PNWD-4230

Hybrid Triple/Relational Knowledge Store Concepts[1]

Alan Chappell, Patrick Paulson, Eric Stephan, and Cliff Joslyn

[alan.chappell; patrick.paulson; eric.stephan; cliff.joslyn]@pnl.gov

Pacific Northwest National Laboratory

1. Summary

This whitepaper describes a mechanism for representing and storing both triple-based information and information about those triples. Triples refer to information coded as two entities and a relationship between them. Triple-based information can be stored in a "triple store" which supports pattern-based querying for retrieval of the information.

We discuss a mechanism to augment the triple store with a coordinated relational store to hold additional information about the triples that is difficult to store in a triple store alone. This will enable more effective storage of triple-based information along with descriptive and quantitative data recording provenance, uncertainty quantification, temporal coding, and other metadata about the individual triples. This will further enable the productive use of the additional information in querying processes to enhance the efficiency of triple retrieval.

Such a hybrid store mechanism removes the need for reification. No extra statements are required in the triple store so there is no associated bloat. Patterns of interest are not changed, so querying is only as complex as the desired pattern. Further, the hybrid may actually improve performance of querying over triple stores by allowing for the segmenting of the triples, thereby substantially reducing the number of triples over which pattern matching query mechanisms must search.

Section 2 describes the overall goals of this project, Section 3 defines the requirements the implemented system must meet, Section 4 gives an overview of the adopted design along with a proposed implementation approach and Section 5 discusses other considered alternatives.

[1] This paper written as part of the Battelle Threat Anticipation Initiative. Inquiries on the content should be directed to the authors. Inquiries on the initiative should be directed to Imran Bashir [bashiri@battelle.org].

2. Goal

As part of an ongoing effort to implement a tool that depends on semantically informed inference processes, this effort is supporting storage of and access to information represented as triples. Triples refer to information coded as two entities and a relationship between them. A triple-based representation was chosen for its support and enabling of many aspects of the overall goals of the effort. The majority of information of interest to the target system can easily and understandably be represented as triples. A set of widely used standards, technologies, and research communities exist for triple-based representation and for access, i.e., query, of that representation. For example, RDF (W3C 2009) and SPARQL (W3C 2008) are widely used standards for formalization of triples and triple querying respectively.

However, through the exploration of desired use cases, we have identified the need to represent supplemental information about the triples. The user may wish to know the source for a given piece of information. Similarly, they may wish to restrict inference processing to what was known before a given point in time. For these and other situations, provenance and other descriptive information are needed about the triples. While the standards discussed above support the representation of this information via reification, the formalisms provided diminish many of the natural benefits of the triple-based representations. Using reification, descriptions of data become larger, abstract, and complex. Section 5.5.1 discusses use of reification and the associated issues in greater depth. For these reason, here we explore other approaches that preserve the benefits of a triple-based representation, but allow the desired augmentation through representation of descriptive information about the triples.

3. Requirements

The representation of metadata about a triple is a problem generic to may uses of triple-based information beyond this specific effort. While little formal research is being published in this area, many commercial projects are developing custom, focused solutions with little effort to generalize.

The following sections discuss the detailed system and functional requirements for this effort. These requirements guided our development process and also act to characterize a class of problems to which this solution may generalize. Other systems with similar requirements are likely to find the proposed system useful in addressing those requirements.

3.1. System Requirements

3.1.1 Adherence to Standards

Adherence to standards has been established as a priority for this project. Adherence to standards enables the use of technologies and communities built around the standards. The availability of such tools and assistance can significantly reduce development cost and promote reuse. Further, adherence to standards avoids single-vendor solutions and enables greater flexibility and adaptability. As new community technologies are developed that may outperform current tools, those new tools may be exploited without large additional investments. Only when other system requirements preclude standards adherence will such methods be adopted. When non-standard elements become necessary, boundaries around those elements will be established as narrowly as reasonably possible. These boundaries will encapsulate the non-standard aspects and support standards-based interaction across the boundaries.

3.1.2 Performance

Specific performance requirements have not been determined; this would require an estimate of the size of the applications knowledge base in triples, example queries from a user interaction, and performance requirements for user interactions. However as a basis we plan to perform tests along the guidelines of the Semantic Interoperability of Metadata In like and unLike Environments (Lee 2004) project who performed bench mark tests on a number of Open Source RDF stores browsing large data stores. In these tests the metrics of primary interest will be related to load speed of the store, load speed of the browser, and queries with large expected results.

