SPARQL Statement Annotations for Temporal Metadata in the Dydra RDF Store*

James A. Anderson[1,*,†], Vimal Kumar[2]

[1]datagraph gmbh, frankfurter tor 1, 10243 berlin, germany
[2]NXP Semiconductors N.V.

Abstract

Revisioned RDF datasets associate transaction-time metadata with statements. Applications use this metadata to constrain query results. Cases which involve dataset snapshots, windows, or streams need apply just global temporal constraints. Applications which embody business logic which interprets statement-level metadata require finer-grained access.

This report describes the application of revisioned storage capabilities in the Dydra RDF storage service to record and report version information in a product life-cycle management (PLM) system at graph node granularity. It describes the current nature of revisioned RDF repositories managed by the service for typical application queries and focuses on one PLM application and its data model for which transaction-time metadata reveal important management information. It discusses an extension to SPARQL to use RDF Star annotation syntax to associate temporal metadata with this application data model, reviews a realization of that approach for the PLM application, and discusses performance issues.

Keywords

RDF, RDF Star, SPARQL, temporal annotation,

1. Introduction

This report describes how revisioned storage capabilities of Dydra RDF storage service have been applied to record and report modifications to semiconductor device specifications in a product life-cycle management (PLM) system. First, it reviews Dydra architecture and relates the approach to alternatives in temporal RDF data management. Then it describes the storage characteristics of the revisioned RDF repositories as they relate to the model of the managed PLM data and illustrates the effect of revisioned storage on retrospective access execution time based on the execution statistics for a number of application queries. Finally it introduces an extension to SPARQL to use RDF Star annotation syntax to permits a PLM application to use SPARQL queries to reveal important product management information.

The exposition includes detailed statistics about the application's production datasets to call attention to their notable characteristics. It reiterates aspects of standard benchmarks and compares the results between well known datasets and those in production to demonstrate

MEPDaw 2023, Athens, Greece

* This reports on work with the service at version e48262281df808888ef66333ef2d7e62eb2cc1a6@20220726T161540

james@dydra.com (J. A. Anderson); vimal.kumartv@nxp.com (V. Kumar)
https://dydra.com/ (J. A. Anderson)
0000-0002-1461-6338 (J. A. Anderson)

that the architectures accommodates a wide range of data models. It presents aspects of the implementation in detail, to highlight space and time tradeoffs which are not yet governed by abstract models for archival RDF data.

2. Dydra Service Architecture

2.1. Implementation

The Dydra store architecture has been described in detail in reports to previous MEPDaw workshops[1][2]. It has also been described in surveys of RDF stores – with the most accurate being Ali's survey[3]. The store provides access to transaction-time timestamp-based revisioned RDF storage through W3C Graph Store and SPARQL protocols, with extensions to target and retrieve dataset content from other than the latest revision. Some mention of the service has also been included in expositions of alternative approaches to versioned data, such as [4], [5] and [6], but in order to resolve ambiguities which these publications have introduced regarding its architecture and behaviour, we first review the store's architecture and operation in order to resolve ambiguities left by these descriptions.

Dydra provides a multi-tenant service in which users create RDF repositories and operate on them through HTTP requests in the manner of the W3C SPARQL and Graph Store protocols. That service layer relies on a repository storage layer built with LMDB[7] B-tree databases to provide a term dictionary, the standard six quad term identifier indices - in this case gspo, gpos, gosp, spog, posg, and ospg, and indices for revision metadata among timestamps, UUID designators, and ordinals, Terms are stored in a global index among term values, their ordinal identifiers and their hash codes. Each B-tree key represents a quad as the respective term identifiers. The particular feature which supports transaction-time metadata is the interpretation of quad index records as transaction indices. As illustrated in figure 1, the index records for a stand-alone revisioned repository contain a sequence of revision ordinals to identify those repository transactions which added and removed that statement over time. Index entries for non-revisioned repositories contain no record data and the records for replicated repositories contain global revision identifiers rather than repository-relative ordinals.

All access operations devolve to just three principal transaction-scoped operations: match, count, and scan. All processing is within a request transaction, which captures its request's revision constraints. In order for a request to address a dataset snapshot, window, or stream, it designates one or more target transactions with timestamps, ordinals or UUIDs. As the query processor interprets the statement patterns in a basic graph pattern (BGP), it resolves any constraints present in the transaction which governs that BGP to ordinal intervals. During a count or scan phase, the revision record of each matched quad is inspected and only those which satisfy the constraints are further processed. This limits the stream of BGP solution results to those which satisfy the request's revision constraints.

