# StarVers - Versioning and Timestamping RDF data by means of RDF* - An Approach based on Annotated Triples

Filip Kovacevic [a,*], Fajar Juang Ekaputra [a], Tomasz Miksa [b] and Andreas Rauber [a]
[a] *E194-01 - Research Unit of Information and Software Engineering, TU Wien, Vienna, Country*
*E-mails: filip.kovacevic@tuwien.ac.at, fajar.ekaputra@tuwien.ac.at, rauber@ifs.tuwien.ac.at*
[b] *SBA Research, TU Wien, Vienna, Austria*
*E-mail: miksa@ifs.tuwien.ac.at*

**Abstract.** To foster reproducible and verifiable research results the RDA Data Citation Working Group issued a set of 14 recommendations. The core of these recommendations revolves around preparing data storages so that arbitrary subsets of any evolving data set can be efficiently identified and cited at any specific state or point in time. Based on these recommendations we identified an efficient solution for RDF/triple stores which so far have only used cumbersome mechanisms to aforementioned ends. Our solution employs RDF* and SPARQL* to annotate data triples with temporal metadata and thereby allows for retrieval of datasets as they were at a specific point in time. It furthermore solely relies on triple stores with RDF* and SPARQL* support, such as Jena TDB, GraphDB, Stardog and others and thus does not require any Git-like versioning systems. We evaluate our work by employing the BEAR framework where we use specific components, such as the BEAR-B dataset (hourly DBPedia snapshots), its corresponding queries, Jena TDB as triple store and the quads-based timestamp-based approach. We also extend the java implementation of this framework by methods and functions for handling RDF* queries and GraphDB as additional triple store. Our BEAR extension is publicly available on Github[1].

Keywords: RDF, RDF*, SPARQL, SPARQL*, Versioning, Triple Store, Data Citation, GraphDB, Jena TDB

## 1. Introduction

Over the last years data has become increasingly central and critical in both research and applications. With the rapid transition towards the fourth paradigm of science (i.e., data-intensive scientific discovery) [1], where data is as vital to scientific progress as traditional publications are, challenges like data provenance, identifying subsets, authorship of data, evolution of data over time and long-term data preservation become apparent. These challenges can also be summarized as Data Citation challenges and to tackle them the Data Citation Working Group (DCWG) of the Research Data Alliance (RDA) released a set of recommendations [2] by which our work is motivated. These recommendations consider a data store where the data inside is versioned and operations on data are timestamped. While these recommendations have been implemented for a wide range of data types (including relational databases, multidimensional data cubes, comma-separated value files) and deployed by several data centers as operational solutions [3], no efficient solution for a prominent type of data, i.e. triple stores, has been presented to-date. Triple

*Corresponding author. E-mail: filip.kovacevic@tuwien.ac.at.
[1] https://github.com/GreenfishK/BEAR

stores are a fundamental storage type for (enterprise) knowledge graphs and their schemata, namely ontologies, which research institutions but also enterprises mine for knowledge discovery. We aim at making ontology- and knowledge graph (KG)-based decisions tractable and reproducible as the knowledge represented by these evolves. In this paper, we present StarVers – a solution to the preparation of RDF/triple stores for Data Citation.
The remainder of the paper is structured as follows: In Section 2 we first give an overview of relevant RDF archiving mechanisms, to determine a baseline to compare versioning in triple stores and queries performed against archives. We further outline the RDA DCWG's recommendations. Last, we introduce RDF* and SPARQL* as our main approach to triple store *preparation*. In Section 3 we walk through StarVers via a running example including our timestamp-based versioning approach and read & write statements that a triple store management system should employ to be able to retrieve specific dataset versions. After that, we show an evaluation of StarVers's timestamp-based queries by employing and extending the BEAR RDF Archiving Benchmark framework [4] in Section 4. We also provide the full source code for the evaluation[2]. Last, we summarize our work and provide arguments why our RDF* approach to versioning RDF datasets should be preferred over named graphs as BEAR's timestamp-based reference approach.

## 2. Related Work

First, we give an overview of RDF archiving systems and frameworks that have been researched and proposed over the last two decades. While we do not propose a fully-fledged RDF archiving system we do characterize our work by features that are typical for such systems. As our work is motivated by reproducible research results, we consider querying specific dataset versions with temporal queries as one of the core requirements. Without such retrieval possibilities data-intensive experiments would not be reproducible. Therefore, we secondly outline the recommendations proposed by the RDA Data Citation Working Group [2] which are a means to reproducible research results and emphasize the recommendations our solution is based upon. Lastly, we introduce RDF* and SPARQL*, which are the principal concepts that we employ in our solution design.

### *2.1. RDF Archiving Mechanisms*

Papakonstantinou et al. [5] and Pelgrin et al. [6] studied RDF Archiving Systems & Frameworks and summarized them based on certain attributes, such as *Storage Paradigm*, *Data Model* or *Retrieval Functionality*. Inhere we focus on the concrete attribute values (=characteristics) from these studies that our work can be described with and contextualize it with respect to related work from the literature.
In **Timestamp-based approaches** triples, revisions or datasets are associated with temporal metadata by means of quads or temporal tables. Temporal metadata most importantly include creation and deletion timestamps. Datasets can be queried by including timestamps or validity intervals. Some of the systems extend the SPARQL language by additional components or clauses ([7], [8], [9], [10]). Other approaches include storing changesets between every two consecutive revisions (Change-based (CB)) or delta-based approaches) and making independent copies (IC) or snapshots after every change. Synonyms for the latter approach are full materialization, snapshots and independent copies. With the emergence of data streams and "live data", such as DBPedia live, the IC approach becomes infeasible due to unattainable storage requirements, say, for secondly snapshots. The CB approach would meet the storage requirements as only changes are stored which can be as small as one record for an inserted triple within few seconds. However, to identify a specific subset as it was at a certain point in time one would need to apply every change up to that timestamp, which again for live data does not scale well. TB approaches combine the advantages of both worlds by having comparable overhead costs to CB approaches as only temporal annotations are stored with every change but also leaving space for fast retrieval methods, such as time interval indexes that resolve to a specific snapshot.
Assigning temporal metadata to triples can be achieved with **nested triples**, based on the nesting paradigm from the field of informatics and has first been described and applied to RDF by [11]. To the best of our knowledge this

[2] http://w3id.org/fkresearch/starvers

metadata annotation approach has so far not been employed in any timestamp-based RDF Archiving System. We explicate this approach in Section 2.3. RDF archiving systems and frameworks from the literature use quad semantics[3] on the physical level to link triples and metadata by either pointing to graphs that represent graph or dataset revisions in which the triple was active or using the fourth component directly for the commit timestamp or interval [6].

**Version Materialization** (VM) queries aim to retrieve a specific dataset version from the history based on a timestamp or revision number. In Data Citation this retrieval functionality is necessary in order to reproduce research results. These are also the most basic queries every archiving system supports. However, archiving systems usually support additional retrieval functionalities on top. Below we list the most typical query types and give an informal and brief description for each type.

