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

Filip Kovacevic [a,*], Fajar J. Ekaputra [c,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, rauber@ifs.tuwien.ac.at
[b] *SBA Research, TU Wien, Vienna, Austria*
E-mail: miksa@ifs.tuwien.ac.at
[c] *Institute of Data, Process and Knowledge Management, WU Wien, Vienna, Austria*
E-mail: fajar.ekaputra@wu.ac.at

Abstract. In the era of data-driven research, we are confronted with the challenge of preserving datasets utilized in experiments for an extended duration. As a result of unretrievable datasets, research cannot be reproduced nor verified. One of the specific challenges is the preservation of the history of evolving datasets. To tackle this challenge, we nowadays use versioning mechanisms on datasets so that each version or revision can be identified. The retrieval of specific versions usually requires queries to be enhanced with some form of version identifiers in order to target specific snapshots. What these version identifiers look like depends on the versioning policy. In timestamp-based policies we would use timestamps in order to retrieve datasets as they were at a specific point in time. The implementation of such policies again depends on the type of data and more importantly on the database system. In the context of RDF data, researchers have contributed with countless temporal RDF models and RDF versioning systems over the last two decades. Many well-known RDF metadata representation models, such as named graphs, reification, n-ary and singletons have simply been applied to represent temporal metadata. However, only little reserach has been done in this direction with one of the more recent RDF extensions with the capability of representing metadata, namely RDF-star. In this paper, we explore the possibilities of RDF-star as basis of a timestamp-based versioning framework. We show how temporal metadata can be represented by utilizing RDF-star's nested triples and stored within RDF-star stores. Moreover, we develop and showcase timestamp-based SPARQL-star templates that can be used to 1) transform RDF datasets into RDF-star datasets 2) update and thereby evolve RDF dtasets and 3) query RDF subsets of time-specific snapshots. We also explain our utilization of the SPARQL query algebra for the purpose of translating SPARQL queries into SPARQL-star queries and link our python-based API that is capable of automatically generating and executing latter queries. Finally, we evaluate our work with datasets and queries from the BEAR benchmark and two RDF stores, namely, Jena TDB2 and GraphDB. Our results suggest that our solution is preferable if the frequency and manner of dataset evolution are uncertain, implying that it outperforms the baseline approaches in sum. However, in cases where datasets are prone to high change rates between consecutive updates our approach does not outperform the baseline approaches in terms of storage consumption. Moreover, the choice of RDF store is significant as the query performance differs vastly between the evaluated RDF-star stores. Our reproducible evaluation process is available on: https://github.com/GreenfishK/starvers_eval

Keywords: RDF, RDF-star, SPARQL, SPARQL-star, Versioning, RDF Store, Data Citation, GraphDB, Jena TDB2

[*]Corresponding author. E-mail: filip.kovacevic@tuwien.ac.at.

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 are even more so met with more structured and machine-readable publications of datasets and research artifacts, such as the ones that have been driven by the Linked Open Data (LOD) movement and described using the Resource Description Framework (RDF). LOD has lately transitioned into a new core technology called Knowledge Graphs (KGs). Late advances in knowledge graphs from the scholarly domain have made it possible to publish discourse elements, such as problem statements, methods, models, results and conclusions in a highly structured and machine-readable way [2, 3]. This enables a whole new set of automatizable applications, ranging from the identification of research trends, over visualization of the evolution history up to comparing, replicating and reproducing research results. It is apparent that these applications require not just the most recent snapshot of data but the whole traceable history. As these technologies are open and community-driven, changes to the data typically happen without any warning [4]. Therefore, we need to ensure that, whenever changes occur, the previous state of data can still be retrieved. In fact, we aim to make any arbitrary subset, that can be described by a query, retrievable as it was at a certain point in time. Our work focuses on SPARQL queries and RDF stores, as these are the core technologies for querying and storing RDF data, i.e. knowledge graphs and their schemata (ontologies). In this paper, we present StarVers – a solution for timestamp-based versioning of RDF data.

The remainder of the paper is structured as follows: In Section 2 we explain the background of our work and what motivated us in first place to experiment with RDF datasets and build the proposed framework. Next, we move on to preliminaries (Section 3) that every reader should have a fundamental understanding of in order to follow our paper more easily. In this Section we also provide examples that should prepare the reader for most of our SPARQL-based queries and update statements in Section 5. Before we get to latter section, we discuss what has already been proposed in the literature and in which intersection we see our work. Then, we introduce and elaborate on our framework for timestamped-based versioning of RDF data, which is the main contribution of our work. In Section 6 we explain our evaluation process including the BEAR benchmark framework and show the results as plots of query & update performance as well as data ingestion and storage consumption. We conclude our work in Section 7 and discuss drawbacks and limitations. Last, we report how we want to extend our work with new features in the near-term.

2. Background and Motivation

The RDA Data Citation Recommendations [5] introduced by the Research Data Alliance Data Citation Working Group aims at improving the reproducibility and support the data re-use by tackling the issue of dynamic and evolving datasets. The recommendations are based upon versioned data, timestamping and a query subsetting mechanism. A recent survey [6] 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. The first two recommendations R1 and R2 suggest that data should be versioned in a timestamp-based manner and these are, in fact, the most adopted recommendations. These recommendations, however, have not yet been discussed for RDF stores. As we came across a novel approach of annotating RDF triples, namely RDF-star, we saw the potential to pair this approach with timestamp-based versioning with the goal to precisely identify arbitrary subgraphs from evolving RDF data sets.
The motivation for this is many-fold. The reproducibility crisis taught us that reproducing and verifying research results is necessary for the scientific method and it is inevitable for assessing the validity of an experiment. In the data-intensive scientific discovery paradigm datasets are the foundation of many experiments. Being able to retrieve the same, unmodified dataset that was used in an experiment is therefore a precondition to reproducible results and citable datasets. Timestamped materialisation queries (R7) are an important means to these ends. Assigning a timestamp to a query does not only enable materializing arbitrary snapshots of a specific point in time but also serve as a