This arrangement implements the transaction-time aspect of a bi-temporal store. It provides efficient storage for minimal transaction-time metadata and constitutes one dimension of Grandi's multi-temporal dimensions[8] . In order to capture the validity-time aspect of higher-dimensional temporal constraints in a form which supports efficient retrieval, rather than introduce additional domains into the statement revision indices, it augments the standard

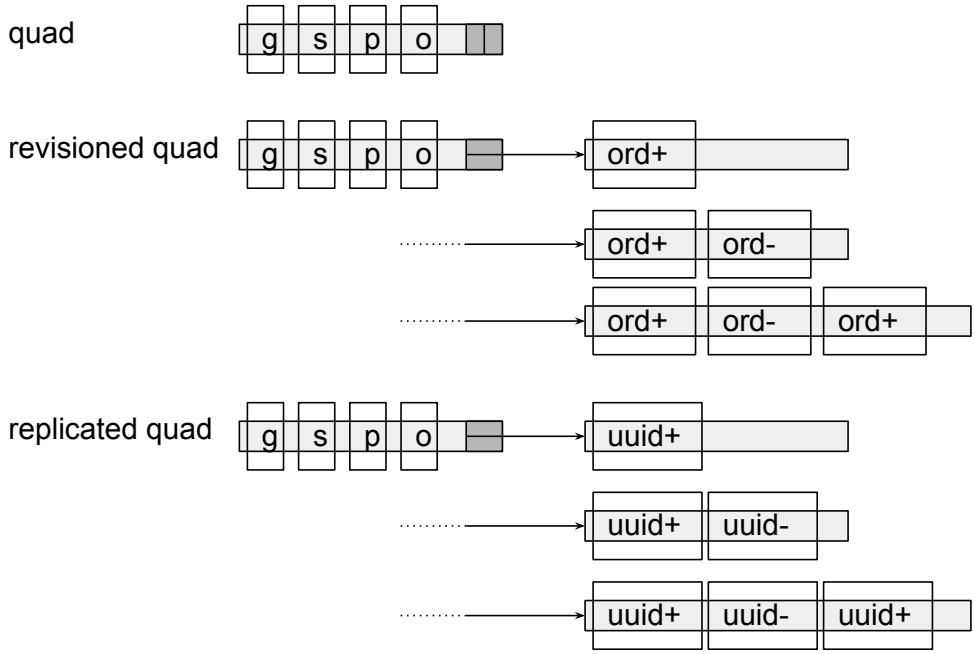

Figure 1: Quad index record variants for static, revisioned, and replicated repositories

quad indices with variants which store and index timestamps for declared predicates in their respective value domains instead of as term identifiers. The two index forms are independent and combine to support bi-temporal models. If an account maintains a provenance repository, a record is generated for each transaction to capture temporal attributes and information about the request agent. Its transaction metadata can be used to derive revision constraints from request arguments and/or to augment result bindings for matched statements.

As a strict TB variant[9], this architecture provides the benefit, that revision information is handled only when a statement is relevant, and is stored in proximity to the respective quad data as it is the index record respective the quad index key. When solutions are constructed, only the relevant data must be marshalled from persistent storage, examined, and collected. This distinction as to dependency and locality is important, but not adequately characterised by abstract RDF data models. In one instance Pelgrin[10], proposed that the Dydra architecture retained revision metadata at the dataset level. Despite that this aspect of the metadata is secondary, the notion persists in later works where, for example Kovacevic[4] emphasises the significance of dataset-level metadata, while neglecting the relation between metadata and the individual statements.

An alternative to TB architectures is the "independent copy" (IC) approach. The BEAR project

itself implemented this variant based in named graphs in Jena in order to test its behaviour[9]. Cuevas[11] also proposes an IC variant which relies on graphs to segment the data into revisions in order to investigate the consequence for index size and query formulation relative to CB and TB variants. Independent of its variants, the approach presents two deficiencies. On one hand, in an application environment where a given PLM repository at one time contained 56,216,953 active quads with 5,377,260 distinct subjects in 1,547,220 graphs, to segment revisions with graphs would conflict with the application data models, as they rely on graphs to segment the device specifications. On the other, an IC architecture which materialized quad datasets which correspond to these PLM models, would set prohibitive space requirements. For example, the same PLM repository, which occupied 67,566,194,688 bytes in the current architecture, would require 1,787,570,499,040 bytes when stored as independent copies of the 33,567 revisions present. (see table 3)

The other principal alternatives to TB architectures are found in change-based (CB) approaches, which seek to minimize the storage requirements by retaining just the changes which effect each revision. Of these, the principal developments have followed from Taelman's research programm[5][6], which endeavours to accommodate large rapidly evolving datasets by trading trade space resources for time by decreasing the interval between snapshots. The latest variant[12] intends to address the misfortune, that "existing approaches for RDF archiving cannot ingest long histories on large datasets" by improving the delta computation process and making its representation more effective. The results demonstrate, however, that in order to achieve acceptable ingestion times, the representation consumes significant space. By contrast, as indicated in table 1, the results for ingestion of the BEAR-B instance dataset demonstrates that a TB architecture accomplishes a reasonable ingestion rate, while producing a much more compact dataset and, in connection with the smaller BEAR-B hour dataset, figure 2 indicates that the performance of the Dydra TB variant approached that of the HDT-IC implementation on the materialised access pattern.[1]

repository	archival statements	revisions	size(MB)	ingestion (min)
Dydra	234764	21045	88.72	86.91
OSTRICH+ high periodicity	234,588	21045	2283.43	57.89
OSTRICH+ low periodicity	234,588	21045	787.75	298.36

Table 1
BEAR-B instance dataset statistics comparing Dydra with an improved OSTRICH variant[12]

This temporal storage architecture manifests the following principles

- The BGP solution stream generation process contrasts to approaches which materialize the set of temporally constrained statements - whether explicitly or implicitly, and only then constrain them with a statement pattern. The BGP processor first applies the pattern to the dataset and then interprets temporal constraints. The average execution statistics for the views included below reflect this effect: the count of matched statements is a small fraction of the statements which would be need to be examined in order to generate a materialized version.