Version (V): Returns the versions' labels for which a particular query yields results.
Delta Materialization (DM): Standard queries defined on the change set of two consecutive versions.
Cross-version (CV): Standard queries that combine multiple dataset versions via set operations, joins and aggregations.
Cross-delta (CD): Like CV-queries, just that the change sets are combined to get a query result.

**Branches & Tags** are versioning features commonly supported by git-like collaborative development tools. While we focus in our work on linear timelines we do consider such features as relevant. None of the timestamp-based systems and frameworks from the literature ([5] and [6]) has implemented these features so far. Thus, we see some potential for future work in such functionalities.

Another feature were we do see a lack of support in timestamp-based systems is the versioning of **multiple graphs** or **Multi-Graphs**. In SPARQL one can refer to and query from different named graphs in one query. If the versioning approach is based on revision numbers a synchronization between these graphs or resolution of these revision numbers at the dataset level would be necessary as each graph can have its own revision history. Timestamps, contrary to revision numbers, have a globally agreed meaning whereas two syntactically equal revision numbers or labels could resolve to different timestamps. At the time of writing, only Dydra [9] support Multi-Graph versioning, which stores revision data including timestamps of each revision at the dataset level.

Handling **concurrent updates** with either automatic or manual conflict resolution has been implemented in some CB and IC-based systems, including [12], [13], [14], [15], [16], which drew their ideas from version control systems. Timestamped-based solutions, on the other hand, have so far been missing out this feature ([6]).

### *2.2. Data Citation*

The RDA Data Citation Recommendations [17] introduced by the Research Data Alliance Data Citation Working Group aims at keeping research results reproducible by tackling the issue of dynamic and evolving datasets. Reproducing and verifying research results is necessary for the scientific method in general and it is inevitable for assessing the validity of an experiment. The recommendations are based upon versioned data, timestamping and a query subsetting mechanism. In this paper we focus on the first two recommendations (R1, R2) as they are directly related to the timestamp-based approach of archiving datasets, which we described in the previous section. A recent survey [3] summarizes a number of reference implementations as well as fully deployed implementations in a range of data infrastructures, covering a broad variety of data types, ranging from relational databases to multidimensional data cubes in different settings. We additionally discovered that data versioning is the most adopted recommendation that the survey explicitly mentions and describes for every implementation. However, no approach for versioning and precisely identifying arbitrary subgraphs from evolving RDF data sets has been discussed yet.

### *2.3. RDF* and SPARQL**

One way to implement the Data Citation Recommendations R1 & R2 for RDF datasets (and probably the only possible way) is to use statement-level annotations to assign temporal metadata to data triples. In the literature we

[3] https://www.w3.org/TR/n-quads/

Table 1
RDA Data Citation recommendations[17]

| Area | RDA Data Citation Recommendation |
|---|---|
| Preparing the Data and Query Store | **R1 – Data Versioning** |
| | **R2 – Timestamping** |
| | R3 – Query Store Facilities |
| Persistently Identifying Specific Datasets | R4 – Query Uniqueness |
| | R5 – Stable Sorting |
| | R6 – Result Set Verification |
| | R7 – Query Timestamping |
| | R8 – Query PID |
| | R9 – Store Query |
| | R10 – Automated Citation Texts |
| Resolving PIDs and Retrieving the Data | R11 – Landing Page |
| | R12 – Machine Actionability |
| Upon modifications to the Data Infrastructure | R13 – Technology Migration |
| | R14 – Migration Verification |

found approaches specifically dedicated to temporal metadata annotations (see summary table in [18]). As far as we can tell, the community has not commonly agreed on any of these approaches, nor have they been integrated as features by currently prominent triple stores, such as Jena, GraphDB, Stardog and others. We, however, found RDF*/SPARQL* [11] – a general purpose statement-level annotation approach – which has indeed been adopted by aforementioned triple stores. It uses nested triples in the form $<<?s?p?o >>?mp?mo$ where $<<?s?p?o >>$, which we call *data triple*, is nested into the subject of a metadata triple. It can therefore conveniently be used to add temporal metadata to data triple statements. RDF*/SPARQL* has been acknowledged by W3C and is on its way to become a W3C standard[4]. Popular serialization formats like turtle and n-triples have also been extended to turtle-star and n-triples-star in W3C's draft version. Libraries like RDF4J do already implement these serialization formats. In benchmark studies it has been compared to other prominent statement-level representations, like singleton properties, reification and shown to be more effective in the number of stored triples [19]. Moreoever, it is already applied in the YAGO4 knowledge graph[5] to represent temporal facts. Authors in [20] showed how to query temporal facts from YAGO4 via SPARQL*. RDF* converters from RDF [21] and heterogeneous [22] sources have also been proposed. The conversion from RDF to RDF* is an important first step in our solution design.

## 3. StarVers Solution Design

In order to version RDF datasets within RDF* stores StarVers solely relies on RDF* and SPARQL* and hence does not require versioning tools, such as CVS, SVN or Git, nor does it require separate (relational) databases to store the versions. We start with defining a running example in Section 3.1 which we use throughout this section to showcase the SPARQL* update & query operations and RDF* result sets. In Section 3.2, we explain the core principles which revolve around the *version timestamp* and two metadata attributes to annotate the start and expiration of a triple.
In order to make an RDF dataset ready for versioning we need to initially annotate every triple in this dataset. We distinguish between two annotation styles – one intuitive and one more structural and less redundant way (see Section 3.3).

[4] urlhttps://w3c.github.io/rdf-star/
[5] https://yago-knowledge.org/downloads/yago-4

Table 2
Two versions of a triple from the UniProt Ontology

| Subject | Predicate | Object |
|---|---|---|
| <http://purl.uniprot.org/diseases/5622> | <http://www.w3.org/2004/02/skos/core#prefLabel> | "Intellectual developmental disorder 59" |
| <http://purl.uniprot.org/diseases/5622> | <http://www.w3.org/2004/02/skos/core#prefLabel> | "Intellectual developmental disorder, autosomal dominant 59" |

For the write operation types insert, update and delete we design SPARQL* templates that can be used to update RDF* datasets. These templates feature timestamping so that with every update a new version is implicitly created. We dedicate a section to each of these write operations (Sections 3.4, 3.5 and 3.6) and demonstrate each of them with our running example. This yields three different versions of the dataset. In the last Section 3.7 we show how each version can be queried by simply changing the version timestamp and what the results represented with RDF* look like.

