Transaction-Time Queries in Dydra

James Anderson and Arto Bendiken

Datagraph GmbH

Abstract Dydra is an RDF graph storage service. It stores and retrieves the contents of RDF datasets through SPARQL, LDF and LDP interfaces. In addition to these basic capabilities, it retains previous store states, in addition to the current state, as active addressable aspects of a dataset analogous to named graphs in a quad store. It incorporates arbitrary revisions into target datasets according to query arguments for HTTP requests and an additional REVISION clause in SPARQL.

This document describes a taxonomy of archival RDF queries and illustrates it with examples drawn from three popular ontologies: gist, schema.org and STW, which demonstrate how the Dydra TB storage architecture combines with simple interface extensions to support the principal tasks and address the primary concerns when working with RDF data over time.

Keywords: RDF, temporal data, SPARQL, revisions

1 Introduction

Dydra is an RDF graph storage service. It operates as a cloud service, a local service or an embedded library. It stores and retrieves the contents of RDF datasets through SPARQL, LDF and LDP interfaces. In addition to these basic capabilities, Dydra retains previous store states, in addition to the current state, as active addressable aspects of a dataset analogous to named graphs in a quad store. It addresses these states in its REST interfaces through the values supplied for a `revision` argument, which acts in a manner analogous to the `graph` argument for a quad store request, or through the `Accept-Datetime` header defined by Memento. Its SPARQL dialect includes a REVISION clause which plays a role for revisions analogous to that which the GRAPH clause plays with respect to named graphs. These facilities suffice to manage and analyze evolving datasets over time. In order to demonstrate this, we present a taxonomy of archival RDF analytics, describe its relation to the BEAR framework[1] for benchmarks for RDF archives, and illustrate individual concrete cases with SPARQL queries.

The next section introduces a taxonomy with which to comprehend the possible query forms, Section 3 aligns this taxonomy with that from the BEAR proposal and illustrates each case with a simple query. Section 4 provides extended examples. Section 5 discusses implementation considerations.

2 A Taxonomy for Archival RDF Analysis

We characterize queries, for the purpose of this discussion, according to two principle dimensions: dataset constitution and algebra combination. *Constitution* concerns which revisions to include in the target dataset and how to address them. *Combination* concerns how the query algebra combines those constituent elements. An a-temporal query specifies a target RDF dataset with respect to named graphs by indicating which graphs are to be merged into the target dataset default graph and/or which are to be made available as named graphs. When a variable is specified in a `GRAPH` clause, it ranges over the specified set or, when none was specified, over a default set. In order to perform inter-graph comparisons, a query includes multiple `GRAPH` clauses and combines the respective results through arbitrary SPARQL[5] algebra operations. In this case, the target dataset constitutes a collection of these graphs, they are addressed by respective IRI and the solutions are combined with or without extensions to bind the graph depending on whether the respective clause took the default graph, a constant named graph, or the domain of an an abstract graph variable as the target.

In a transaction-time query, revisions play a role analogous to graphs. The query can specify one or more revisions to indicate the transaction state(s) which constitute(s) the target dataset. If none is specified, as a default, the dataset reflects the latest revision. Any revision variable ranges over known revisions. A revision variable extends solutions within their scope with a binding for its value, which contributes to algebra operations in the same manner as any other binding. In contrast to graphs, however, in addition to specifying an individual revision, a revision designator can compose revisions, for example, to incorporate all states over a temporal interval, or to indicate the difference between the states which correspond to transactions. The query algebra then matches graph patterns against the composed revision datasets and combines them to produce the results.

In terms of dataset constitution and algebra combination, descriptions of transaction-time queries supports the following characterisations:[1]

- dataset revision constitution : none ($\varnothing$), single (@), multiple (n), ranges ($\cdots$), or differences (Δ)
- algebraic combination : default ($\varnothing$), constant (V_i)[2], or abstract ($?v$) .

The revision designators provide means to constitute datasets corresponding to various temporal entities:

- A single revision is identified directly by it UUID, for example `58bd5ff7-7d46-48f8-b64a-43f257c48817`, or by a timestamp in the interval in

[1] This exposition concerns neither streaming data nor graph store operations. Any use cases related to streaming data will still require an individual query reduction to occur on a static dataset, but may require additional means to refer to transaction when constituting the dataset. Use cases related to graph store operations are always identity projections of a composed dataset.

[2] A *constant* corresponds to a date, a revision name, or the revision associated with some user label.

Table 1. Query taxonomy and correspondence to the BEAR framework.