3.1.3 Cost

No maximum cost requirement has been determined, but designs that enable free implementations are strongly favored.

3.2. Functional Requirements

3.2.1 Ability to annotate statements

The primary functional requirement of the store is to support the addition of structured metadata to the binary relations represented in the triple store. These annotations will be domain and application specific. A particular application will have a fixed number of pre-defined annotations that can be applied to each statement. While the set of these annotations may evolve over time, such evolution is infrequent and is primarily additions. The annotations could include a wide range of information including date and time ranges, source references, confidence, and many types of provenance, metadata, and knowledge refinement. An RDF

Property resource will be associated with each annotation, defining the semantics of the annotation. Table 1 gives an example list of annotations for a particular application.

Table 1 A list of possible annotations

Annotation	*Description*
Source	At least the URL associated with the source document
Probability	A number between zero and one indicating a subjective probability or a frequency
Uncertainty interval	A lower and upper bound on the uncertainty of the statement, perhaps a confidence interval or imprecise probability
Temporal interval	Pair of quantities indicating start and end times
Knowledge Refinement	Additional information refining the knowledge represented by the triple. Example: negation of a statement

3.2.2 Accessing metadata associated with a triple

For any particular triple, the system must provide access to set, modify, and retrieve the annotations on that triple. When requested, the system will provide the values of all the annotations or of specified annotations based on user request.

3.2.3 Partitioning of triples according to attached metadata

For given values and ranges of metadata, the system must identify and return the set of triples that have the given value or fall within the given range. The system will allow annotation level configuration over behaviors when data is not present. For example, if a date range query is requested, the system will support both inclusion and exclusion of triples with no date specified. The user will be able to select the desired behavior.

When appropriate, the system will further support pattern-based (SPARQL) querying over the partitioned sets of triples.

3.2.4 Pattern-based querying of metadata is not required

Triple-based representation is extremely helpful in formulating and supporting pattern-based queries of the data. These types of queries are either not required, or are of very limited use over the triple annotations. Instead, queries over the annotations are predominately describable independently of values on other annotations. Hence the system need not support pattern-based queries including metadata values.

3.2.5 Ability to generate RDF representation of annotations

To support interactions with other RDF-based systems, support should be provided by the store to produce valid RDF for all represented data when requested. Hence, even if reification is not used for storage, the system should be able to generate RDF-standard reified statements and associated annotations on those reifications.

4. Design

Given the above requirements, we propose a design for a hybrid triple-RDB store that fulfills those requirements. The sections below describe this design and provide some interpretations of such a system. We also describe a possible implementation of this design. Finally, in the following section, alternative designs that were explored are discussed.

4.1. Hybrid Triple-RDB Store

This design exploits the separable nature of the two sets of data, that represented in the triples and that in the annotations. Triples continue to be stored in a traditional triple store. This store is augmented with a coordinated relational store for the metadata. The coordination between the stores is critical as it allows the hybrid system to exploit both stores. This coordination is accomplished by creating an identifier for each triple. In simple triple stores this can be a functional combination of the identifiers for the three elements of the triple, such as concatenation. If the generated identifier already exists, i.e. the same triple already exist perhaps from a different source, then simple augmentations can be generated to identify different triples. More advanced triple stores support a named subgraph or context for each triple so that arbitrary identifiers can be assigned. In such cases, a unique identifier for each triple, such as a URI, would be created and assigned within the triple store.

This identifier would then be used as the key into the relational store recording additional information about the relationship. Contents of the relational store would be domain and application specific but could include a wide range of information.

This segregation of the data will support efficient retrieval of information when the request is cleanly decomposable into the separate aspects. Typical query mechanisms over the triple store would function unchanged. Subsequent to the graph query though, it would be easy to find all sources for information using the unique identifiers of all the triples returned. Similarly, a query could be restricted by first finding the identifiers in the relational store that meet the requested criteria, then partitioning those triples for subsequent standard pattern querying.

As described above, the resultant triple store would not be standard RDF. There are easy extensions to the information that would move the data incrementally closer to the standard if needed by an application. First, the triple identifier could also be used as an entity in the triple store allowing for the triple based representation of any of the information in the relational store or even other information of interest to the system. This could be necessary to support queries that are not cleanly separable into the triple and relational aspects as described above. When the queries, or parts of the queries, are separable, the hybrid approach will likely yield better performance than the complex graph querying than would be required over pure graph solutions.