### *3.1. Running example*

To demonstrate our solution we will assume a dataset with only two triples from the Uniprot KG[6], as shown in Table 2 which depict an intellectual developmental disorder:

The second triple is the updated version of the first with a new object literal. Both appear in snapshots of the Uniprot KG (release 2021_02[7] and release 2021_03[8]) which had been consecutively released with an almost two months interval in between. We assume following *version timestamps* for these triples:

version timestamp 1: 2021-04-07T12:00:00.000+00:00
version timestamp 2: 2021-06-02T12:00:00.000+00:00
version timestamp 3: 2022-03-01T12:00:00.000+00:00

The first two we derived from the snapshots' release dates, which are 07.04.2021 and 02.06.2021 respectively. The third one we made up for the sake of simulating the deletion process of the 2nd triple version (see later in Section 3.7). For the SPARQL update and query statements we show in the subsequent sections we assume following prefixes to be at the beginning of each statement:

```
PREFIX vers: <https://w3id.org/fkresearch/starvers/versioning/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
```

*vers* is used to denote one of the two timestamp predicates we use for versioning (see next section), *xsd* resolves to a common timestamp schema and *skos* points at an IRI from the UniProt Onotology.

### *3.2. Representing timestamped RDF triples with RDF**

To version triples we employ two attributes as RDF predicates – *valid_from*, denoting a creation timestamp and *valid_until* denoting an expiration timestamp – to bridge data triples (subject) and timestamp values (object). This results in nested RDF* triples, as explicated in Section 2.3.
In Table 3 we show how the data triple from our running example is represented as timestamped RDF* triple. We call this annotation style *flat RDF** in analogy to flat data tables where data is redundantly stored in flat tables and in return *easier* to query. Alternatively, we can enforce more structure and less redundancy by representing the timestamped triple in a hierarchical way, as shown in Table 4. Inhere, we refer to this annotation style as *hierarchical RDF**. The two version predicates have a prefix *vers* which resolves to

[6]https://www.uniprot.org/uniprot/
[7]https://www.uniprot.org/news/2021/04/07/release
[8]https://www.uniprot.org/news/2021/06/02/release

Table 3
Timestamped triples example from the UniProt Ontology 2021_02 release represented as flat RDF*

| <<Subject Predicate Object>> | Predicate | Object |
|---|---|---|
| << <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" >> | vers:valid_from | "2021-04-07T12:00:00.000+00:00" ^^xsd:date. |
| << <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" >> | vers:valid_until | "9999-12-31T00:00:00.000+00:00" ^^xsd:date. |

Table 4
Timestamped triples example from the UniProt Ontology 2021_02 release represented as hierarchical RDF*

| <<<<Subject Predicate Object>> Predicate Object >> | Predicate | Object |
|---|---|---|
| <<<< <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" >> vers:valid_from "2021-04-07T12:00:00.000+00:00"^^xsd:date. >> | vers:valid_until | "9999-12-31T00:00:00.000+00:00" ^^xsd:date. |

https://w3id.org/fkresearch/starvers/versioning/. This IRI is just a concatenation of our Github page where we published our evaluation of StarVers plus /versioning as an additional suffix to give more context. The first version timestamp denotes the release date of the snapshot where this triple comes from. The second timestamp is an artificially created timestamp that is far in the future and depicts the triple being valid until further notice.

### *3.3. Making an RDF Dataset Versioning-Ready*

Usually, a new RDF dataset would be versioned ab-initio, i.e. each tripe would receive the timestamp according to its insertion into the triple store. In cases where an existing dataset should be made a versioned one, or where an RDF dataset is constructed before being released to be queried, we can apply a more efficient approach by assigning the same initial timestamp to all triples. In such settings, any RDF dataset can be transformed into a timestamp-based versioned dataset by attaching version timestamps to it, like we showed in all RDF* examples so far in this section. One simple way how to construct a versioned dataset is to use SPARQL's update language to set the predicates vers:valid_from and vers:valid_until and version timestamps as objects for the data triples. This can be realized with one single SPARQL statement where we first delete all matched data triples via the delete block and second re-insert the matched data triples as nested triples together with their version timestamps via the insert block. This logic is independent of the timestamped triples' representation (flat or hierarchical RDF*). We show either way in Listings 1 and 2. The difference between these two approaches is that in *hierarchical RDF** we have one more hierarchy of nesting and the data triple is unique.

Listing 1: Construct a timestamp-based versioned RDF* dataset with flat RDF*

```
delete
{
    ?s ?p ?o .
}
insert
{
    <<?s ?p ?o>> vers:valid_from
    ?currentTimestamp.
    <<?s ?p ?o>> vers:valid_until
    "9999-12-31T00:00:00.000+02:00"
    ^^xsd:dateTime .
}
where
{
   ?s ?p ?o .
   BIND(xsd:dateTime(NOW()) AS ?
       currentTimestamp).
}
```

Listing 2: Construct a timestamp-based versioned RDF* dataset with hierarchical RDF*

```
delete
{
    ?s ?p ?o .
}
insert
{
    <<<<?s ?p ?o>> vers:valid_from
    ?currentTimestamp>>
    vers:valid_until
    "9999-12-31T00:00:00.000+02:00"
    ^^xsd:dateTime.
}
where
{
   ?s ?p ?o .
   BIND(xsd:dateTime(NOW()) AS ?
       currentTimestamp).
}
```

### *3.4. Insert*

Once the RDF* dataset has been constructed and every triple has thereby been annotated with the *valid_from* system timestamp and the artificial *valid_until* timestamp, we can use the same principle as in the insert block of Listings 1 and 2 from the previous section to add newer triples to it. The only difference is that we bind actual resources to variables ?s, ?p and ?o. In Listings 3 and 4 we demonstrate how we insert the first version of the triple from our running example in the style of flat and hierarchical RDF*, respectively. If we assume for the sake of this example that these two SPARQL* update statements were executed on the 2021_02 snapshot's release date we ultimately get the resources including a timestamp from that day in the structure as presented in Tables 3 and 4, respectively.

While the code for inserting only one triple can be conciser, the statements we present have a flexibility and scaling advantage. They can be used for bulk inserts by just listing more triples inside the VALUES block without the need to restate the nested triples or re-execute the whole statement for every triple we want to insert.

Listing 3: Running example using insert template for flat RDF* timestmaped triples

```
insert {
    <<?s ?p ?o>>
    vers:valid_from
    ?newVersion.
    <<?s ?p ?o>>
    vers:valid_until
    "9999-12-31T00:00:00.000+02:00"^^xsd:
        dateTime.

}
where {
    values (?s ?p ?o) {
        (<http://purl.uniprot.org/diseases
            /5622>
         skos:prefLabel
         "Intellectual␣developmental␣disorder
             ␣59")
        # Add further triples
    }
    BIND(xsd:dateTime(NOW()) AS ?newVersion)
}
```

Listing 4: Running example using insert template for hierarchical RDF* timestamped triples

```
insert {
    <<
        <<?s ?p ?o>>
        vers:valid_from
        ?newVersion
    >>
    vers:valid_until
    "9999-12-31T00:00:00.000+02:00"^^xsd:
        dateTime .
}
where {
    values (?s ?p ?o) {
        (<http://purl.uniprot.org/diseases
            /5622>
         skos:prefLabel
         "Intellectual␣developmental␣disorder
             ␣59")
        # Add further triples
    }
    BIND(xsd:dateTime(NOW()) AS ?newVersion)
}
```