[1]This graph of the BEAR-B performance is drawn of the results presented at the MEPDaw-2019 workshop, which covered all ingestion and query variants for the hour dataset.[2]

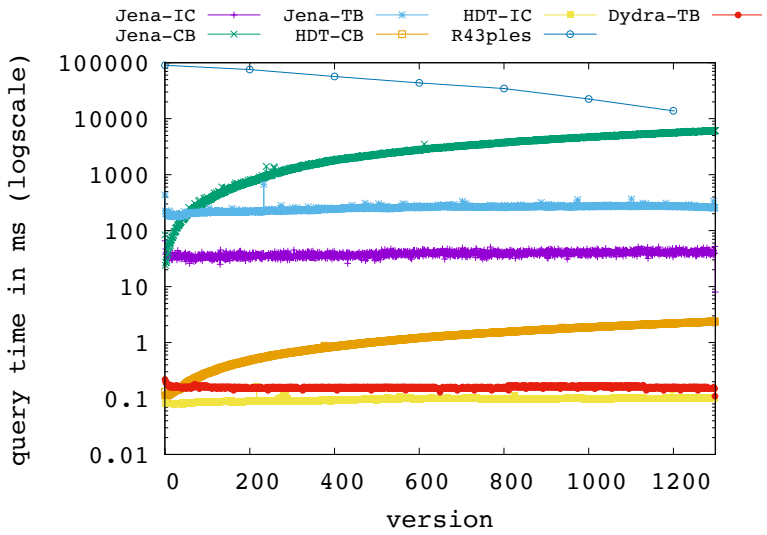

Figure 2: BEAR-B materialised query performance over the hour dataset revision history

repository	view	matchCount	statementCount
nxp/plm__rev	get_list_changeditemname_for_eco	1619419	54085015
nxp-hws/plm__rev	sfdc_salesiteminfo	18627813	75486524
nxp/plm__rev	fetch_parent_for_the_Item	218297	54085015
nxp/plm__rev	ebiz_article_group	3012268	54085015

Table 2
BGP match count v/s repository statement count for typical views

- The transaction-time dimension is represented with revision ordinals which are specific to the repository history and internal to the data management service.
  They are not immediately available data as dictionary-encoded terms to be bound to solution variables for algebra operations.
- The transaction time domain is distinct from the validity time domain. The first is integer ordinals while the latter is RDF temporal values.
  Each is best supported by an index appropriate to its domain.
- The logic to apply transaction constraints is implemented in an efficient value domain, it requires no joins and - for predominant cases, can rely on data proximity.
- Transaction time metadata associates with individual triples, while validity time is associated with subject resources.
  The respective retrieval patterns will entail different index access patterns which suggest different index forms.
- Provenance or other metadata are decoupled from the revision index. Access to an independent provenance repository is mediated through transaction UUID designators.

2.2. API

Operations under this architecture are available to client applications through as simple HTTP API. In order to interpret a repository as revisioned, a request designates the target revision by timestamp, revision identifier, revision ordinal or some combination of those values in a manner similar to the specification of a target graph. The presentation in [1] describes the variants in detail.

Under this API, a request to apply a the view `listCompanies` to a snapshot of a previous revision of the repesitory `SBA/PPP` would take any of the following forms

```
https://dydra.com/SBA/ppp/listCompanies?revision-id=HEAD-1
https://dydra.com/SBA/ppp/listCompanies?revision-id=7a4cff4c-31c6-11ee-9363-f02f7494a8ed
https://dydra.com/SBA/ppp/listCompanies?revision-id=1999-01-01T24:02:03.5Z
```

Application to a revision window or stream for a view such as `totalDisbursements` would take a form which designates revisions according to ISO time intervals or repetitions.

```
https://dydra.com/SBA/PPP/totalDisbursements?\
     revision-id=1999-01-01T24:02:03.5Z--1999-01-01T24:02:03.6Z
https://dydra.com/SBA/PPP/totalDisbursements?\
     revision-id=1999-01-01T24:02:03.5Z/1999-01-01T24:02:03.6Z/P1DT2H3M4S
```

This API applies uniform revision constraints over the entire extent of a transaction. In order to limit their extent, the query must factor distinct constraints into distinct `SERVICE` clauses, each of which is governed by its own transaction.

3. Repository Storage Statistics

In order to address questions as to the efficiency of the chosen Dydra storage architecture and its effects on query performance we consider two sources of information. One is the performance on known benchmark datasets, as described above, which suggests parity with alternatives. Another is its performance with enterprise datasets. For this, we compare access to analogous revisioned and un-revisioned dataset instances to examine the consequence of revisioned storage for access at scale.

The facility has been in service over the past year. Over this interval several repositories have grown to approach 10^8 statements. A review of the storage statistics for these repositories offers some insight into the resource requirements of revisioned storage. Table 3 summarizes these statistics. It depicts the cumulative space, statement count and revision count as well as values for the number of total changes and adds computed values for the size of equivalent IC and CB repositories.[2]

Of note is that, while the absolute per quad space has grown, in most cases, the space per archived quad is just a fraction of that required per quad by an un-revisioned repository. This is the case despite that the quad change ratio[9] is a large multiple of that which is reported for benchmark datasets.[3] The `planning-integration__rev` repository demonstrates, in

[2] In the instance, the CB value reflects just the deltas with no intermediate version materialized.