basis for computing unique dataset identifiers.
Results have sometimes to be updated as new information becomes available. Releasing new information in form of periodical snapshots of ontologies or knowledge graphs is a common practice. However, it always comes with an information gap between the snapshots which impairs downstream applications such as Semantic Data Integration, Knowledge Management, Ontology-based Reasoning and Evolutionary Analysis, in cases when they rely on a full picture of the changes. For example, analysing the popularity of specific entities in large-scale KGs by observing the frequency of mentions and identifying specific moments when the entities gained significant attention would be inaccurate or even impossible with just snapshots of data. RDF datasets on different scales could profit from a timestamp-based versioning approach as it would make their snapshot releases obsolete and bridge the information gap between them. Examples are DBPedia live[1] (large-scale) and the Uniprot ontology[2] which both release snapshots at irregular dates. What impact a timestamp-based versioning approach used in RDF-star and SPARQL-star supporting RDF stores would have on the query & update performance and storage costs with different update frequencies, sizes and change ratios is what our research aims to find out.

3. Preliminaries

The goal of this section is twofold. First, we introduce several topics that are related to our work on a fundamental level to provide context to the reader. Second, we provide examples with protruding features that become relatable in the Solution Design Section 5.

3.1. *RDF, SPARQL and RDF stores*

The Resource Description Framework (RDF)[3] is a W3C recommendation for modeling data interchange on the web. Among its most notable features are the representation of data as triples, the linkage of these triples and subsequent emergence of a graph, the use of IRIs to identify and describe entities and relations and the use of declarative semantics so that RDF reasoners can infer additional triples. RDF is a highly extensible framework which is why many extensions have been proposed since the release of version 1.0 in 2004[4]. In particular, there are extensions which allow for annotating single RDF statements with context or metadata. Named Graphs [7] is one such example of these extensions and have been integrated to the latest version of the RDF specification. More recently, RDF-star [8] has emerged as the upcoming extension to further allow for rich annotation of RDF statements, which will be discussed in more detail in Section 3.2.

SPARQL[5] is the W3C recommendation for querying RDF data. Similar to RDF, SPARQL has been extended for many purposes, such as for querying metadata annotations [8], facilitating timestamping/versioning [9–12], querying RDF data stream [13–15] and many others. Tables 1, Table 2 and Table 3 provide the definition of a set of concepts from SPARQL specifications[5,6] that are relevant for our approach and will be used in our SPARQL templates in Section 5. Within the scope of this paper, we will use the term *SPARQL statement* to collectively refer to SPARQL queries and update statements. SPARQL update statements include INSERT statements and DELETE statements. We assume that updates can only be achieved by combining DELETE and INSERT. Also, we use *SPARQL term* as a generic term for SPARQL keywords, concepts, (graph) patterns and other expressions. For chosen SPARQL term we informally describe their main characteristics and also some additional ones that aid in understanding our *SPARQL statements* later on. In the third column of the table we provide either a full SPARQL query or a snippet and highlight the relevant parts.

RDF stores are purpose-built databases for the storage and retrieval of any type of data expressed in RDF [16] and are considered as a subclass of graph-oriented DBMS [17]. RDF stores typically offer various mechanisms

[1] https://www.dbpedia.org/resources/live/
[2] https://ftp.uniprot.org/pub/databases/uniprot/previous_releases/
[3] https://www.w3.org/TR/rdf11-concepts/
[4] https://www.w3.org/TR/rdf-primer/
[5] https://www.w3.org/TR/sparql11-query/
[6] https://www.w3.org/TR/sparql11-update

for data storage, indexing, query processing including join processing and partitioning. These mechanisms are often abstracted from the user or application, e.g. the user may not be aware of whether relational storage mechanisms (Triple Tables, Vertical Partitioning, Property Tables), native graph storage mechanisms (Graph-based storage, Tensor-based storage) or other mechanisms are employed at the low-level. At the high-level RDF stores offer a middleware for user interactions, which includes client connectors, parsers, serializers and query engines [18, 19]. The terms RDF triple stores, RDF stores, or simply triple stores are sometimes interchangeably used [18, 20]. In the context of our work and for practical reasons, we use the term RDF stores to refer to systems capable of storing and processing RDF data.

3.2. RDF-star and SPARQL-star

RDF-star/SPARQL-star [8] is a general purpose statement-level annotation approach. It uses the nesting paradigm to form (multi-level) nested triples and thereby enable making statements about single triples. Let us demonstrate this paradigm SPARQL-star. If we consider following triple pattern:

$$?s\ \ ?p\ \ ?o$$

then

$$<<?s\ \ ?p\ \ ?o>>\ \ ?mp\ \ ?mo$$

is called a *quoted triple pattern*, specifically a *subject quoted triple (SQT) pattern*. Semantically, `?mp` is the annotation property and `?mo` the annotation value. The triple pattern inside the pointy brackets is called a quoted triple pattern and represents the statement that is being annotated. A quoted triple pattern can have multiple nesting levels. If we embed this triple pattern into the subject of another triple pattern, we get:

$$<<<<?s\ \ ?p\ \ ?o>>\ \ ?mp\ \ ?mo>>\ \ ?mmo\ \ ?mmo$$

which is referred to as a *nested quoted triple in subject position (NQT-SP) pattern*. In the scope of this work, we will refer to the inner-most triple (pattern) as *data triple (pattern)*. Logically, either triple pattern can be seen as a metadata triple pattern. Especially, the NQT-SP pattern can be interpreted as a *data triple pattern* with two metadata annotation patterns. Analogously, the same syntax can be applied to RDF-star, i.e. we can quote and nest triples in an RDF-star dataset the same way.

RDF-star/SPARQL-star has been endorsed by W3C and is on its way to become a W3C standard[7]. It is already part of popular RDF store technologies, such as GraphDB, Jena TDB2 and Stardog. 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 [21]. Moreoever, RDF-star is already applied in the YAGO4 knowledge graph[8] to represent temporal facts and authors in [22] show how to query them via SPARQL-star. RDF-star converters from RDF [23] and heterogeneous [24] sources have also been proposed.