### 3.5. Update

In SPARQL, update statements refer to either inserts, deletion or a combination of them. A deletion followed by an insert is exactly what we need to do when we want to replace an old version of a resource by a new one. In here, we showcase the timestamped update as an object value update. Nevertheless, this logic can be applied to resources in subject or predicate positions in an analogous way. In Listing 5 and 6 we update the triple from our running example to the state as recorded in Uniprot's 2021_03 snapshot. We again assume that we perform this update on the release date of latter snapshot in order to simulate a timestamp from that day returned by the *now()* function. We start by deleting the nested triple with the artificial *valid_until* timestamp and subsequently re-insert the same triple just with the system timestamp for the *valid_until* predicate instead. This way we mark the end of this triple's validity. Next, we insert the new triple by the same logic as we do when we insert a new timestamped triple (see previous section). Note, that the expiration timestamp from the outdated triple and the creation timestamp from the new triple are the same. Like before, we use variables in the delete and insert blocks that we replace with actual resources in the where clause. Additionally, we make sure that the old and new value are not the same. The elaborated logic works for either annotation style.

Listing 5: Running example using the update template for flat RDF* timestamped triples

```
delete {
    <<?s ?p ?o>> vers:valid_until
        "9999-12-31T00:00:00.000+02:00"^^xsd:
            dateTime
}
insert {
    <<?s ?p ?o>> vers:valid_until ?newVersion
        .
    <<?s ?p ?newValue>> vers:valid_from ?
        newVersion
        ;vers:valid_until "9999-12-31T00
            :00:00.000+02:00"^^xsd:dateTime.
}
where {
    bind(<http://purl.uniprot.org/diseases
        /5622>  as ?s)
    bind(skos:prefLabel as ?p)
    bind("Intellectual␣developmental␣disorder
        ␣59"  as ?o)
    bind("Intellectual␣developmental␣disorder
        ,␣autosomal␣dominant␣59"  as ?
        newValue).
    # versioning
    <<?s ?p ?o>> vers:valid_until "9999-12-31
        T00:00:00.000+02:00"^^xsd:dateTime.
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
    # nothing should be changed if old and
        new value are the same
    filter(?newValue != ?o)
}
```

Listing 6: Running example using the update template for hierarchical RDF* timestamped triples

```
delete {
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from >> vers:valid_until
        "9999-12-31T00:00:00.000+02:00"^^xsd:
            dateTime
}
insert {
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from>> vers:valid_until ?
        newVersion.
    <<<<?s ?p ?newValue>> vers:valid_from ?
        newVersion >> vers:valid_until "
        9999-12-31T00:00:00.000+02:00"^^xsd:
        dateTime.
}
where {
    bind(<http://purl.uniprot.org/diseases
        /5622>  as ?s)
    bind(skos:prefLabel as ?p)
    bind("Intellectual␣developmental␣disorder
        ␣59"  as ?o)
    bind("Intellectual␣developmental␣disorder
        ,␣autosomal␣dominant␣59"  as ?
        newValue).
    # versioning
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from>> vers:valid_until "
        9999-12-31T00:00:00.000+02:00"^^xsd:
        dateTime.
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
    # nothing should be changed if old and
        new value are the same
    filter(?newValue != ?o)
}
```

### 3.6. Outdate

Since it should always be possible to retrieve data as of a specific timestamp we do not delete triples, we *outdate* them. Trivially, a timestamped triple can only be outdated if it was constructed or inserted before. Hence, an artificial *valid_until* timestamp must exist on that triple. Outdating a timestamped triple simply means replacing its artificial end date by an actual one, which in our solution is always the system timestamp. In terms of SPARQL*, we achieve this by a delete block where we state the currently valid triple followed by an insert where we state the triple with the new valid_until timestamp. In Listings 7 and 8 we show how we apply this logic for either annotation style to outdate the latest version of the triple from our running example.

Listing 7: Running example using the outdate template for flat RDF* timestamped triples

```
delete {
    <<?s ?p ?o>> vers:valid_until "9999-12-31
        T00:00:00.000+02:00"^^xsd:dateTime
}
insert {
    <<?s ?p ?o>> vers:valid_until ?newVersion
        .
}
where {
    bind(<http://purl.uniprot.org/diseases
        /5622> as ?s)
    bind(skos:prefLabel as ?p)
    bind("Intellectual␣developmental␣disorder
        ,␣autosomal␣dominant␣59" as ?o)
    <<?s ?p ?o>> vers:valid_until "9999-12-31
        T00:00:00.000+02:00"^^xsd:dateTime .
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
}
```

Listing 8: Running example using the outdate template for hierarchical RDF* timestamped triples

```
delete {
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from>> vers:valid_until "
        9999-12-31T00:00:00.000+02:00"^^xsd:
        dateTime
}
insert {
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from>> vers:valid_until ?
        newVersion
}
where {
    bind(<http://purl.uniprot.org/diseases
        /5622> as ?s)
    bind(skos:prefLabel as ?p)
    bind("Intellectual␣developmental␣disorder
        ,␣autosomal␣dominant␣59" as ?o)
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from>> vers:valid_until "
        9999-12-31T00:00:00.000+02:00"^^xsd:
        dateTime .
    BIND(xsd:dateTime(NOW()) AS ?newVersion).
}
```

### *3.7. Version Materialisation*

To materialize a specific version of the dataset we use the SPARQL queries shown in Listings 9 and 10 for flat and hierarchical RDF* annotation styles, respectively. To get a snapshot from a specific point in time all we need to do is to plug in a timestamp into the query template. Every triple for which the timestamp lies between its *valid_from* and *valid_until* timestamps will be returned. Hence, we do not need to know the exact update timestamp, unlike systems with version labels. To retrieve the latest version of the dataset, the current system timestamp does the trick for us.

In these SPARQL* templates we use <timestamp> as a placeholder for the timestamps defined for our running example in Section 3.1. If we execute these queries with each of the three timestamps we get the result sets as presented in Tables 5 and 6, respectively. The first query returns the triple as it was at the release data of Uniprot's 2021_02 snapshot. This is the result of our simulated insert statement. By looking at the expiration timestamp, we see that this triple expired with the release of a new snapshot, namely 2021_03. By querying with the 2nd timestamp we hence do not get this triple as result anymore. Instead, we get its updated version where the object value changed. This update we simulated with our write statement from Section 3.5. Also this query did already expire as we outdated it in Section 3.6. Finally, if we execute Query3 to retrieve the dataset past this expiration date we get an empty set.

For demonstration we used a simple lookup query with one triple statement inside the BGP. However, a more complex query can include more than one triple statement. In this case, each data triple statement needs its own pair of timestamp variables as the bindings can differ between these pairs. To elucidate this, imagine a second data triple statement *?o ?p2 ?o2* which together with the first data triple statement, *?s ?p ?o* forms a sequence path. If we annotate the second data triple statement with the same timestamp variable pair as the first one, we would next to ?o, also perform a join on ?valid_from and ?valid_until which will yield no match if the data triple statements' underlying subsets do not share a common timestamp. Due to these SPARQL technicalities we extend the timestamp variables with a numerical suffix which we simply increase for each data triple statement inside a BGP. We illustrate

this in Listings 11 and 12 by extending the simple case with an additional data triple statement and corresponding filter.