[3] For example the revisioned form of the production `plm__rev` repository replaced 613% of the statements over thirty-three thousand revisions, while the BEAR-B instant dataset demonstrated a change ratio of just 0.011% over its twenty-one thousand revisions.

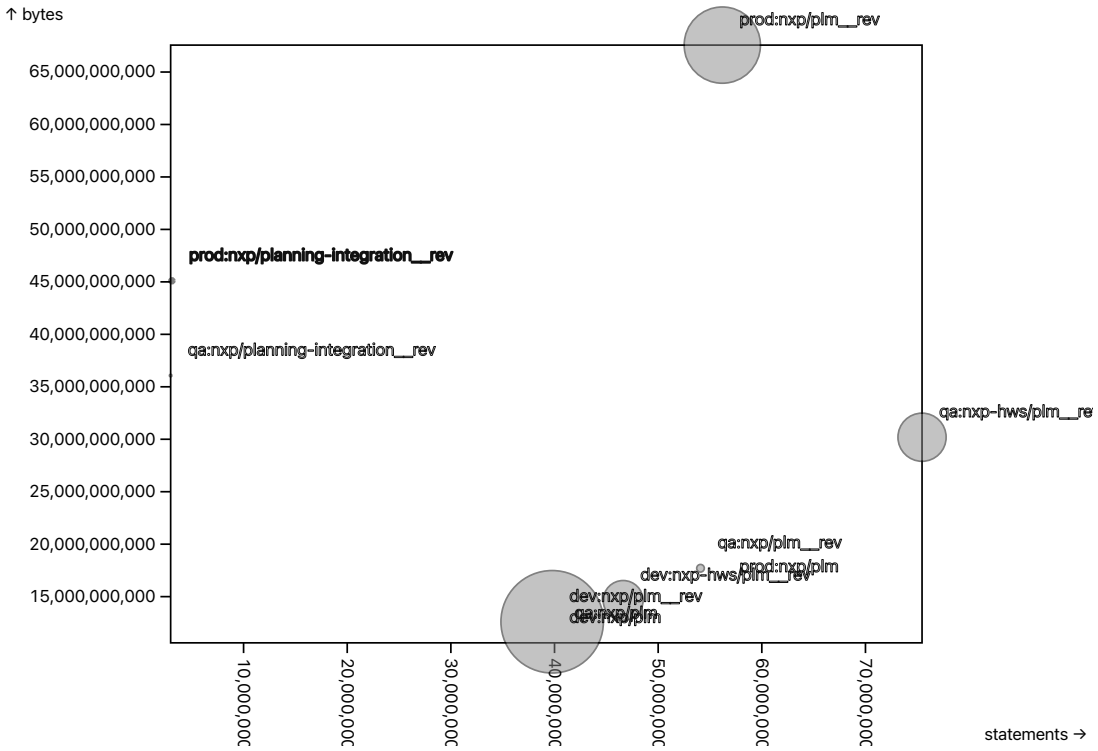

Figure 3: Repository resource usage: size v/s statement count, indicating revision count

contrast, the consequence of churn in a TB architecture: the storage requirements evolve in the direction of IC storage. In this repository, requests over a long interval were Graph Store protocol PUTs with the entire repository as their content.

The relation between size and statement count and revision count is illustrated in figure 3. Their projection in parallel coordinates form in figure 4 suggests that there is no clear correlation.

The statistics demonstrate that,

- in most cases, storage grows linearly with revision count, with a factor below 1.0, but particular update patterns can significantly increase that factor.
- in most cases, each individual revision record involves a very small percentage of the revisions, while n those cases where the percentage is high, storage requirements increase.
- in the limited cases where repositories exist in both static and revisioned forms, with data models which are almost identical, the ratio of resource requirements per quad corresponds to the number of revisions made to each quad.
- with normal application use, the majority of statements record a small number of revisions. This is demonstrated by figure 5.
- in cases where the application updates frequently delete and insert a statement in the