3.3. SPARQL algebra

SPARQL queries can be translated into a unary algebra expression tree, as shown in [25]. This relational algebra was further endorsed by W3C[9](also see 18.2 and 18.5 of SPARQL 1.1 Query Language[5]) and is also part of Python's RDF API rdflib[10]. Such a well-defined grammar comes in handy when we want to manipulate SPARQL queries. Instead of using regular expressions directly on SPARQL queries, one can simply operate on the query tree to add, delete or modify any expression within the query. This, of course, requires one SPARQL-to-Algebra and one

[7]https://www.w3.org/2021/12/rdf-star.html
[8]https://yago-knowledge.org/downloads/yago-4
[9]https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html
[10]https://rdflib.readthedocs.io/en/stable/apidocs/rdflib.plugins.sparql.html#module-rdflib.plugins.sparql.algebra

Table 1
Selected SPARQL concepts

Concept	Description	Example
Triple Pattern	A triple pattern is a pattern in the form of $(I \cup V \cup L \cup B)x(I \cup V)x(I \cup V \cup L \cup B)$ where I is an IRI, V a variable, L a literal and B a Blank Node. Literals are practically not used in the subject position but per definition allowed. In our example we use a triple pattern in the form of V x I x L .	`?person <http://example.com/hasName> "Albert" .`

Projection and Projection variables (PV)	Projection variables are variables that are bound in the query and they determine the "header of the result set" as part of the SELECT clause. Only variables that are bound can be projected. There can be multiple sets of PVs in a query, e.g. one projected by the subquery and one projected by the outer query. An asterisk in the SELECT clause projects all bound variables that are within the scope of that query.	`SELECT ?a ?b ?c { ?a ?b ?c }`
		
		`SELECT * { ?a ?b ?c . }`

Bindings	A 'binding' is a pair (variable, RDF term). We say that a variable is bound to an RDF term (not the other way around). Using bindings in the plural form we refer to the tabular result set returned by the query (e.g as JSON). A variable can also be unbound, which translates to an empty cell within the result set. A new binding can also be directly introduced in the query, e.g. to enrich the dataset with additional labels (see Basic graph patterns example).	`"bindings": [`
		`{`
		`"first_name": { "type": "literal" , "value": "Barack" } ,`
		`"last_name": { "type": "literal" , "value": "Obama" }`
		`} ,`
		`{ "first_name": { "type": "literal" , "value": "Donald" } }]`

Prologue and namespaces	A prologue is a set of namespace prefix bindings which can also be empty (optional). It is always at the top of every *SPARQL statement*. In our example we have a prefix xsd that binds to the common `<http://www.w3.org/2001/XMLSchema#>` namespace. The prefix can be seen as an abbreviation for the namespace so that it is easier to refer to in the query. The default namespace is denoted by a colon. For the remainder of the paper we assume <http://example.com/> to be our default namespace.	`PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>`
		`PREFIX : <http://example.com/>`
		
		`Select * { ...}`

Basic graph pattern (BGP)	A Basic Graph Pattern is a set of Triple Patterns. There can be more than one BGP in a query and they can occur at different nesting levels. Among other operations, we can perform JOIN and UNION on BGPs. This allows us to imagine queries like m-ary trees with BGPs as the nodes and JOINs as edges. Two BGPs are joined by their intersecting named variables. So, in fact, we are joining individual triple patterns where the variable names match.
	In our example, we first bind a new variable ?size in either of the inner BGPs to categorize each subset. Then, the two inner BGPs are combined and finally joined with the outer BGP by ther common variable ?s.
		`SELECT ?page ?size`
		`{`
		`?s foaf:page ?page .`
		`{`
		`?s rdfs:label "Microsoft"@en .`
		`BIND ("Large corporate" as ?size)`
		`}`
		`UNION`
		`{`
		`?s rdfs:label "Ubitech"@en .`
		`BIND ("SME" as ?size)`
		`}`
		`}`

Property paths (fixed length)	A property path is a possible route through a graph between two graph nodes. Most notable are Sequence Paths, which is a linked list of one or more properties and Alternative Paths, which represents the logical OR or alternative sequences. The length of the path is determined by the list of properties. These property paths can be resolved to alternative forms. The example shows two identical queries, once written using property paths and once in its alternative form. Nodes will be matched that are either connected by foaf:knows/foaf:name (first sequence) or foaf:knows/foaf:full_name (second sequence).	`SELECT * {`
		`?x foaf:knows / (foaf:name | foaf:full_name) ?name .`
		`}`
		
		`SELECT *`
		`WHERE {`
		`?x foaf:knows ?friend .`
		`{ ?friend foaf:name ?name .}`
		`UNION`
		`{ ?friend foaf:full_name ?name . }`
		`}`

Property paths (arbitrary length)	A multipath or path of arbitrary length is a path the length of which is determined by the available links in the data. Thus, the path is only determined during query execution. This implies that the number of joined triple patterns is unknown prior to query execution. We use the same quantifiers as for regular expressions, namely, *, + and ? to denote how often a property can be consecutively repeated in a path. In our example we retrieve the names of all people that can be reached from Albert.	`Select * {`
		`?x foaf:name "Albert" .`
		`?x foaf:knows+/foaf:name ?name .`
		`}`

Join	Every Triple Pattern within a BGP can be considered as a join where the keys are the common subject or object variables between the statements.	`Select ?a ?b ?c {`
		`?a ?b ?c .`
		`?c ?b2 ?c2 .}`

Table 2

Selected SPARQL terms that are common in other query query languages

Keyword	Description	Example
SELECT	The SELECT marks the beginning of a SELECT-query and is followed by a set of projection variables that are included in the result set.	`SELECT ?name ?address { ... }`