The flat RDF* annotation style needs an additional *distinct* keyword in the select clause for the special case that triples are deleted and then re-inserted at a later point in time. In this case the condition $?valid\_from < ?tsBGP = system\_timestamp$ would be satisfied by both – the firstly inserted and the re-inserted triple where the valid_from timestamp is attached. These would then be joined with the triple with the artificial deletion timestamp annotation, according to the nested triple statements in Listings 9 and 11 and hence retrieve duplicates. With *distinct* we remove the duplicates from the result set.

Listing 9: Querying a specific version of the flat RDF* annotated dataset - one data triple statement

```
SELECT distinct ?s ?p ?o {
    <<?s ?p ?o>> vers:valid_from ?
        valid_from_0 .
    <<?s ?p ?o>> vers:valid_until ?
        valid_until_0 .

    filter(?valid_from_0 <= ?tsBGP && ?tsBGP
        < ?valid_until_0)
    bind("<timestamp>"^^xsd:dateTime as ?
        tsBGP)
}
```

Listing 10: Querying a specific version of the hierarchical RDF* annotated dataset - one data triple statement

```
SELECT ?s ?p ?o {
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from_0>> vers:valid_until ?
        valid_until_0.

    filter(?valid_from_0 <= ?tsBGP && ?tsBGP
        < ?valid_until_0)
    bind("<timestamp>"^^xsd:dateTime as ?
        tsBGP)
}
```

Listing 11: Querying a specific version of the flat RDF* annotated dataset - sequence path

```
SELECT distinct ?s ?p ?o {
    <<?s ?p ?o>> vers:valid_from ?
        valid_from_0 .
    <<?s ?p ?o>> vers:valid_until ?
        valid_until_0 .
    <<?o ?p2 ?o2>> vers:valid_from ?
        valid_from_1 .
    <<?o ?p2 ?o2>> vers:valid_until ?
        valid_until_1 .

    filter(?valid_from_0 <= ?tsBGP && ?tsBGP
        < ?valid_until_0)
    filter(?valid_from_1 <= ?tsBGP && ?tsBGP
        < ?valid_until_1)
    bind("<timestamp>"^^xsd:dateTime as ?
        tsBGP)
}
```

Listing 12: Querying a specific version of the hierarchical RDF* annotated dataset - sequence path

```
SELECT ?s ?p ?o {
    <<<<?s ?p ?o>> vers:valid_from ?
        valid_from_0>> vers:valid_until ?
        valid_until_0.
    <<<<?o ?p2 ?o2>> vers:valid_from ?
        valid_from_1>> vers:valid_until ?
        valid_until_1.

    filter(?valid_from_0 <= ?tsBGP && ?tsBGP
        < ?valid_until_0)
    filter(?valid_from_1 <= ?tsBGP && ?tsBGP
        < ?valid_until_1)
    bind("<timestamp>"^^xsd:dateTime as ?
        tsBGP)
}
```

## 4. Evaluation

We employ parts from the BEAR RDF Archiving Benchmark framework [4] to evaluate query and storage performance of flat and hierarchical RDF* annotation styles and compare them with named graphs. BEAR encompasses materialization-, version- and delta queries, three datasets, namely, BEAR-A - Dynamic Linked Data, BEAR-B - DBPedia Live, and BEAR-C - Open Data Portals and the three common RDF archiving policies, which are Indepen-

Table 5
Result sets in flat RDF* annotation style from three queries based on different timestamps from our running example

| <<Subject Predicate Object>> | Predicate | Object | Query |
|---|---|---|---|
| << <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" >> | vers:valid_from | "2021-04-07T12:00:00.000+00:00" ^^xsd:date | Query1 |
| << <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" >> | vers:valid_until | "2021-06-02T12:00:00.000+00:00" ^^xsd:date | |
| << <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder, autosomal dominant 59" >> | vers:valid_from | "2021-06-02T12:00:00.000+00:00" ^^xsd:date | Query2 |
| << <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder, autosomal dominant 59" >> | vers:valid_until | "2022-03-01T12:00:00.000+00:00" ^^xsd:date | |
| Ø | Ø | Ø | Query3 |

dent Copies (IC), Change-based- (CB) and timestamp-based (TB) versioning. From the viewpoint of Data Citation only materialization queries and timestamp-based policies are relevant, which is our focus of this paper.
BEAR performs their evaluation on the Jena TDB and HDT triple stores. We replace HDT with GraphDB in order to have two triple stores with RDF* support.

### *4.1. Experiment settings*

We build our evaluation framework on top of OSTRICH[9] which is a more stable fork of the original BEAR repository on Github and a synonym for a more recent publication featuring BEAR as evaluation framework [23]. We refactored the Java project used for the evaluation of Jena TDB so that it can be extended by other triple store vendors. Then we integrated the evaluation of GraphDB into this project as its developers strongly promoted RDF* and SPARQL* as feature. We automated the repository creation and data load step with the aim to make our experiment reproducible in fewer manual steps.
From the BEAR homepage[10] we download the hourly BEAR-B (=DBPedia live) dataset, which is versioned via named graphs, to use it for the evaluation of the TB policy without any further pre-processing. However, in order to evaluate the flat and hierarchical RDF* TB versioning approach we first need to construct such datasets. We start with computing changesets from BEAR-B's 1299 ICs. Next, we create two RDF*-based TB datasets from the initial IC and the change sets. A version is now a subset of triples which share a common timestamp. These timestamps do not reflect the original timestamps but are artificially constructed with one second increment between consecutive versions. We annotate triples with their version's timestamp as outdated (=valid_until) if they cease to exist in the original IC of that version. The difference between these two datasets solely lies in the representation of timestamped RDF* triples (see Section 3.2). We serialize each dataset as .ttl file, despite the representations being

[9]https://github.com/rdfostrich/BEAR
[10]https://aic.ai.wu.ac.at/qadlod/bear.html

Table 6
Result sets in hierarchical RDF* annotation style from three queries based on different timestamps from our running example

| <<Subject Predicate Object>> | Predicate | Object | Query |
|---|---|---|---|
| <<<< <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" >> vers:valid_from "2021-04-07T12:00:00.000+00:00" ^^xsd:date >> | vers:valid_until | "2021-06-02T12:00:00.000+00:00" ^^xsd:date | Query1 |
| <http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder, autosomal dominant 59" >> vers:valid_from "2021-06-02T12:00:00.000+00:00" ^^xsd:date >> | vers:valid_until | "2022-03-01T12:00:00.000+00:00" ^^xsd:date | Query2 |
| Ø | Ø | Ø | Query3 |

closer to n-triples due to the redundancy. However, not all triple stores we use are able to import RDF* datasets serialized as .nt.