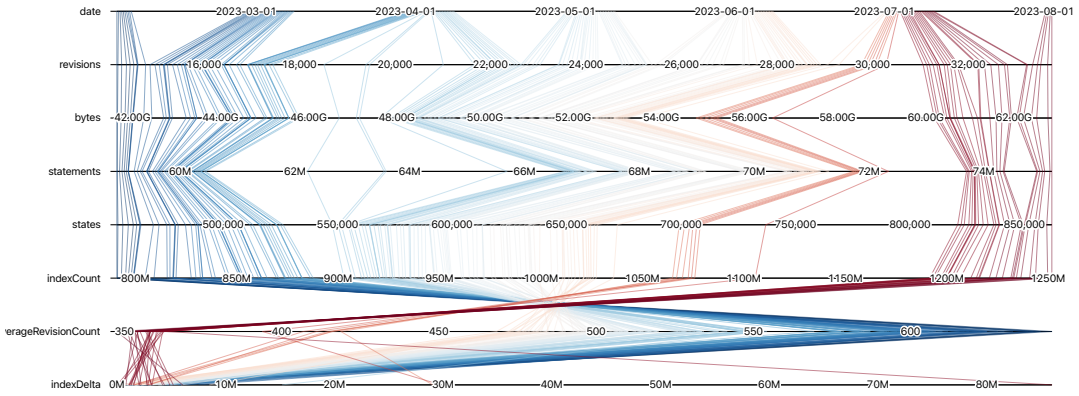

Figure 4: Correlated repository storage attributes

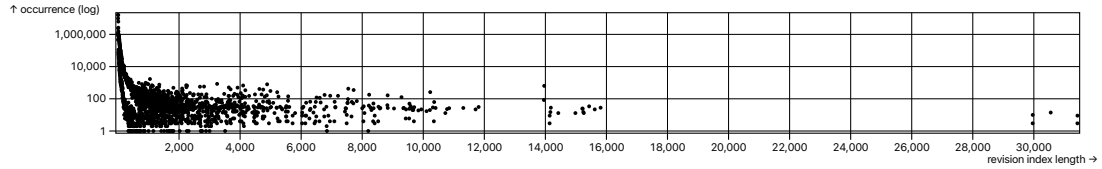

Figure 5: Quad index record length distribution for infrequent deletion/addition cycles

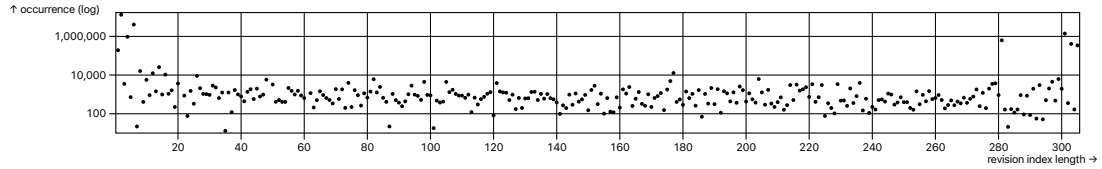

Figure 6: Quad index record length distribution for frequent deletion/addition cycles

same revision - for example the `planning-integration__rev` repository mentioned, above, this patterns leads to a large number of updates per statement, which affects the total repository size in a non-linear manner, not only due to an increased revision record length, but also because excessive index records promote inefficiency in index space utilization. This is demonstrated by figure 6

4. Query Execution Performance

As a demonstration of the effect of revisioned storage on retrospective access execution time, we report the results for frequently requested views of the principal PLM repository. We conclude from the examples that the storage architecture trade-off between space and execution time

is acceptable. Three frequently repeated views were extracted from service logs. For each, arguments were derived respective the accumulated revisions and the view was applied retrospectively. The `ebiz_article_group` and `get_list_changeditemname_for_eco` views from figures 8 and 10 involved a dataset with close to four hundred revisions of approximately fifty-five million statements. The `sfdc_salesiteminfo` in figure 12 involved a dataset with over thirteen hundred revisions of close to eighty million statements. [4]

The figures include also the "baseline" elapsed and execution times. These were measured by applying the same views to the equivalent non-revisioned repository. In both the times for the revisioned storage is approximately twice that of the un-revisioned storage.

The graphs depict both the elapsed time and the execution time, of which the latter accounts for time spent in parallel threads. In all cases the time was governed more by combinations of coincident activity, solution size, and processing overhead than by retrospective access mechanisms.

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name ?description
WHERE { { {
   ?nxpAG a ex:0 ;
          ex:1 ?name ;
          ex:2 ?description .
   ?nxpAGGraph foaf:primaryTopic ?nxpAG ;
               ex:3 ?createdGraphdate }
   BIND ( if(bound(?date), xsd:dateTime(?date),
                           '1900-11-27T18:34:55Z'^^xsd:dateTime)
          AS ?dateArg)
   FILTER (( xsd:dateTime(?createdGraphdate)) >= ?dateArg) .
   FILTER ( bound(?name) && (( ucase(?name)) = ?name) ) . }}
```

Figure 7: view 'ebiz_article_group' SPARQL text

5. Application Practice

NXP - a prominent semiconductor vendor, approached us to contribute to a project at the interface between their product lifecycle management system (PLM) and their vendor collaboration portal (VCP). Dydra acts as the product integration hub in the NXP information architecture. As a consequence, all downstream systems, including VCP, receive NXP product information through Dydra, whereby the W3C RDF access protocols constitute the programming interface.

Vendor collaboration requires agents to navigate the graph of a bill of materials (BOM) to extract the product details. The support application involves a core ontology for product descriptions, according to which each device resource associates a type, a name, a release state,

[4]This view is included to demonstrate the complexity of the standard queries, rather than their detailed SPARQL forms.

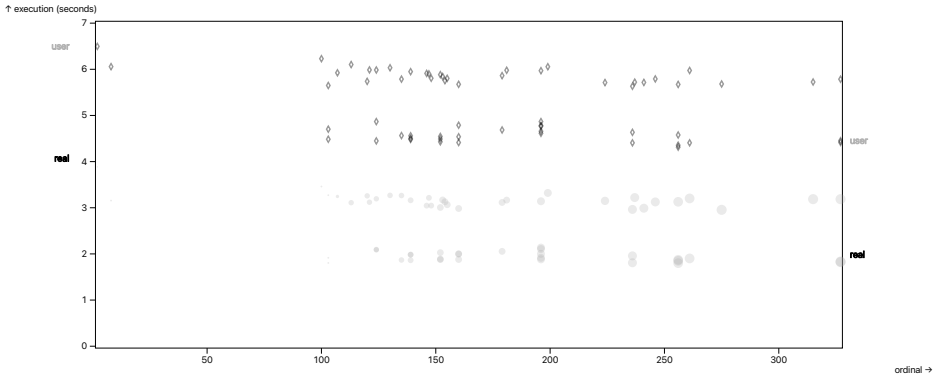

Figure 8: view 'ebiz_article_group' real and user execution times across revisions