WHERE	They WHERE keyword marks the beginning of the query body where Triple Patterns, filters and a whole set of other SPARQL operations follow. In SPARQL, the WHERE keyword can be omitted and the beginning of the query body is purely marked by an open curly bracket.	`SELECT ?name ?address WHERE { ... }`
		or
		`SELECT ?name ?address { ... }`

Graph update operations: INSERT and DELETE	In SPARQL we can use graph update operations to insert and delete data in two ways. With INSERT/DELETE DATA we plainly provide a set of RDF triples that should be inserted/deleted. If we want to link new data to already existing data in the graph, we can pair an INSERT/DELETE-body with a WHERE-clause. The set of bindings matched by triple patterns in the WHERE-clause can then be accessed in the graph update operation blocks to link new resources or delete the matched triples. These blocks can also be combined and used in the same statement.
	In our example we first query for a company with the label "Microsoft" in the WHERE clause. Then we use the DELETE block to delete the company's label followed by an INSERT-block where we add a new label and link a webpage to the company. The deletion and insertion of the label can also be seen as an update of the matched triple's object.
		`DELETE`
		`{`
		`    ?company rdfs:label "Microsoft" .`
		`}`
		`INSERT`
		`{`
		`    ?company :has_page <https://www.`
		`        microsoft.com/> .`
		`    ?company rdfs:label "Microsoft Corp."`
		`        .`
		`} WHERE {`
		`    ?company rdf:type :Company .`
		`    ?company rdfs:label "Microsoft" .`
		`}`

IF	As in many programming languages we can use the IF functional form to define what should happen when a condition is met and what should happen alternatively. Unlike in other languages, IF cannot be used as a "stand alone" construct but is used within other SPARQL terms. As an example, we can use this functional from together with BIND and BOUND to set default values for otherwise empty cells in a result set.	See example in Table 3 for BOUND.

CONCAT	With CONCAT we can concatenate string literals to form new string literals.
	We can e.g. use them in FILTER expressions to reduce the number of logical functions and make the comparison more concise and easier to read.
		`SELECT * WHERE {`
		`    ...`
		`    FILTER (CONCAT(?firstName, " ",`
		`        ?familyName) = "John Doe")`
		`}`

Algebra-to-SPARQL translator, which again, works with regex if we take rdflib's implementation as example. The advantage, however, for designing solutions which require us to perform modifications on a query is that the modifications can be well-defined, as well. Moreover, with an unary tree we can more easily illustrate query operation patterns.

In Listings 1 and 2 we show the SPARQL query from the "fixed length property paths alternative form" example in Table 1 and its algebra expression tree, respectively. From the SPARQL query we can tell that it has in total three BGPs which we also see in the query tree. Two BGPs are first combined and then joined with the third BGP. Since we use an asterisk in the SELECT clause, we want to project all variables. This is reflected in the PV set. We can see that this set occurs twice in the query tree. The first occurrence of PV specifies the variables used inside the Join operation, while the second occurrence of PV specifies the variables to be included in the final query results.

The algebra expression tree is based on rdflib's expression tree implementation with some terms changed to be closer to the W3C syntax. However, this tree does not fully reflect all W3C expressions for convenience reasons. In particular, there is no such term as "triples" in W3C's grammar. Triples can be expressed by first drilling down through a set of linked grammar symbols, i.e. GroupGraphPatternSub -> TriplesBlock -> TriplesSameSubjectPath, which makes it inconvenient for showcasing a simple example. In here, we want to draw the attention to the tree representation and its flexibility to modify queries. We could e.g. insert a new triple pattern `?x foaf:name "Albert" .` into the first BGP (p1 = BGP). This only requires us to locate the BGP inside the tree, e.g. by recursively traversing the tree, and make a new entry to the list of triples.

Listing 1: Example from Table 1 - Property paths (fixed length), 2nd query

Table 3
Selected SPARQL terms that are specific to SPARQL

Keyword	Description	Example
GRAPH	To specify specific graphs, so called named graphs, as source of information we can use the GRAPH keyword in the *SPARQL statement*. This is a.o. useful if we have an RDF dataset with multiple named graphs and we want to query from a set of named graphs that fulfill certain criteria. Our example shows a template for querying from all named graphs that are labeled with version 1. We see that we once refer to a specific named graph resource and then we use the variable ?g to refer to the set of named graphs.	`SELECT * {` `GRAPH <http://example.org/versions>` `{` `?graph <http://www.w3.org/2002/07/owl#versionInfo> 1 .` `}` `GRAPH ?graph {` `...` `}` `}`
NOW	The NOW() function returns the query execution timestamp as annotated literal using the xsd:dateTime datatype. W3C, however, does not define how this function should be implemented.	`"2023-03-01T12:00:00.000+00:02"^^xsd:dateTime`
BIND	The BIND pattern enables the creation of bindings during the execution of a *SPARQL statement*. It is useful in many cases, e.g. to rename a variable name in the result set or to create a temporary variable bound to a specific value and refer to it in the query. In the former case, the BIND keyword is not needed explicitly. For example, we can bind a new variable, namely ?executionTimestamp, to the query execution timestamp returned by the function NOW(). Then we can use this newly introduced variable in the filter condition. In the SELECT clause we can change some of the variable names to more meaningful names.	`SELECT` `?s` `(?p as ?hasTimestamp)` `(?o as ?timestamp) {` `?s ?p ?o .` `BIND(NOW() as ?executionTimestamp)` `FILTER (?o = ?executionTimestamp)` `}`
BOUND	To check whether a variable is bound, we can use the functional form BOUND. This is a.o. useful to specify default values in case a variable is unbound. Our example demonstrates how we can assign a default value "noLastName" to make up for otherwise empty cells for the variable ?lastName in the result set.	`SELECT ?firstName ?familyName {` `?person :firstName ?firstName .` `OPTIONAL {` `?person :lastName ?optionalFamilyName .` `}` `BIND((IF(!BOUND(?optionalLastName), "noLastName", ?optionalLastName)) as ?familyName)` `}`
FILTER	Within FILTERs arithmetic and regular expressions can be formulated to restrict values and exclude them from the result set. Filters can be combined with other SPARQL terms. If our data is timestamped, one exemplary usage is to filter for data as they were valid at a specific point in time. As timestamps are usually considered temporal metadata, we would often see them attached to named graphs in plain SPARQL 1.1.	`SELECT * WHERE {` `GRAPH ?g { ?s ?p ?o . }` `?g :valid_from ?valid_from .` `?g :valid_until ?valid_until` `FILTER(xsd:dateTime(?valid_from) <= "2023-02-14T00:00:00Z"8sd:dateTime && xsd:dateTime(?valid_until) > "2023-02-14T00:00:00Z"8sd:dateTime)` `}`
VALUES and UNDEF	We can provide inline data directly in the *SPARQL statement* via the VALUES clause. This clause works like many tabular data formats where we define the header row and the values as tuples inside the table body. This data can then be joined with data from a graph that the *SPARQL statement* is executed on. We can use this clause a.o. to update existing data and insert new data within the same *SPARQL statement*. In our example we update an exam question by providing a tuple of old and new question as literals. Additionally, we add a new question by leaving the first value from the tuple undefined (UNDEF) and only providing a new question.	`DELETE {?exam :hasQuestion ?question .}` `INSERT {?exam :hasQuestion ?newQuestion .}` `WHERE {` `VALUES (?question ?newQuestion) {("What is love?" "What is platonic love?")(UNDEF "What is romantic love?")}` `?exam :hasQuestion ?question .` `}`