Finally, if completely standard RDF is required, perhaps for interaction with another system, then RDF standard mechanisms (reification) would be added to the stored triple information. These could be added to the triple store, but more likely would be added post processing to limit the amount of triple bloat in the triple store cause by reification. Reification further enables the description of information without stating its truth. As such, the hybrid store would require additional information in the relational indicating which triples are asserted and which are not. This type of additional knowledge can be further extended to negation of statements.

Our conceptual design consists of an API that provides the following methods against the knowledge store:

1. Given criteria for metadata, return an RDF Graph
2. Given a statement id, return the associated metadata from the knowledge store
3. Accept a new triple with the knowledge store along with associated metadata and return the generated URI for the new statement
4. Update the metadata for a statement with a given URI as the statement id

By ‘RDF Graph’ we refer to a collection of triples that can be operated API. For example, if the implementation is targeted for the Jena API (HP Labs Semantic Web Research 2004) the RDF graph would be represented by a Jena Model; for the OpenRDF API (openRDF.org 2008) the RDF graph might be represented by a SailRepository.

4.2. Interpretations of a separate attribute store

There are several ways to conceptually interpret the described coordinated use of a relation store with a triple store. We can view the relation store as a compilation of the RDF standard reifications. We can also view the relational store as “live” properties on the triples. We explore both these interpretations below.

4.2.1 "Compilation" of RDF reification statements

We can view the triples in the triple store and the associated rows in the relational store as a compilation of the fully reified statements in a standard RDF representation. This compilation reduces the size of the "original" representation and puts the information in forms more effectively used to achieve system outcomes. Care must be taken for any given application to appropriately choose the information to be place in the relational store so that the resulting compiled form achieves both system and performance goals.

4.2.2 Live properties of triples

Another interpretation is to view the attribute store as holding the "live" properties about the contents of the triple store. As described by Whitehead and Goland (Whitehead and Goland 2004) a live property "is one where the server performs a computation associated with setting or retrieving its value". (*Dead* properties have values that are maintained exclusively by the application). Examples of the live properties that might be maintained include the time a triple was added to the store, the source of a triple, and the confidence associated with a triple. The decision of what constitutes a live property suitable for storage in the attribute store depends on the applications using the triple store, but some there are some common characteristics to consider:

1. The property is defined the time the store is created, rather than being defined by an application using the store
2. The value of the property is generally set by the storage management software; if set by an application consistency is maintained by the store
3. Applications are more likely to read the attribute value rather than modify it
4. The attribute is applicable to all triples in the store

An archetypal example of a live property on a file system, for example, is file modification time.

1. We argue here that the characteristics of live properties have ramifications for the design of RDF triple stores. In particular
2. Live properties should not be set by applications except through predefined mechanism provided by the storage software
3. Live properties cannot be modified or removed from the store by applications except through the store's predefined mechanism
4. Live properties should be maintained for all triples in the store
5. RDF Applications should be able to be written in aware that ignores the live properties, since the properties are specific to a particular type of store

4.3. Implementation

The knowledge base will be implemented by a Sesame data store and an associated relational datastore. In consideration of existing RDF APIs and other components of the larger effort this work supports, the API will be written in Java. The API will tie into a Key Generation service that will generate unique URIs to serve as statement ids. The interface between the API and the RDB tables will be via stored procedures. Stored procedures are precompiled and offer a level of abstraction between the API and the underlying schema. We anticipate the interfaces between the API and the RDB stored procedure to be stable, while the stored procedure and underlying schema change as necessary for performance and tuning. In order to simplify distribution we are initially targeting an embedded RDBMS such as Apache Derby, but if more functionality is required MySQL or PostgreSQL might be considered. The RDF store will initially use Sesame's native storage methods.

The API will consist of two components, one that handles hand shaking across the hybrid stores when a SPARQL query is initiated. The second component exposes the internals of the storage mechanism for development of administrative tools such as bulk loading tools, garbage collection facilities to clean up dangling references, and reporting tools.

4.3.1 Annotating statements using 'quads' in the Sesame triple store

The statement id for each statement will be stored along with the statement as its "context" in the Sesame RDF store. The identifier will be a unique valid URI associated with the triple store. The identifier will be system generated.

This implementation will require the use of a quad-store; the application will be dependent on the use of Sesame or similar triple-store rather than a triple store that doesn't support quads. As discussed above this approach will also function with non-quad stores, but the necessary support will not be included in this implementation.

