RDBMSes, http://www.w3.org/2001/sw/Europe/reports/scalable_rdbms_mapping_report/
6. T. Berners-Lee, Web Services and Semantic Web, keynote speech at World Wide Web Conference, 2003, http://www.w3.org/2003/Talks/0521-www-keynote-tbl/
7. S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, K. Tolle, The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases, 2nd Intl Workshop on the Semantic Web (SemWeb'01, with WWW10), pp. 1-13, Hongkong, May 1, 2001.
8. D. Reynolds, Jena Relational Database Interface – Performance Notes, in Jena 1.6.1 download: http://www.hpl.hp.com/semweb/download.htm
9. KAON - The Karlsruhe Ontology and Semantic Web Tool Suite, http://kaon.semanticweb.org/

10. J. Broekstra, A. Kampman, F. van Harmelen, Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema PDF, First International Semantic Web Conference (ISWC'02), Sardinia, Italy, June 9-12, 2002.
11. D. Banks, personal communication.
12. The ICS-FORTH RDFSuite: High-level Scalable Tools for the Semantic Web. http://139.91.183.30:9090/RDF/
13. PARKA-DB - A Scalable Knowledge Representation System.
14. http://www.guha.com/rdfdb/internals.html
15. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. http://sesame.aidministrator.nl/publications/del10.pdf
16. L. Ding, K. Wilkinson, C. Sayers, H. Kuno, Application-Specific Schema Design for Large RDF Datasets, HP Laboratories Technical Report HPL-2003-170.
17. A. Seaborne, An RDF NetAPI, , HP Laboratories Technical Report HPL-2002-109.