Constitution \ Combination	none (∅)	single (@)	multiple (n)	range (···)	difference (Δ)
default (∅)	→ single	$Mat(Q,$**HEAD**$)$	-	-	-
constant (V_i)	-	$Mat(Q, V_i)$ $Join(Q_1, V_i, Q_2, V_j)$	-		-
abstract (?v)	$Ver(Q)$	$Diff(Q, v_i, v_j)$ $Change(Q)$	-	-	-

which the revision was current for its repository, in that case `2016-03-15T01:11:38Z`.
– A relative revision is designated by inflection, for example `58bd5ff7-7d46-48f8-b64a-43f257c48817` The compositon of two revision is designated by their sequence, for example `58bd5ff7-7d46-48f8-b64a-43f257c48817,f47ac10b-58cc-4372-a567-0e02b2c3d479`.
– The additions and deletions between two revision is designated by connecting identifiers for the bound with "..", for example `58bd5ff7-7d46-48f8-b64a-43f257c48817..f47ac10b-58cc-4372-a567-0e02b2c3d479`.

The combined characterisations yield the taxonomy shown in Table 1. In these terms, all query variations present in the BEAR framework are accommodated in four of the twelve combinations, as indicated in the *none* and *single* columns. In particular, a mechanism which provides just a request revision specification analogous to the SPARQL `graph` clause is sufficient. For more complex use cases, the constitution forms *multiple*, *range*, and *difference* compose a basic graph match target a dataset which comprises distinct revisions, but is processed as a single entity. This supports diachronic use cases, such as

– Retrieve those concepts changed during a given calendar interval.
– Compute those ontology items which contradict the state recorded in the previous revision.
– Retrieve those concepts which are universally valid across the entire repository lifetime.

Once the distinction has been made between dataset constitution and algebraic composition, in order to extend SPARQL to query revisioned datasets, is it necessary only to provide means establish the scope of a given revision and, where the value is not constant, to bind it to a variable. This is accomplished with a REVISION clause, analogous to a GRAPH or SERVICE clause. Where the latter limit the application of contained patterns to composed local graphs and or to a graph at a remote location, the REVISION clause limits the application to composed versions. Where alternative approaches, suggest to conflate revision metamodel with the domain model either by reifying the domain data in order to store it in the revision model[4], by using named graphs to associate revision

```
[[56]] GraphPatternNotTriples ::=
    OptionalGraphPattern | GroupOrUnionGraphPattern |
    MinusGraphPattern | GraphGraphPattern |
    RevisionGraphPattern | ServiceGraphPattern
[[60a]] RevisionGraphPattern ::=
    'REVISION' (VarOrIRIref | String) GroupGraphPattern
```

Figure 1. SPARQL grammar REVISION extension

transaction information with domain data[6], or by extending the domain model to include revision transaction attributes[2], the goal of this approach is to abstract the revision model from the domain model and facilitate an implementation which is at once simpler and more flexible. Rather than extend the data model, temporal attributes are factored out to a provenance repository, where the service maintains transaction time information, the application can augment the provenance records as necessary, and federated queries compose revisions as required.

The implementation extends one grammar production, as indicated in figure 1[3], to establish the scope of a revision designator and permit its binding. If the value is constant, the target is that revision. If the value is a bound variable, the query applies to all revision values apparent for that variable in the respective solutions. If the variable is free, the revision ranges over all repository revisions, in reverse chronological order.

3 BEAR Comparison

Each element from the BEAR blueprint for temporal RDF analytics is realized as in Table 3. In Dydra, the semantics diverge from that proposed by [1], in that a temporal annotation takes the same form as that for named graphs: a variable binding. That means the annotations are present in solutions and, as such, figure in any compatibility computation. Under this semantics, any join and aggregation operations must account for the binding. Table 3 contains the SPARQL query which implements each BEAR case in Dydra.

4 Examples

In order to illustrate how the facility applies to concrete cases, we present examples for archival analysis of ontology datasets. One is drawn from each of three popular, evolving ontologies: gist schema.org, Standard Thesaurus for Economics, and Each alternative is illustrated below with a case related to ontology curation.

4.1 gist

Use the provenance records to determine the revisions current at given dates and analyse the ontology state for each.