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
    ?x foaf:knows ?friend .
    { ?friend foaf:name ?name .}
    UNION
    { ?friend foaf:full_name ?name . }
}
```

Listing 2: rdflib's algebra expression tree for the SPARQL query in Listing 1

```
SelectQuery(
  p = Project(
      p = Join(
          p1 = BGP(
              triples = [(VAR1('x'), IRIREF('http://xmlns.com/foaf/0.1/knows'), VAR1('friend'))]
              )
          p2 = Union(
              p1 = BGP(triples = [(VAR1('friend'), IRIREF('http://xmlns.com/foaf/0.1/name'), VAR1('name'))])
              p2 = BGP(triples = [(VAR1('friend'), IRIREF('http://xmlns.com/foaf/0.1/full_name'), VAR1('name'))])
              )
          )
      PV = [VAR1('x'), VAR1('friend'), VAR1('name')]
      )
  PV = [VAR1('x'), VAR1('friend'), VAR1('name')]
)
```

3.4. Versioning RDF datasets

Versioning is a means to track changes associated with dynamic data over time [11] and thereby ensure that earlier states of datasets can be retrieved [5]. Specific states of a dataset are called versions or revisions. When it comes to RDF datasets, we will find three approaches, namely, 1) independent copies (IC) or snapshots 2) change-based (CB) approaches and 3) timestamp-based (TB) approaches to be the most common ones in the literature [26], [27]. ICs are the simplest and most intuitive form of versioning as a new copy of the dataset is created to represent the updated version of the former. Obviously, this approach comes with high storage costs due to the huge redundancy with repeated copying of unchanged data.

CB approaches can vastly reduce the storage costs as they only store the differences between two versions. For CB-approaches we will most commonly find change sets, deltas and patches as interchangeably used names for these differences. In general, to reconstruct a specific version, we have to apply a chain of chronologically ordered patches either from the very first snapshot or from an intermediate snapshot. This also means that, as opposed to the IC paradigm, we have higher reconstruction costs which add up to the retrieval of a specific dataset version. Also, CB apporaches are not nateively supported by SPARQL, i.e. there is no way to retrieve and apply a number of patches to reconstruct a dataset version purely with SPARQL as it is lacking procedural features.

TB approaches assign timestamps or validity intervals to data on different granularity levels that can range from statement to dataset level so that a dataset can be retrieved as it was at a certain point in time. 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 patch up to that timestamp, which again for live data does not scale well. A trade-off between storage costs and query performance can be achieved by materializing intermediate snapshots every x changes. However, with highly dynamic data using version numbers or alphanumeric revision strings to query specific states datasets is impractical for obvious reasons. Also, we are in that case rather interested in specific time frames or time points in which a dataset was valid. With TB approaches we can use timestamps in the query to pinpoint a specific snapshot. Generally, compared to CB approaches, we logically only have to store temporal information in addition. This additional storage overhead can be reduced by applying certain compression techniques [12, 28]. Also, by implementing clever temporal indexes and partitioning methods we can achieve fast retrievals that add only little query performance overhead compared to querying isolated snapshots directly [12].

[11] https://www.ands.org.au/working-with-data/data-management/data-versioning

4. Related Work

Our work touches on both – temporal RDF models which are capable of associating triples with timestamps but also RDF archiving system which next to archiving and versioning also have multifaceted retrieval mechanisms. First, we give an overview of the former related topic where we describe basic temporal concepts that the vast majority of authors employ and subsequently elaborate on three different types of temporal RDF models. Second, we give an overview of RDF archiving system features that are commonly addressed in the literature. While we do not propose a fully-fledged RDF archiving system we do characterize our work by timestamped-based versioning, that some of the systems are based on, and materialisation queries. As our work is motivated by reproducible research results, we consider latter queries as the most important type of retrieval. Without being able to identify specific dataset versions data-intensive experiments would not be reproducible.

4.1. Temporal RDF models

Temporal metadata is a specific type of metadata [29] where researchers from semantic web communities but also other communities, such as formal logic, computational logic, mathematics, database and others have proposed many conceptual, logical and physical models for working with temporal metadata in RDF. We found a handful of surveys[30],[31],[32],[33] covering a wide range of temporal metadata models for RDF. In principal, we can divide these models into following categories: 1) Conceptual models that are not specific to RDF syntax, 2) logical models using plain RDF to add temporal annotations, 3) logical models extending RDF with additional syntax and semantics to specifically target the representation of time and 4) metadata representation models that are extensions of RDF and can be used to represent any kind of metadata, thus, also temporal metadata. Then, there are are physical models which propose specific indexes like B+trees to index temporal metadata [12],[34],[35],[36],[37]. However, our work does not propose any physical models which is why we will not discuss them in further detail.