To start the evaluation process we use a shell script similar as in BEAR and OSTRICH. We configure the script so that it executes the original two sets of BEAR-B lookup queries (62 in total) against each version of the dataset, for each TB policy and for each triple store in a Docker container. Prior to query execution the Java application creates the given repository, loads the given dataset and records ingestion time, raw dataset file size and the size of the repository including its indexes. The outputs are csv files with descriptive statistics of the query execution time for each of the TB policies, triple stores and dataset versions.

We run our evaluation on a Lenovo T14s notebook with a AMD Ryzen 7 PRO 4750U with Radeon Graphics processor. The processor uses 16 CPUs with 1400 MHz clock frequency.

### *4.2. Data ingestion and storage consumption*

We document the data ingestion time and storage consumption in Table 7. In terms of raw file size, the RDF* annotated datesets are compressed down to 113 MB (85:1) and 83 MB (116:1) from the uncompressed 9630 MB named graphs dataset. For the corresponding triple store files including indexes there is a smaller but still a notable difference between RDF* and named graphs with regards to storage consumption. RDF* triple store files occupy 173-256MB (flat) and 183-267MB (hierarchical) of storage, whereas triple stores loaded with the named graphs datasets need 1,282-7,129MB. This translates into compression factors of 26.7 - 27.85 for Jena TDB and 7.0 - 7.41 for GraphDB. This difference in compression between the triple stores is truly peculiar, showing that implementations for named graphs indexing can differ vastly. The opposite can be observed with the RDF* store files as they need more storage space than their corresponding raw files in either triple store, but the storage consumption stays comparable between the triple stores. The storage consumption scaling factors for Jena TDB are 2.265 (flat RDF*) and 3.21 (hierarchical RDF*) and the factors for GraphDB are 1.53 (flat RDF*) and 2.20 (hierarchical RDF*). Jena TDB outperforms GraphDB with ingestion time of the RDF* datasets as GraphDB takes twice as much time to ingest them.

### *4.3. Query Performance*

We follow the BEAR approach of measuring query performance. BEAR provides two query sets for their BEAR-B dataset – one set with 49 p-lookup-queries and one set with 13 predicate-object-lookup-queries. These lookup

Table 7
Performance measures for three TB policies and Jena TDB as triple store

| Triple Store | Policy | Raw file size in MB | Triple store dataset size in MB | Mean ingestion time in sec |
|---|---|---|---|---|
| Jena TDB | Named graphs | 9630 | 7129 | 601,5 |
| | RDF* flat | 113 | 256 | 10 |
| | RDF* hierarchical | 83 | 267 | 10 |
| GraphDB | Named graphs | 9630 | 1282 | 569 |
| | RDF* flat | 113 | 173 | 20,5 |
| | RDF* hierarchical | 83 | 183 | 21 |

queries consist of one triple statement and simply project the statement's subject, predicate and object variables. The prefix in BEAR's lookup query naming convention (e.g. p or po) determines which variables are bound to a specific resource IRI. This means that predicate-lookup-queries result sets are, in general, bigger than po-lookup-queries result sets and we expect a better performance with former ones.
In Figure 1 we show the query performance for the three evaluated TB approaches with Jena TDB. We aggregate the query execution time on version and query set level and use the arithmetic mean as measure. As the performance for each approach is stable across all versions we can say that it is version independent. Live version lookups are as fast as the ones for historic versions. If we compare the performance of each versioning approach we can easily spot that named graphs excel the two RDF* approaches. To put this into numbers, named graphs are approximately 7-12x faster than hierarchical RDF* and 8-14x faster than flat RDF*. These figures do not repeat itself with GraphDB. In Figure 2 we see how queries are, in general, significantly faster: for most versions the average query time falls below 1 ms. the performance relation between the three approaches still holds, meaning that named graphs perform best, followed by hierarchical RDF* and flat RDF*. However, queries against datasets based on named graphs versioning perform less than 2x better than timestamped queries against the RDF*-versioned datasets.

## 5. Conclusion and Discussion

In this paper we designed an RDF* and SPARQL* framework for timestamp-based versioning of RDF datasets. The framework includes the usual update statements (insert, update, delete) in a timestamped manner, timestamped version materialisation queries, construction of an RDF* dataset from an RDF dataset and two possible ways (=flat and hierarchical RDF* annotation styles) how to annotate and store data triples with a creation and deletion timestamp.
We employed the BEAR framework to evaluate our solution and used their via named graphs versioned BEAR-B dataset (=DBPedia live hourly snapshots) as baseline. We constructed two information equivalent RDF* datasets – one in each annotation style – from the BEAR-B independent copies. We achieved a compression down to approximately 1% of the baseline dataset's file size. During the evaluation run these all-together three datasets where loaded into Jena TDB and GraphDB triple store repositories. While former triple store was already present in BEAR we added GraphDB as additional triple store. The resulting RDF* store files including indexes consumed significantly less storage than the baseline triple store files in either triple store.
To evaluate the query performance we took the two BEAR-B query sets where one contains p-lookup-queries and the other contains po-lookup-queries as a basis. Out of these we created timestamped queries separately for each dataset and version. We made a cartesian product of queries, versions, datasets and triple stores and measured the time of each timestamped query execution. The results suggest that query performance is independent of the queried version and that, absolutely speaking, timestamped queries perform best against datasets versioned via named graphs, followed by hierarchical RDF* and flat RDF* datasets. However, in the context of Data Citation where retrieving a

specific dataset version via a SPARQL query is not time critical we suggest to to employ our RDF*-based solution to version RDF datasets. We also argue that the difference in the dataset and repository size between RDF* and named graphs is another strong support for our suggestion.

While the hierarchical RDF* annotation style does yield a better performance than its *flat* counterpart it is not equally treated by currently available RDF* stores [24]. Especially, not all of the current RDF* systems can handle arbitrary nesting levels. This might be irrelevant for a stable production system but it should be considered when migrating data to another RDF* store. Using the proposed approach, even large scale and highly dynamic RDF datasets can be efficiently versioned, allowing reproducible research, traceability and re-evaluation based on earlier states of a data set for any arbitrary subgraphs identified via RDF* queries. The triple store storage consumption is relatively small (7x-27.85x compression) compared to named graphs, while query execution times remain generally low (< 250s per query).

As we expect the Linked Data community to increasingly adopt RDF* for a wide range of applications we want to re-evaluate our work with other triple stores which are yet to become RDF* stores. This primarily includes extending our fork of the BEAR framework but also looking into new frameworks which encompass more complex queries. Furthermore, we want to evaluate new datasets and queries from real world applications, such as curated ontologies like the Uniprot Ontology from our running example. This way we hope to engage into interesting projects with ontology curators and RDA DCWG recommendation adopters across different domains where StarVers would with our help become part of the adoption story.

Once the turtle-star RDF serialization format becomes widely adopted we will fit our datasets into this format to even further reduce the raw file size and possibly the triple store files.