[3] See http://www.w3.org/TR/2013/REC-sparql11-query-20130321/ as available on 2016-04-25 .

```
SELECT  ?concept (count(?subConcept) as ?frequency)
                 (sample(?releaseDate) as ?date)
                 (sample(?label) as ?release)
WHERE {
 { SERVICE <http://localhost/schema/gist-provenance> {
   { SELECT ?releaseDate (max (?revisionDate) as ?releaseRevisionDate)
     WHERE {
       VALUES ?releaseDate {
         '2009-02-28T12:00:00Z'^^<http://www.w3.org/2001/XMLSchema#dateTime>
         '2009-07-31T12:00:00Z'^^<http://www.w3.org/2001/XMLSchema#dateTime>
       }
       GRAPH ?revision {
        ?revision <http://www.w3.org/ns/prov#generatedAtTime> ?revisionDate .
       }
       FILTER (?revisionDate <= ?releaseDate)
     } GROUP by ?releaseDate
  } { GRAPH ?revision {
    ?revision rdfs:label ?label .
    ?revision <http://www.w3.org/ns/prov#generatedAtTime> ?releaseRevisionDate .
   } }
 } }
 REVISION ?revision {
  ?subConcept rdfs:subClassOf ?concept .
 }
}
GROUP BY ?concept ?revision
ORDER BY DESC (?frequency)
LIMIT 10
```

4.2 STW

Indicate the prevalence of descriptors across thesaurus revisions.

```
prefix  skos:   <http://www.w3.org/2004/02/skos/core#>
prefix  zbwext: <http://zbw.eu/namespaces/zbw-extensions/>
#
# Show the number of versions in which a descriptor is present
#
select ?prevalence (count(?prevalence) as ?frequency)
where {
  select ?s (count(?r) as ?prevalence)
  where {
    revision ?r { ?s a zbwext:Descriptor . }
  } group by ?s
}
group by ?prevalence
order by desc (?frequency)
```

4.3 schema.org

Indicate the prevalence of classes which have been marked as deprecated.

```
select ?concept ?revisionDeprecated ?revisionUsed
where {
  { select ?concept ?rDeprecated where {
    revision ?rDeprecated {
      ?concept <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
      filter (regex(?comment, '.*deprecated.*')) } } }
  { revision ?rUsed {
      ?concept a ?type .
```

```
    } }
  { service <http://localhost/schema-org-test/provenance> {
      graph ?rUsed { ?rUsed rdfs:label ?revisionUsed }
    } }
  { service <http://localhost/schema-org-test/provenance> {
      graph ?rDeprecated { ?rDeprecated rdfs:label ?revisionDeprecated }
    } }
} order by ?concept
```

5 Implementation Considerations

The store implementation has been designed for efficient mutation at scale balanced with the requirement to access historical revisions of data. It combines graph-partitioned triple tables with clustered, persistent B+tree indexes, based on a memory-mapped MVCC design with full ACID semantics. By default, repository data is comprehensively indexed six ways: GSPO, GPOS, GOSP, SPOG, POSG, OSPG, enabling any quad-pattern match to be answered from indices. RDF terms are interned on an installation-wide basis into integer ordinals. In the storage for a repository, B+tree keys consist of four integers representing the graph, subject, predicate, and object terms. B+tree values store a revision visibility map indicating which revisions a particular quad is visible in.

The revisioning can be disabled in a per-repository basis in which case B+tree values are of zero length; further, the trade-off between mutation performance versus query performance can be tuned by configuring the revision visibility map to be used only on the GSPO index, which speeds up mutation about six-fold at the constant cost of a factor two increase in B+tree lookups during query processing of non-GSPO patterns.

There are various encodings of revision visibility maps as succinct data structures that optimize for efficient revision lookup and compact space utilization. The base storage requirements for an un-versioned repository involve sixteen or thirty-two bytes per statement, depending on intended capacity, times the index count plus storage for term strings. With revisions, the space should increase in a sublinear relation to mutation count, where those statements not modified since first insertion require no additional space, while mutated quads require a visibility map, the size of which depends on the mutation pattern.

Table 2 compares the space for RDF document and indexed representations of the example datasets. In addition to the gist, schema.org and STW datasets, the table includes statistics from the revision history of the "Experimental Factor Ontology" [3] in the rows "efo" , "efo @2.67" and "efo unrevisioned". contain the space requirements for the complete revision history and for a single-revision repository which includes the latest 2.69 version only. The results indicate that the representation for revisions adds significant overhead with respect to unique statements, but provides and advantage with respect to total statement count.

As illustrated by the green components in figure 2 [4], the implementation effort to support REVISION clause was limited. The service depends on a

[4] See http://arxiv.org/pdf/1504.01891v2, as available on 2016-04-25.