4.3.2 Storing additional information about statements in the relational store

Predefined attributes about statements will be stored in a relational store. These attributes can be used to group statements according to their provenance, the time they were generated, and the level of confidence in the statements, for example. Since all statements for a given application will have the same set of possible attributes, one table can be used to hold metadata for all statements; the statement id is used as the key for the table. Missing values are allowed. Evolution of the attributes is supported largely through additional columns in the relational store with all values for existing rows assumed to be missing/blank.

4.3.3 Filtering triples for an application

The API will accept a specification of a filter to apply to metadata and return an RDF graph that represents only triples that match the specified filter. One method to specify the filter would be a SPARQL GRAPH query that treats annotations as properties of reified statements; only statements that match the query would be returned (but the statements themselves, rather than their reifications, would be in the returned graph).

4.3.4 Accessing relational store

The statement id can be accessed from the Sesame store through the GRAPH syntax of SPARQL (Figure 1). The statement id can then be passed to the relational API to access metadata about the statement.

```
SELECT ?src ?bobNick

FROM NAMED <http://example.org/foaf/aliceFoaf>

FROM NAMED <http://example.org/foaf/bobFoaf>

WHERE

                          {

    GRAPH ?src
```

Figure 1 GRAPH syntax in SPARQL (Prud'hommeaux and Seaborne 2005)

The knowledge base API provides a method that, when presented with the statement id URI, returns the metadata associated with the statement. Again, one representation for the metadata would be as triples with the statement-id as the subject and the metadata properties and values.

4.3.5 Generating an RDF representation of statement annotations

If completely standard RDF is required, perhaps for interaction with another system, then RDF statement reification would be generated for stored triple information. The unique statement id can be used as the URI of an RDF Statement resource, and each annotated value can be represented as a property (using the RDF property associated with the annotation) of the reified statement. These could be added to the triple store, but more likely would be generated as needed unless the reified statements where accessed often. This could be one of the representations of metadata returned by the knowledge base API; with this mechanism a RDF-compliant version of the metadata can be generated.

5. Other approaches

5.1.1 Reified RDF

The standard RDF method for annotating statements is to create a new resource with type rdf:Statement. The subject, object, and predicate properties of this resource are analogs of the 3 elements of the annotated triple. The advantages of this approach are:

1. You can say things about statements without asserting the statement.
2. It's pure RDF, and will be supported by any RDF engine
3. It supports multiple levels of annotation – you can annotate the annotation of a statement.

The disadvantage is a linear increase in the number of statements in the triple store, which can approach 4n if every one of n statements is reified.

The ability to describe unasserted statements could be extremely important—one needs to be able to state "curveball says that Iraq has wmd" without having the statement "iraq has wmd" being asserted in the knowledge base.

5.1.1.1 Comparison of storage requirements

Assume a total of *N* statements, with A annotation statements about M of these statements. In addition, P of the statements are asserted.

The reification mechanism requires *P* + 3*M* + *A* statements. In the best case, if there are no annotations and all statements are asserted, only N statements are required. The worst case, where every statement is both asserted and annotated, requires *4N + A* statements.

The statement-id mechanism requires *N* + *P* + *A* statements – the statements, an additional statement for each asserted statement, and the annotation statements. The worst case, every statement is asserted requires 2N + A statements.

If most statements are asserted and few statements are annotated, the reification mechanism is viable. If few statements are asserted and most statements are annotated, then the statement-id mechanism might be better. For the TAI, we foresee that most statements will have one or more annotations, so we settled on the statement-id mechanism.

5.1.1.2 Relation to OWL 2

Many of the functional requirements are met by the emerging standard for OWL 2 (Motik, Patel-Schneider et al. 2009). In particular, OWL 2's annotation mechanism allows annotations to be attached to statements and ignored by reasoners. However, mapping OWL 2 to RDF implementations results in the generation of reified statements(Patel-Schneider and Motik 2009), which this design avoids.

5.1.2 Relational solution

One could also take the approach used by Oracle's semantic support (Oracle 2008). In this scheme, RDF is implemented by a table within a relational database. A separate table could be provided to contain statement metadata; alternatively, additional columns could be added to the triple-table. This approach seems viable but was rejected because of cost and the lack of tools to support RDF tools such as standard APIs and SPARQL.