On a conceptual level, a temporal metadata representation model (MRM) is either interval-based, point-based or both. A point-based model simply assigns a timestamp to a triple in the form of `<s> <p> <o>[ts]`. An interval-based model annotates triples with an interval in the form of `<s> <p> <o>[t1,t2]`, where t1 and t2 denote a start and an end timestamp, respectively. Most works from the literature including our work adapt – though not always exclusively – the interval-based model [10], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62]. To add meaning to these intervals, the models furthermore decide whether they use transaction time, validity time or user-defined time. The valid time of data refers to the time period during which it holds true in the modeled world, whereas the transaction time denotes the moment when the data is physically recorded in the database[39]. In timestamp-based versioning systems where data is automatically versioned and managed transaction time is typically the preferred dimension [13], [37], [40], [42], [45], [58], [60], [61], which is also what StarVers uses, without precluding the additional definition of valid time as part of the actual data being represented. Lastly, some of these models explicitly model continuous validity, that is, an annotation to state that a triple is valid until changed or that this triple is the most recent version. StarVers is inspired by the notions from the literature [10], [39], [40], [43], [44], [45], [50], [51], [52], [58], [60], [61] and also implements continuous validity but does not conceptually add anything new.

On a logical level, there are many theoretical but also practical works that use plain RDF to specify temporal vocabularies and annotations. A famous and probably the earliest plain RDF metadata representation approach is the reification of triples. Works employing reification to add temporal metadata include: [39], [42], [60], [61], [62]. Other models that do not require any extension of RDF are singleton [49] properties, 4-d fluents [47] and n-ary relations [47], [51]. Then, there are models that extend RDF and SPARQL with additional syntax, clauses and semantics, to allow for representing time intervals, time points [13], [40], [53], [54], [55], [57], [58], [59], [61] and other concepts such as time-based windows [15].

Our work mostly relates to those models which use established extensions of RDF, such as named graphs/quads and RDF-star that we can use to a.o. add temporal statement-level metadata to triples. Named graphs can be used as statement identifiers to refer to a quad, i.e. a triple along with the named graph it resides in [63]. Such a triple can be part of many named graphs with different provenance associated to it. This way, not just the triple but also its associated named graphs are identifiable. Temporal metadata among other metadata can then be added to these statement identifiers. The temporal metadata is typically encoded in literals, which can simply contain timestamps [63] but also more verbose spatio-temporal metadata & conditions [64] and time intervals [10]. For such complex literals custom RDF datatypes are used. Moreover, ontologies can be employed to model a richer set of temporal predicates [43], with predicates such as `time:isBeginningInclusive` and `time:isEndInclusive`. In sum, named graphs have been shown to serve well as link between triples and temporal metadata. Apart from named graphs, RDF-star is capable of representing temporal information in the following way: `«(s, p, o)» :startValidity tb ; :endValidity te`. In the literature, this representation is also referred to as "Reification Done Right" (RDR) [65][41]. In Section 5.2.1 we discuss the drawback of this naïve approach that we initially also experimented with before changing to another approach (see Section 5.2.2). Combinations of RDF-star and named graphs are also possible, such as in the case of RDF streams [15]. Timestamps are attached to named RDF-star graphs which in turn can hold nested RDF-star triples with other types of meta information. Last, version numbers instead of timestamps can also serve as a basis for "temporal information" [38], [66], which is also the approach that BEAR employs. The graph identifier is conceptually a maximal contiguous interval of versions in which a triple holds. In Section 6.1 we explain how the graph identifiers in BEAR's timestamp-based RDF datasets are actually materialised.

4.2. RDF Archiving Mechanisms

Papakonstantinou et al. [4] and Pelgrin et al. [26] studied RDF Archiving Systems & Frameworks and summarized them based on certain dimensions, such as *Storage Paradigm*, *Data Model* or *Retrieval Functionality*. Inhere we focus on materialisation queries and systems that use the timestamp-based storage paradigm as these are the vertices of our work.

Version materialisation (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 data sets as used in a specific study, for reproducibility or comparability reasons. 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 [66], [67], [26], [68].

Version (V): In the literature we can find the following two definitions of version queries whereas definition 1 entails definition 2.

Definition 1: Queries all versions of the dataset and returns a pair of version label and result set for each version where the result set is non-empty [66], [67].
Definition 2: Queries all versions of the dataset and only returns the versions' labels for each version where the result set is non-empty [26].

Delta materialisation (DM): Queries defined on the change set of two particular versions/timestamps .
Cross-version (CV): Queries defined on multiple versions, whereas these versions can be combined via set operations, joins and aggregations.
Cross-delta (CD): Like Cross-version queries, but defined on multiple changesets, e.g. to study the evolution of knowledge over time.

The **RDF versioning systems** we are about to discuss all propose a custom storage and indexing approach. As we said earlier, we are not proposing any physical storage model which is why we will not dive into the details of B+trees and other indexing techniques. However, by studying these systems we noticed that even the TB versioning approaches are very heterogeneous from their concepts down to their physical models. Especially, "timestamp-based" is not necessarily associated with actual timestamp literals but sometimes with version numbers instead.
xRDF3X [37] employs a interval-based and transaction-time-based storage model to attach a creation and a deletion timestamps to triples during SPARQL update operations. Their approach to continuous validity is to leave out the deletion timestamp for the most recent version of the triple, which is different from our approach as we will see in Section 5.2.1. Another difference to our approach is that their solution is built for RDBMS systems and not for RDF stores. In fact, the creation and deletion timestamps are stored in separate *transaction inventory*. We do not get to