## References

[1] A.J. Hey, S. Tansley, K.M. Tolle et al., *The fourth paradigm: data-intensive scientific discovery*, Vol. 1, Microsoft research Redmond, WA, 2009.

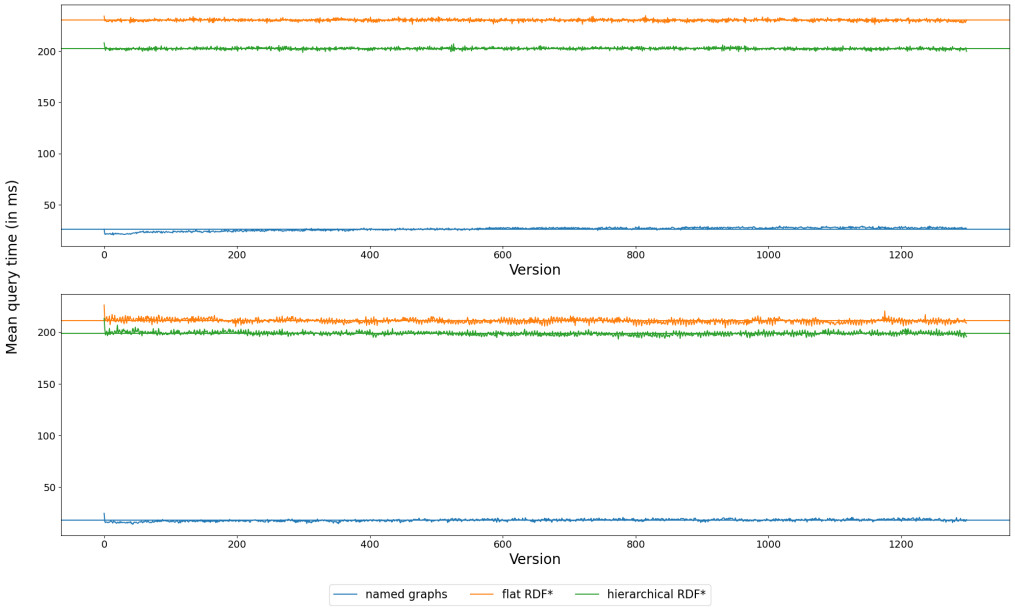


Fig. 1. Jena TDB p-lookup-query (upper) and po-lookup-query (lower) performance for the three TB policies

[2] A. Rauber, A. Asmi, D. Van Uytvanck and S. Proell, Identification of reproducible subsets for data citation, sharing and re-use, *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation* **12**(1) (2016), 6–15.
[3] A. Rauber, B. Gößwein, C.M. Zwölf, C. Schubert, F. Wörister, J. Duncan, K. Flicker, K. Zettsu, K. Meixner, L.D. McIntosh et al., Precisely and Persistently Identifying and Citing Arbitrary Subsets of Dynamic Data (2021).
[4] J.D. Fernández, J. Umbrich, A. Polleres and M. Knuth, Evaluating query and storage strategies for RDF archives, *Semantic Web* **10**(2) (2019), 247–291.
[5] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis and G. Roussakis, Versioning for Linked Data: Archiving Systems and Benchmarks., *BLINK@ ISWC* **1700** (2016).
[6] O. Pelgrin, L. Galárraga and K. Hose, Towards fully-fledged archiving for RDF datasets, *Semantic Web* (2020), 1–24.
[7] T. Neumann and G. Weikum, x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases, *Proceedings of the VLDB Endowment* **3**(1–2) (2010), 256–263.
[8] S. Gao, J. Gu and C. Zaniolo, RDF-TX: A Fast, User-Friendly System for Querying the History of RDF Knowledge Bases., in: *EDBT*, 2016, pp. 269–280.
[9] J. Anderson and A. Bendiken, Transaction-Time Queries in Dydra., *MEPDaW/LDQ@ ESWC* **1585** (2016), 11–19.
[10] A. Cerdeira-Pena, A. Farina, J.D. Fernández and M.A. Martínez-Prieto, Self-indexing rdf archives, in: *2016 Data Compression Conference (DCC)*, IEEE, 2016, pp. 526–535.
[11] O. Hartig, Foundations of RDF* and SPARQL*: (An Alternative Approach to Statement-Level Metadata in RDF), in: *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web 2017 (Vol. 1912)*, 2017. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1141963&dswid=549.
[12] M. Völkel and T. Groza, SemVersion: An RDF-based ontology versioning system, in: *Proceedings of the IADIS international conference WWW/Internet*, Vol. 2006, Citeseer, 2006, p. 44.
[13] D.-H. Im, S.-W. Lee and H.-J. Kim, A version management framework for RDF triple stores, *International Journal of Software Engineering and Knowledge Engineering* **22**(01) (2012), 85–106.
[14] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens and R. Van de Walle, R&Wbase: git for triples, in: *LDOW*, 2013.
[15] M. Graube, S. Hensel and L. Urbas, R43ples: Revisions for triples, in: *Proceedings of the 1st Workshop on Linked Data Quality co-located with 10th International Conference on Semantic Systems (SEMANTiCS 2014)*, Citeseer, 2014.
[16] N. Arndt, P. Naumann, N. Radtke, M. Martin and E. Marx, Decentralized Collaborative Knowledge Management Using Git, *Journal of Web Semantics* **54** (2019), 29–47, Managing the Evolution and Preservation of the Data Web. doi:https://doi.org/10.1016/j.websem.2018.08.002. https://www.sciencedirect.com/science/article/pii/S1570826818300416.
[17] A. Rauber, A. Asmi, D. Van Uytvanck and S. Proell, Identification of reproducible subsets for data citation, sharing and re-use, *Bulletin of IEEE Technical Committee on Digital Libraries, Special Issue on Data Citation* **12**(1) (2016), 6–15.

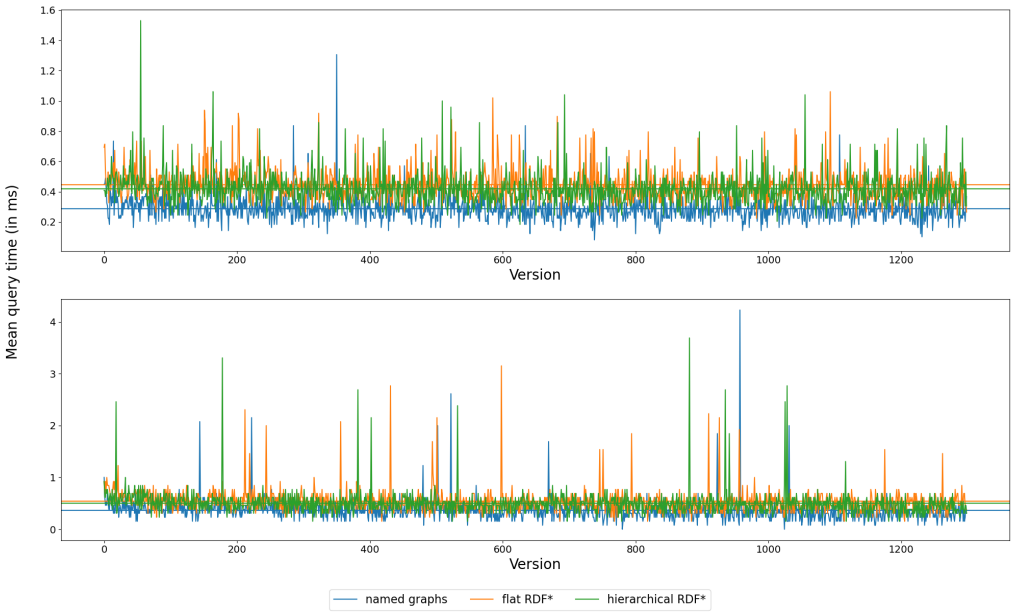


Fig. 2. GraphDB p-lookup-query (upper) and po-lookup-query (lower) performance for the three TB policies