5.1.3 RDF/RDB federation systems

Since 1998 when Tim Berners-Lee (Berners-Lee 1998) articulated the possible link between the semantic web and Entity Relationships, industry and research communities have been exploring linkages between RDF and RDB. From our initial review, a preponderance of the commercial and open source efforts, including D2RQ (Bizer 2007), Virtuoso (Blakeley 2007), DartGrid (Chen, Wu et al. 2006), SPASQL (Prud'hommeaux), and SquirrelRDF (Steer), focus on either the federation or abstraction of the two existing databases from a motivation to integrate. Whether the goal is stated as layering semantic information on existing relational data stores or as integrating disparate sources of information, the approaches focus on making productive, coordinated use of existing but separate sources.

The obvious distinction between our research and these efforts is that we are developing a specialized RDB auxiliary component through the convention of a hybrid to optimize and scale the RDF store. The relational store may be viewed as an index or compilation of data used to make certain types of access more effective. The RDB has no purpose or use outside the augmentation of information in the triple-store.

Despite this distinct difference there have been at least one W3C working group (Malhotra 2009), and different workshops (Prud'hommeaux 2007) studying best practice integration of the two stores and there have been many lessons learned that we may exploit as guidelines in our development. All of these solutions bear monitoring to assess if they develop into a more general solution to the problems we address with our hybrid store.

5.1.4 Key generation

As mentioned in the design, a key generation service will be called to build an identifier to handle coordination between the two stores. Other approaches that have been examined include creating a URI derived from the contents of the triple or using a blank-node identifier generated by the triple store

5.1.4.1 Construction from statement contents

A Statement can be represented with a REST (Fielding 2000) type URI that specifies the content of the statement. For example, assuming that 'triple:' is the prefix for a namespace that is used for this purpose, the statement

http://ns1#t http://ns2#p http://ns1:e

might be represented (using URL escaping) as

triple:http%3A%2F%2Fns1%23t/http%3A%2F%2Fns2%23p/http
%3A%2F%2Fns1%23e

Note that ways of dealing with datatype properties and blank nodes would have to be determined; this style was rejected because of these complexities.

5.1.4.2 Blank node identifiers

In the Sesame triple store, blank node ids could be generated for the context of each statement. The blank node ids will be guaranteed to be unique; however there are a couple of drawbacks:

4. If you extracted the triple store to a serialization and then tried to recreated it, the recreation may have different values for the blank node identifiers for the triples
5. In OWL-DL, blank nodes cannot be used as the object of two different statements

6. References

Berners-Lee. (1998). "What the Semantic Web can represent." 2009.

Bizer, C. (2007, September 8, 2007). "D2RQ - Lessons Learned." Retrieved February 27, 2009, from http://www.w3.org/2007/03/RdfRDB/papers/d2rq-positionpaper/.

Blakeley, C. (2007). "Mapping Relational Data to RDF with Virtuoso's RDF Views." Open-Link Software.

Chen, H., Z. Wu, et al. (2006). "DartGrid: a semantic infrastructure for building database Grid applications." Concurrency and Computation: Practice and Experience **18**(14).

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. Information and Computer Science. Irvine, CA, University of California, Irvine. **Ph.D.**

HP Labs Semantic Web Research. (2004). "Jena Home Page." from http://jena.sourceforge.net/.
Lee, R. (2004). Scalability Report on Triple Store Applications. Cambridge, Mass., MIT.
Malhotra, A. (2009). "W3C Incubator Group Report 26 January 2009." from http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/#StateOfArt.
Motik, B., P. F. Patel-Schneider, et al. (2009). OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax, World Wide Web Consortium.
openRDF.org (2008). OpenRDF.org, home of sesame, OpenRDF.org.
Oracle. (2008). "Semantic Technologies Center."   Retrieved 2/27/2009, 2009.
Patel-Schneider, P. F. and B. Motik (2009). OWL 2 Web Ontology Language: Mapping to RDF Graphs, World Wide Web Consortium.
Prud'hommeaux, E. (2007, 2007). "W3C Workshop on RDF Access to Relational Databases."
Prud'hommeaux, E. and A. Seaborne (2005). SPARQL Query Language for RDF, World Wide Web Consortium.
Prud'hommeaux, E. SPASQL: SPARQL Support In MySQL.
Steer, D. SquirrelRDF-Querying existing SQL data with SPARQL.
Whitehead, E. J. J. and Y. Y. Goland (2004). "The WebDAV property design." Software: Practice and Experience **34**(2): 135-161.