A Taxonomy for Transaction-Time Queries with Examples from 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 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[4] 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[2] : none ($\varnothing$), single (@), multiple (n), ranges ($\cdots$), or differences (Δ)

[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] Revison designators take various forms:

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

– algebraic combination : default ($\varnothing$), constant (V_i)[3], or abstract ($?v$) .

This characterisation yields the following taxonomy, with the indicated correspondence to the BEAR framework.

constitution \ combination	none ($\varnothing$)	single (@)	multiple (n)	range ($\cdots$)	difference (Δ)
default ($\varnothing$)	$\rightarrow$	$Mat(Q,\mathbf{HEAD})$			
constant (V_i)		$Mat(Q,V_i)$ $Join(Q_1,V_i,Q_2,V_j)$			
abstract ($?v$)	$Ver(Q)$	$Diff(Q,v_i,v_j)$ $Change(Q)$			

The correlation to the BEAR taxonomy indicates that its proposed variations can all be accommodated by a mechanism which supports just request specification analogous to the `graph` SPARQL protocol. For more complex use cases, where the basic graph match must target a dataset which is comprises distinct revisions, but is processed as a single entity, constitution form *multiple*, *range*, *difference* are necessary to compose the dataset:

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

3 BEAR Comparison

Each element from the BEAR blueprint for temporal RDF analytics is realized as in the table in figure 2. 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 2 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,

- 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`.

[3] A *constant* corresponds to a date, a revision name, or the revision associated with some other identifier.

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.

```
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 unversioned 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. Figure 1 compares the space for n-quads and indexed representations of the example datasets.

References

1. Fernandez Garcia, J.D., Umbrich, J., Polleres, A.: Bear: Benchmarking the efficiency of rdf archiving. Tech. rep., Department für Informationsverarbeitung und Prozessmanagement, WU Vienna University of Economics and Business (2015)
2. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing time into rdf. Knowledge and Data Engineering, IEEE Transactions on 19(2), 207–218 (2007)
3. Lopes, N., Lukácsy, G., Polleres, A., Straccia, U., Zimmermann, A.: A general framework for representing, reasoning and querying with annotated semantic web data. Tech. rep., Technical report, DERI (2010)
4. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. ACM Transactions on Database Systems (TODS) 34(3), 16 (2009)
5. Tappolet, J., Bernstein, A.: Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In: The Semantic Web: Research and Applications, pp. 308–322. Springer (2009)
6. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Van de Walle, R.: R&wbase: git for triples. In: LDOW (2013)
7. Zimmermann, A., Lopes, N., Polleres, A., Straccia, U.: A general framework for representing, reasoning and querying with annotated semantic web data. Web Semantics: Science, Services and Agents on the World Wide Web 11, 72–95 (2012)

Figure 1. Space Characeristics

ontology	files	quads	space (MB)	bytes/quad
gist	2941371	47299	14.2	291
schema.org	19129181	156150	40.6	260
STW	94225471	771375	240	311

Figure 2. 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} }
		  }
		}