[18] S. Sen, M.C. Malta, B. Dutta and A. Dutta, State-of-the-art approaches for meta-knowledge assertion in the web of data, *IETE Technical Review* **38**(6) (2021), 672–709.
[19] F. Orlandi, D. Graux and D. O'Sullivan, Benchmarking RDF Metadata Representations: Reification, Singleton Property and RDF, in: *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, IEEE, 2021, pp. 233–240.
[20] D. Graux, F. Orlandi, T. Kaushik, D. Kavanagh, H. Jiang, B. Bredican, M. Grouse and D. Geary, Timelining Knowledge Graphs in the Browser, in: *VOILA! 2021-6th International Workshop on the Visualization and Interaction for Ontologies and Linked Data*, 2021.
[21] J. Eriksson and A. Hakim, Format Conversions and Query Rewriting for RDF* and SPARQL, 2018.
[22] T. Delva, J. Arenas-Guerrero, A. Iglesias-Molina, O. Corcho, D. Chaves-Fraga and A. Dimou, RML-star: A declarative mapping language for RDF-star generation, in: *ISWC2021, the International Semantic Web Conference*, 2021, pp. 1–5.
[23] R. Taelman, M. Vander Sande and R. Verborgh, OSTRICH: versioned random-access triple store, in: *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 127–130.
[24] F. Orlandi, D. Graux and D. O'Sullivan, How many stars do you see in this constellation?, in: *European Semantic Web Conference*, Springer, 2020, pp. 175–180.
[25] D.D. Nart, D. Degl'Innocenti and M. Peressotti, Well-Stratified Linked Data for Well-Behaved Data Citation, *CoRR* **abs/1512.02898** (2015). http://arxiv.org/abs/1512.02898.
[26] S. Gupta, C. Zabarovskaya, B. Romine, D.A. Vianello, C.H. Vitale and L.D. McIntosh, Incorporating Data Citation in a Biomedical Repository: An Implementation Use Case, *AMIA Summits on Translational Science Proceedings* **2017** (2017), 131.
[27] J.P. James Duncan, Ecosystem Monitoring Collaborator Network.
[28] C.R. Reyna Jenkyns Melissa Cuthill, Ocean Network Canada.
[29] C.M. Zwölf, N. Moreau and M.-L. Dubernet, New model for datasets citation and extraction reproducibility in VAMDC, *Journal of Molecular Spectroscopy* **327** (2016), 122–137.
[30] C. Schubert and H. Bamberger, Handling Continuous Streams for Meteorological Mapping, in: *Service-Oriented Mapping*, Springer, 2019, pp. 251–268.
[31] B. Gößwein, T. Miksa, A. Rauber and W. Wagner, Data identification and process monitoring for reproducible earth observation research, in: *2019 15th International Conference on eScience (eScience)*, IEEE, 2019, pp. 28–38.
[32] M. Klein, D. Fensel, A. Kiryakov and D. Ognyanov, Ontoview: Comparing and versioning ontologies, in: *Collected Posters of First Int. Semantic Web Conf.(ISWC 2002)*, 2002.
[33] H. Kondylakis, M. Despoina, G. Glykokokalos, E. Kalykakis, M. Karapiperakis, M.-A. Lasithiotakis, J. Makridis, P. Moraitis, A. Panteri, M. Plevraki et al., EvoRDF: A framework for exploring ontology evolution, in: *European Semantic Web Conference*, Springer, 2017, pp. 104–108.
[34] J.J. Carroll, C. Bizer, P. Hayes and P. Stickler, Named graphs, *Journal of Web Semantics* **3**(4) (2005), 256, World Wide Web Conference 2005——Semantic Web Track. doi:https://doi.org/10.1016/j.websem.2005.09.001. https://www.sciencedirect.com/science/article/pii/S1570826805000235.
[35] V. Nguyen, O. Bodenreider and A. Sheth, Don't like RDF Reification? Making Statements about Statements Using Singleton Property, in: *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 759–770–. ISBN 9781450327442. doi:10.1145/2566486.2567973.
[36] B. Kasenchak, A. Lehnert and G. Loh, Use Case: Ontologies and RDF-Star for Knowledge Management, in: *The Semantic Web: ESWC 2021 Satellite Events*, R. Verborgh, A. Dimou, A. Hogan, C. d'Amato, I. Tiddi, A. Bröring, S. Mayer, F. Ongenae, R. Tommasini and M. Alam, eds, Springer International Publishing, Cham, 2021, pp. 254–260. ISBN 978-3-030-80418-3.
[37] R. Keskisärkkä, E. Blomqvist, L. Lind and O. Hartig, RSP-QL*: Enabling Statement-Level Annotations in RDF Streams, in: *Semantic Systems. The Power of AI and Knowledge Graphs*, Springer International Publishing, Cham, 2019, pp. 140–155. ISBN 978-3-030-33220-4.
[38] M. Wudage Chekol, G. Pirrò and H. Stuckenschmidt, Fast interval joins for temporal SPARQL queries, in: *Companion Proceedings of The 2019 World Wide Web Conference*, 2019, pp. 1148–1154.
[39] J. Frey and S. Hellmann, MaSQue: An Approach for Flexible Metadata Storage and Querying in RDF., in: *SEMANTICS Posters&Demos*, 2017.
[40] O. Bruns, T. Tietz, M. Vafaie, D. Dessí and H. Sack, Towards a representation of temporal data in archival records: Use cases and requirements, in: *Proceedings of the International Workshop on Archives and Linked Data*, 2021.
[41] T. Baccaert, Querying Linked Data with High Availability (2021).
[42] N. Lasolle, O. Bruneau, J. Lieber and L. Rollet, Temporal Knowledge Representation for Historical Corpora: Application to the Henri Poincaré Correspondence Corpus (2021).
[43] S. Govindapillai, L.-K. Soon and S.-C. Haw, An empirical study on Resource Description Framework reification for trustworthiness in knowledge graphs, *F1000Research* **10** (2021).