Table 2. Revision Space Characteristics

ontology	revisions	files (MB)	quads total	quads HEAD	store (MB)	bytes/quad
gist	16	2.9	27,519	849	11.4	434
schema.org	23	19.1	156,146	11,228	22.4	151
STW	7	94.2	771,375	108,967	116.5	158
efo	144	1,384.0	17,614,527	260,503	3,079.0	183 (total)
efo	144	1,384.0	4,221,100	260,503	3,079.0	765 (unique)
efo	@2.69	2.8	260,503	260,503	93.2	375
efo	unrevisioned	2.8	260,367	260,367	93.5	377

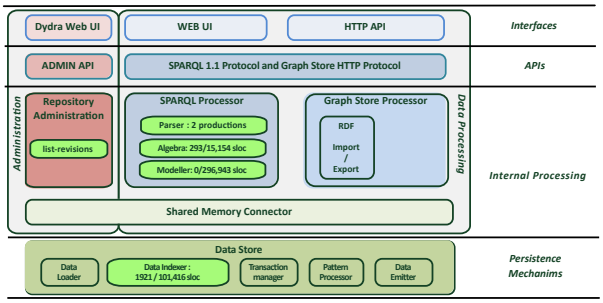

Figure 2. Service Implementation Changes for Revision Support

strict separation between the SPARQL processor and the RDF store. This limits the query processor to operations which manage transactions for specific repository revisions and to perform count, match and scan operations with respect to statement patterns. As a consequence of this interface, the changes to the SPARQL processor are limited to 293 sloc for the algebra operator and the two productions in the grammar in figure 1.

This compares well, for example, to the SERVICE operator, which requires 453 sloc and to the 13,151 sloc for the entire algebra implementation. The store implementation is more substantial. In this case, 1,921 / 101,416 sloc implement the revisioned index, which is still a relatively small amount, considering the capabilities.

References

1. Javier David Fernandez Garcia, Jürgen Umbrich, and Axel Polleres. Bear: Benchmarking the efficiency of rdf archiving. Technical report, Department für Informationsverarbeitung und Prozessmanagement, WU Vienna University of Economics and Business, 2015.
2. Claudio Gutierrez, Carlos A Hurtado, and Alejandro Vaisman. Introducing time into rdf. *Knowledge and Data Engineering, IEEE Transactions on*, 19(2):207–218, 2007.

3. James Malone, Ele Holloway, Tomasz Adamusiak, Misha Kapushesky, Jie Zheng, Nikolay Kolesnikov, Anna Zhukova, Alvis Brazma, and Helen Parkinson. Modeling sample variables with an experimental factor ontology. *Bioinformatics*, 26(8):1112–1118, 2010.
4. Marios Meimaris, George Papastefanatos, Stratis Viglas, Yannis Stavrakas, and Christos Pateritsas. A query language for multi-version data web archives. *arXiv preprint arXiv:1504.01891*, 2015.
5. Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):16, 2009.
6. Jonas Tappolet and Abraham Bernstein. Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In *The Semantic Web: Research and Applications*, pages 308–322. Springer, 2009.

Table 3. Dydra - BEAR alignment

Dydra SPARQL forms corresponding to BEAR Abstract Notation		
Version materialisation	Mat(Q,vi)	SELECT * WHERE { Q :[vi] }

	(@.V_i)	SELECT * WHERE {
		  REVISION <urn:uuid:12345678-····-123456789012> {
		  ?s ?p ?o
		  }
		}

Delta materialisation	Diff(Q,vi,vj)	SELECT * WHERE { { { {Q :[vi]} MINUS {Q :[vj]} } BIND(vi AS?V) } UNION { { {Q :[vj] } MINUS {Q :[vi]}} BIND(vi AS?V) }

	(@.?v)	SELECT * WHERE {
		  REVISION ?v {
		  { {?s ?p ?o} MINUS {REVISION "~" {?s ?p ?o}}}
		  UNION
		  { { REVISION "~" {?s ?p ?o}} MINUS {?s ?p ?o}}
		  }
		}

Version Query	Ver(Q)	SELECT * WHERE { P :?V }

	(∅.?v)	SELECT * WHERE {
		  REVISION ?v {
		  ?s ?p ?o
		  }
		}

Cross-version join	join(Q1,vi,Q2,vj)	SELECT * WHERE { {Q :[vi]} {Q :[vj]} }

	(@.V_i)	SELECT * WHERE {
		  {REVISION <urn:uuid:12345678-····-123456789012> {
		  {?s ?p ?o} }
		  {REVISION <urn:uuid:87654321-····-098765432109> {
		  {?s ?p ?o} }
		}

Change materialisation	Change(Q)	SELECT ?V1 ?V2 WHERE {
		  {{P :?V1 } MINUS {P :?V2}}
		  FILTER(abs(?V1-?V2) = 1)
		}

	(@.?v)	SELECT ?v WHERE {
		  REVISION ?v {
		  {?s ?p ?o}
		  MINUS
		  { REVISION '~' {?s ?p ?o} }
		  }
		}