```
SELECT ?changedItemName
 WHERE {
    { ?eco <http://example.org/63> ?ecoNumber ;
                <http://example.org/4> ?changedItem .
        ?changedItem <http://example.org/1> ?changedItemName
    }  UNION {
    ?eco <http://example.org/10> ?ecoNumber ;
              <http://example.org/7> ?changedItem .
    ?changedItem <http://example.org/10> ?changedItemName
    }
}
```

Figure 9: view get_list_changeditemname_for_eco SPARQL text

a product revision identifier, and a description, as illustrated in figure 13. For individual device types this would be extended with performance, physical, and electrical characteristics. Each attribute is represented by an individual triple. With respect to this model typical queries included clauses of the form in figure 14.

The vendor collaboration portal fetches following information from Dydra.

- ECOs (Engineering Change Orders) describe changes to device specifications which need to be acknowledged by vendors. These enumerate the details of changes to individual properties in the model.
- Manufacturing sheets are generated with information specific to the various manufacturing stages in the semiconductor product life cycle like Wafer Test, Assembly, and Final Test. The sheet content is specific to the respective manufacturing stage. There are two variants.
  - The basic page is generated to convey the information specific to the respective manufacturing stage.

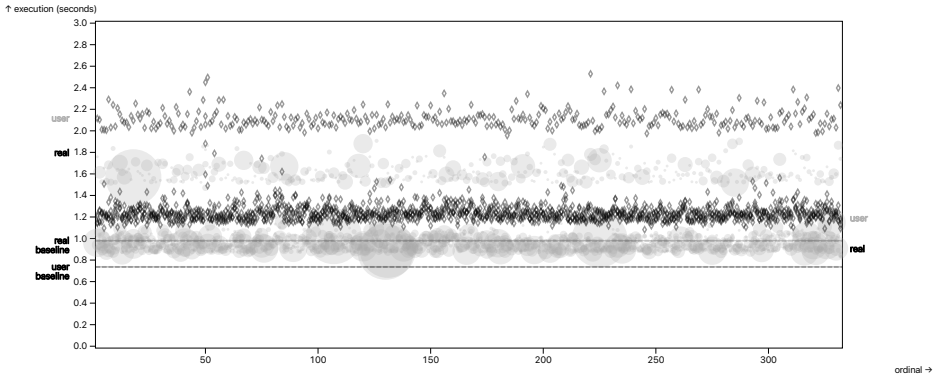

Figure 10: view get_list_changeditemname_for_eco real and user execution time across revisions

– A change order specific page conveys the same device and manufacturing information, but emphasises the differences in the manufacturing data relative to the previous ECO for the same manufacturing stage object.

The second manufacturing sheet flavour compares the latest state of the BOM for an object with the state when an object was modified by a specific ECO. At this point, the revisioned repositories in Dydra - such as the `nxp/plm__rev` repository described, above, contribute essential information. Each ECO is posted into the Dydra PLM repository in one transaction and each transaction creates a new revision of the repository. In the new revision, the revision histories of the statements which correspond to modified properties for the changed item are annotated with the new revision identifier in the manner illustrated in figure 1, above.

This revision metadata is made available to SPARQL queries via RDF-star annotation syntax[13]. The first attempt to integrate involved explicit temporal attributes extracted according to terms which corresponded to Allen interval algebra[14] and combined as tests. The initial strategy used the two metadata properties `met-by` and `starts` with a following approach

- Query for a `met-by` property within the statement pattern. This returns the ordinal where the statement, returned by the pattern, was deleted
- Also query for `starts` property for the same pattern. This returns the ordinal where the statement was inserted.
- Compare these ordinals with the ordinal which corresponds to the ECO in a such a way that start the current ordinal should not be equal to `met-by` and should be equal to `starts` .
- Combine the above conditions such that, if the statement is not deleted in the current ordinal but is inserted in the current ordinal, the test returns a true value in the response, otherwise false.
- If the test returns true, then the statement is newly inserted in the ECO repository revision. In such case, the UI redlines that specific property, i.e. applies red font, when presenting it.

A family of queries evolved through this approach to integrate the statements' revision metadata into the query solutions following the pattern illustrated in figure 16. They realized a variant of the VM and VQ access patterns[9] which drew solutions from a virtual materialised version, but, instead of annotating the solutions with version metadata, in this case, individual bindings were augmented with version metadata drawn from the respective statement. These queries produced the intended results, but they involved conversion between domains to make temporal values available to general algebra operations. This turned out to be problematic.

Dydra follows the customary RDF storage architecture, whereby repository content is dictionary-encoded. Normalized term values are replaced with term identifiers and stored quads comprise term identifiers rather than term values. This manifests the RDF principle of term identity and permits the SPARQL algebra implementation to move less data and apply faster comparisons than were solution compatibility to be determined in the term value domain. This principle does not, however, extend to terms with temporal value domains. In order to index them efficiently and to order results properly, their representation must reflect their respective value domain. The linear revision index associated with each statement comprises monotonically increasing revision ordinal or time-based UUID values. If temporal properties are to be combined with results from statement patterns in a filter, as in figure 16, they must be transformed into term identifiers in order to pass them through the algebra operators. Where they are dictionary-encoded in order to incorporate them into a solution, that introduces a delay: the current term interning rate is only seventy-thousand per second.

The limited performance of this approach demonstrated the issue noted above, that revision metadata is not immediately suitable to be included in BGP solutions. A more efficient approach is to reduce interning by applying as much logic as possible in the temporal value domain. Rather than formulating the queries with annotations and filters, a more efficient implementation is to introduce temporal operators which embody the complete logic. In order to circumvent that rate limit, we introduced a virtual temporal attribute to designate an operator which applies the described logic to compute whether the statement had been introduced in a given revision. This attribute was bound in the respective solution for each device parameter to a single boolean variable specific to the parameter.

The result was queries according to the pattern illustrated in figure 17 where the implementation encapsulates this logic in a more optimum system function called `not-asserted`. This not only executes the above operation more optimally, it also provided for cleaner code. The approach permitted queries of the complexity illustrated in figure 18. The effect of the revised logic was to reduce their execution times from minutes to tens of seconds.

Acknowledgments

Thanks to the developers of SBCL and LMDB upon which Dydra is implemented.

References

[1] J. Anderson, A. Bendiken, Transaction-time queries in dydra, in: J. Debattista, J. Umbrich, J. D. Fernández, A. Rula, A. Zaveri, M. Knuth, D. Kontokostas (Eds.), Joint Proceedings of

the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW 2016) and the 3rd Workshop on Linked Data Quality (LDQ 2016) co-located with 13th European Semantic Web Conference (ESWC 2016), Heraklion, Crete, Greece, May 30th, 2016., volume 1585 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 11–19. URL: http://ceur-ws.org/Vol-1585/mepdaw2016_paper_02.pdf.

[2] J. Anderson, Rdf graph stores as convergent datatypes, in: Companion Proceedings of The 2019 World Wide Web Conference, 2019, pp. 940–942.

[3] W. Ali, M. Saleem, B. Yao, A. Hogan, A.-C. N. Ngomo, A survey of rdf stores & sparql engines for querying knowledge graphs, The VLDB Journal (2022) 1–26.

[4] F. Kovacevic, F. J. Ekaputra, T. Miksa, A. Rauber, Starvers-versioning and timestamping RDF data by means of RDF* – an approach based on annotated triples.

[5] R. Taelman, M. Vander Sande, J. Van Herwegen, E. Mannens, R. Verborgh, Reflections on: triple storage for random-access versioned querying of rdf archives, in: Journal Track at the 18th International Semantic Web Conference (ISWC 2019), volume 2576, 2019.

[6] R. Taelman, T. Mahieu, M. Vanbrabant, R. Verborgh, Optimizing storage of rdf archives using bidirectional delta chains, Semantic Web 13 (2022) 705–734.

[7] H. Chu, Mdb: A memory-mapped database and backend for openldap, in: Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany, volume 35, 2011.

[8] F. Grandi, Multi-temporal rdf ontology versioning., in: IWOD@ ISWC, 2009.

[9] J. D. Fernández, J. Umbrich, A. Polleres, M. Knuth, Evaluating query and storage strategies for rdf archives, in: Proceedings of the 12th International Conference on Semantic Systems, ACM, 2016, pp. 41–48.

[10] O. Pelgrin, L. Galárraga, K. Hose, Towards fully-fledged archiving for rdf datasets, Semantic Web 12 (2021) 903–925.

[11] I. Cuevas, A. Hogan, Versioned queries over rdf archives: All you need is sparql?, in: MEPDaW@ ISWC, 2020, pp. 43–52.

[12] O. Pelgrin, R. Taelman, L. Galárraga, K. Hose, Scaling large rdf archives to very long histories, in: 2023 IEEE 17th International Conference on Semantic Computing (ICSC), IEEE, 2023, pp. 41–48.

[13] O. Hartig, P.-A. Champin, G. Kellogg, A. Seaborne, D. Arndt, J. Broekstra, B. DuCharme, O. Lassila, P. F. Patel-Schneider, E. Prud'hommeaux, et al., Rdf-star and sparql-star. w3c draft community group report, 2022.

[14] J. F. Allen, Maintaining knowledge about temporal intervals, Communications of the ACM 26 (1983) 832–843.

instance	revisions	quads (M)	bytes (G)	bytes/quad	bytes/rev. quad	IC bytes (G)	CB bytes (G)	quad deltas	quad change ratio.
prod:nxp/planning-integration__rev	93	3.08	45.10	14625	157.25	402.32	29.53	1845416608	11668.99
prod:nxp/plm	0	56.19	15.43	275					
prod:nxp/plm__rev	33569	56.19	67.57	1202	0.03582	1,787.57	7.38	461107538	613.28
qa:nxp-hws/plm__rev	13396	75.46	30.19	400	0.02987	370.87	2.31	144462051	181.99
qa:nxp/planning-integration__rev	36	2.96	36.07	12170	338.05	16.00	8.17	510373696	17219.74
qa:nxp/plm	0	40.27	11.00	273					
qa:nxp/plm__rev	336	54.08	17.70	327	0.97428	310.13	0.062	3880837	7.159
dev:nxp-hws/plm__rev	9178	46.62	14.63	314	0.03420	209.61	0.007	438046	0.936
dev:nxp/plm	0	39.77	10.60	266					
dev:nxp/plm__rev	60629	39.77	12.60	317	0.00523	232.92	0.002	115350	0.284

Table 3: Repository storage requirements and revision properties for variants from production, Q/A, and development environments

Figure 11: view 'sfdc_salesiteminfo' SPARQL text

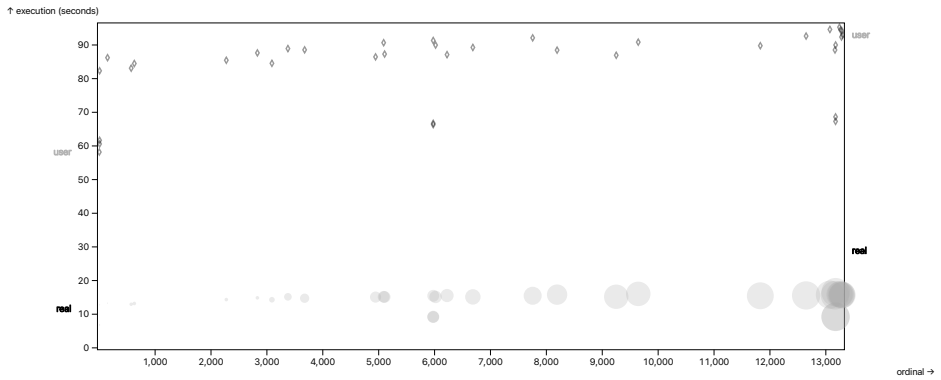

Figure 12: view 'sfdc_salesiteminfo' real and user execution time across revisions

```
prefix plm: <http://nxp.com/data> .
prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
plm:Device a rdfs:Class .
plm:name a rdfs:predicate;
     rdfs:domain plm:Device;
     rdfs:range xmxls:string .
plm:state  a rdfs:predicate;
     rdfs:domain plm:Device;
     rdfs:range xmxls:string .
plm:revision  a rdfs:predicate;
     rdf:domain plm:Device;
     rdfs:range xmxls:string .
plm:description  a rdfs:predicate;
     rdfs:domain plm:Device;
     rdfs:range xmxls:string .
```

Figure 13: Illustrative PLM ontology core

```
prefix : <urn:dydra:>
select *
from :all
where {
  ?s a ?type ;
    plm:name ?name ;
    plm:state ?state ;
    plm:revision ?revision ;
    plm:description ?description .
}
```

Figure 14: a query which projects the minimal device description

```
prefix : <urn:dydra:>
prefix plm: <https://nxp.com/plm#>
select *
from :all
where {
  ?s a ?type
       {| :met-by ?type_metBy ;  :starts ?type_starts |} ;
    plm:name ?name
       {| :met-by ?name_metBy ;  :starts ?name_starts |} ;
    plm:state ?state
       {| :met-by ?state_metBy ; :starts ?state_starts |} ;
    plm:revision ?revision
      {| :met-by ?revision_metBy ; :starts ?revision_starts |} ;
    plm:description ?description
       {| :met-by ?description_metBy ; :starts ?description_starts |} .
}
```

Figure 15: an annotated device query

```
prefix : <urn:dydra:>
select *
from :all
where {
  ?s a ?type
         {| :met-by ?type_metBy ; :starts ?type_starts |} ;
    plm:name ?name
         {| :met-by ?name_metBy ; :starts ?name_starts |} ;
    plm:state ?state
         {| :met-by ?state_metBy ; :starts ?state_starts |} ;
    plm:revision ?revision
         {| :met-by ?revision_metBy ; :starts ?revision_starts |} ;
    plm:description ?description
         {| :met-by ?description_metBy ; :starts ?description_starts |}   .

  bind(?type_metBy != ?type_starts as ?type_redline)
  bind(?name_metBy != ?name_starts as ?name_redline)
  bind(?state_metBy != ?state_starts as ?state_redline)
  bind(?revision_metBy != ?revision_starts as ?revision_redline)
  bind(?description_metBy != ?description_starts as ?description_redline)
}
```

Figure 16: a query which compares temporal attributes to determine "redline" status

```
prefix : <urn:dydra:>
select *
from :all
where {
  ?s a ?type {| <urn:dydra:notAsserted> ?type_redline |} ;
    plm:name ?name
          {| <urn:dydra:notAsserted> ?name_redline |} ;
    plm:state ?state
          {| <urn:dydra:notAsserted> ?state_redline |} ;
    plm:revision ?revision
          {| <urn:dydra:notAsserted> ?revision_redline|} ;
    plm:description ?description
          {| <urn:dydra:notAsserted>  ?description_redline |}    .
}
```

Figure 17: a query reformulated to use a natively implemented predicate

Figure 18: an indicative redline query : 90 statement patterns, 61 annotations, 143 algebra expressions