see any SPARQL queries, though the authors claim that time travel queries, like in our work, are possible.
Dydra [9] extends SPARQL with a new clause, called the REVISION clause, which is similar to the GRAPH or SERVICE clause and can be used to reduce the scope of a query to a specific revision. Revisions are not only associated with an UUID but also with a timestamp. Compared to the previous work, this solution is truly built for RDF stores and the logic is mostly implemented in a SPARQL processor which is strictly separated from the RDF store. Our python-based SPARQL-star API can be considered a SPARQL processor, as well. The most important difference to our work is that we do not extend the SPARQL grammar but only use what is provvided by standard SPARQL-star (see Section 5).
v-RDFSCA [28] design and implement an RDF archiving system that achieves high compression by encoding triples and their versioning information as bitsequences. The approach to encode the versioning information is conceptually similar to [38] and [66]. Their *triple per version* (tpv) approach arranges versions as rows and triples as columns in a $v \times n$ binary matrix, where v and n are the numbers of versions and triples, respectively. If a triple is present in a version, it is represented by a `1` in the corresponding cell. The authors only theoretically discuss materialisation queries but do not provide any SPARQL patterns. Based on the function header *Mat(Q',i)* and the description we assume that they use version numbers and not timestamps.
RDF-TX [12] propose an interval-based temporal RDF model by using graph identifiers (fourth element of a quad) to assign a time interval to a triple. For example, `<California, governor, Arnold Schwarzenegger >:[11/17/2003 ... 01/02/2011]` represents a triple that has been valid between 11.17.2003 and 01.02.2011. This cited example also alludes to the validity time paradigm. This system focuses on the retrieval of facts and their real-world temporal information, such as the term of a governor's office. Thus, this system is conceptually different from our transaction-time based versioning approach. RDF-TX also models continuous validity by using the term `NOW` as an end date to express that a triple is valid until further notice, which again is different from our approach. Like Dydra, they extend the SPARQL query language by additional temporal constructs with the aim to efficiently express temporal queries.
OSTRICH [67] is an RDF archiving solution which defines itself as a hybrid IC/CB/TB approach. However, we would primarily consider it an efficient CB approach, which uses compression techniques to remove redundancies, metadata and delta chain dictionaries and separate stores for positive and negative change sets which store all data from the delta-chain. They do not elaborate on the timestamp-based aspect of their hybrid approach but from their running example and their materialisation query algorithm we conclude that there are no timestamps involved.

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 ([4] and [26]) 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 was 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 [69], [70], [71], [72], [73], which drew their ideas from version control systems. Timestamped-based solutions, on the other hand, have so far been missing out this feature ([26]).

5. StarVers Solution Design

In order to version RDF datasets within RDF-star stores StarVers solely relies on RDF-star and SPARQL-star and hence does not require versioning tools, such as CVS, SVN or Git, nor does it require separate (relational) databases to store the versions or the definition of new extensions to the language. We start with defining a running example in Section 5.1 which we use throughout this section to showcase the SPARQL-star update & query operations and RDF-star result sets. In Section 5.2, we discuss two paradigms to represent temporal metadata with RDF-star which revolve around the *query execution timestamp* and two temporal metadata attributes to annotate the start and expiration of a triple. For the initialization of datasets, i.e. assigning initial timestamps, three common write operation types and materialisation queries we show a timestamp-based versioning approach that employs SPARQL-star queries that integrate one of the previously discussed models. We dedicate a Section to each of these operations (See Sections 5.4 - 5.7). Finally, we outline and discuss our concept to automatically translate SPARQL queries and a given timestamp to such a timestamped SPARQL-star query and show how we implemented an API that achieves this in practice (Section 5.8).

5.1. Running example

To demonstrate our solution, let us assume one exemplary RDF datasets with one initial triple `tr1` and a set of triples that will be added at different points in time. The triple `tr2` is an update of `tr1`. The triples `tr3` and `tr4` are new and they share the same subject. All triples are taken from the UniprotKG snapshot releases 2021_02[12] (`tr1`) and 2021_03[13] (`tr2`, `tr3`, `tr4`).

Table 4
Initial RDF dataset (Subset from the UniprotKG 2021_02 release)

Dataset	Triple label
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .	
<http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder 59" .	`tr1`

Table 5
Updated RDF dataset (Subset from the UniprotKG 2021_03 release)

Dataset	Triple Label
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .	
<http://purl.uniprot.org/diseases/5622> skos:prefLabel "Intellectual developmental disorder, autosomal dominant 59" .	`tr2`
<http://purl.uniprot.org/diseases/6011> skos:prefLabel "Albinism, oculocutaneous, 8" .	`tr3`
<http://purl.uniprot.org/diseases/6011> skos:altLabel "Oculocutaneous albinism, type VIII" .	`tr4`

For the purpose of simulating an RDF dataset that has been converted to an RDF-star dataset and subsequently updated three times throughout a period of time, we define following chronologically ordered timestamp constants as annotated RDF literals:

```
t1: "2021-04-07T12:00:00.000+00:00"^^xsd:dateTime
t2: "2021-06-02T12:00:00.000+00:00"^^xsd:dateTime
t3: "2021-06-01T12:05:00.000+00:00"^^xsd:dateTime
t4: "2022-01-01T12:00:00.000+00:00"^^xsd:dateTime
```

We will use the labels `t1` - `t4` instead of the timestamps in the upcoming sections for the sake of readability. We derived the first three timestamps from the snapshots' release dates, which are 07.04.2021 and 02.06.2021 respectively, whereas `t2` and `t3` are only 5 minutes apart from each other. The fourth is made up for the sake of

[12]https://www.uniprot.org/news/2021/04/07/release
[13]https://www.uniprot.org/news/2021/06/02/release

demonstrating different hypothetical scenarios.

Last, for all RDF triples and *SPARQL statements* in Section 5 we assume the following prefixes and namespaces:

```
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 datatype schema and `skos` points at an IRI from the UniProt Onotology.

5.2. Two temporal RDF-star-based models

We experimented with two RDF-star-based logical representations of a conceptual temporal RDF model, which are, respectively, based on the SQT and NQT-SP patterns that we introduced earlier in Section 3.2. For either model we employ two temporal properties – `vers:valid_from` and `vers:valid_until`. The first model is a naïve and semantically correct model that, however, comes with a limitation in the case that a *data triple* has multiple time intervals in which it was valid. We will elaborate on it in Section 5.2.1. The second model that is based on the NQT-SP pattern overcomes this limitation by design and also has a more compact logical representation. We demonstrate latter model in Section 5.2.2 and due to its advantages employ it in our solution design.

5.2.1. The SQT temporal metadata representation model

Using the SQT pattern we can express a triple that has been part of the first snapshot in our running example as a timestamped triple in the following way:

```
<< tr1 >> vers:valid_from t1 .
<< tr1 >> vers:valid_until tE .
```

We assume that the triple was created at `t1`. We denote a timestamp that is assigned by the `vers:valid_from` property to a triple as *creation timestamp*. To denote that a triple is valid until further updates are provided, we use a timestamp that is far in the future, which we will refer to as *artificial expiration timestamp*. Its RDF literal is given by "9999-12-31T00:00:00.000+00:00"^^xsd:dateTime and we abbreviate it with `tE` hereinafter. We assign it using the `vers:valid_until` property. Moreover, we say *currently valid triple (pattern)* if we refer to a triple (pattern) that has an *artificial expiration timestamp*. By *valid* we only speak in the context of temporal validity and do not assume anything about its truth value.

As long as each *data triple* has only one time interval, i.e. only one pair of the previously mentioned temporal properties, this model is solid for representing temporal metadata about facts. However, if a triple has more than one time interval, representing temporal facts this way becomes flawed. As it is not so obvious why, let's assume a scenario where `tr1` should appear as deleted at timestamp `t2` and reappear as being valid again at `t3`. To delete a triple we only replace the *artificial expiration timestamp* `tE` with the actual expiration timestamp `t2`. This way, the triple is not physically deleted but outdated and hence can still be retrieved. To insert a triple, we assign a *creation timestamp* and an *artificial expiration timestamp*, like before. After both operations, we have in total four *nested quoted triples* in our dataset looking like the following:

```
nqt1: << tr1 >> vers:valid_from t1 .
nqt2: << tr1 >> vers:valid_until t2 .
nqt3: << tr1 >> vers:valid_from t3 .
nqt4: << tr1 >> vers:valid_until tE .
```

A very intuitive way of querying a snapshot valid as of `t1` from such a dataset is given in Listing 3:

Listing 3: Timestamped query using the SQT pattern

```
select ?s ?p ?o ?valid_from ?ts ?valid_until where {
    <<?s ?p ?o>> vers:valid_from ?valid_from .
    <<?s ?p ?o>> vers:valid_until ?valid_until .
    filter(?valid_from <= ?ts && ?ts < ?valid_until)
    bind(t1 as ?ts)
}
```

Table 6
Result set of the timestamped query in Listing 3

s	p	o	valid_from	ts	valid_until
<http://purl.uniprot.org/diseases/5622>	skos:prefLabel	"Intellectual developmental disorder 59"	t1	t1	t2
<http://purl.uniprot.org/diseases/5622>	skos:prefLabel	"Intellectual developmental disorder 59"	t1	t1	tE

The idea here is to first match two sets of quoted triples where the triples are annotated with the creation and deletion timestamps, respectively. We achieve this by using the two triple patterns stated in the query in Listing 3, which together form a join pattern. Then, we want to reduce the result set to only those triples that have a `vers:valid_from` timestamp smaller and a `vers:valid_until` timestamp greater than `t1`. However, with the SQT-based representation, distinguishing which quoted triples belong together, i.e. form a pair, becomes problematic. More precisely, `nqt1` is joined with both, `nqt2` and `nqt4` and both matches also fulfil the filter condition. Hence, the query above yields two records leading to a logically flawed picture of the snapshot where a triple is valid and has expired at the same time. Because we do not want to observe the behavior of triples at quantum level we will introduce a paradigm that is free of such issues in the next section.

5.2.2. *The NQT-SP temporal metadata representation model*

A triple can be annotated with our two temporal properties using the NQT-SP pattern as follows:

```
<< << tr1 >> vers:valid_from t1 >> vers:valid_until tE .
```

If we replay the same scenario as in the previous section, just with the new pattern, we get following dataset:

```
nqtsp1: << << tr1 >> vers:valid_from t1 >> vers:valid_until t2 .
nqtsp2: << << tr1 >> vers:valid_from t3 >> vers:valid_until tE .
```

The modified version of the query in Listing 3 and the result set it yields can be seen in Listing 4 and Table 7, respectively.

Listing 4: Timestamped query using the NQT-SP pattern

```
select ?s ?p ?o ?valid_from ?ts ?valid_until where {
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until ?valid_until .
    filter(?valid_from <= ?ts && ?ts < ?valid_until)
    bind(t1 as ?ts)
}
```

This query now yields the true representation of the data that has been valid at `t1`. Moreover, there is less redundancy in the graph and in the query as the *data triple* `tr1` and the *data triple pattern* `?s ?p ?o` are needed only once to assign both temporal information. From here on, we refer to the variables `?valid_from` and `?valid_until` as *temporal information variables*. This paradigm can be related to relational DBMS timestamp-based versioning approaches. There, it is common practice [74] to use two columns or attributes per table to physically store the same temporal information that we assign via our two temporal properties. To retrieve a snapshot

Table 7
Result set of the timestamped query in Listing 4

s	p	o	valid_from	ts	valid_until
<http://purl.uniprot.org/diseases/5622>	skos:prefLabel	"Intellectual developmental disorder 59"	t1	t1	t2

as of a specific point in time we simply filter for the records that are in between their creation and deletion timestamps. The single NQT-SP pattern in Query 4 can be related to a relational SQL SELECT-clause with five columns, i.e. `Select ?s ?p ?o ?valid_from ?valid_until` and the filter is putting constraints on the last two columns. Thus, we are doing the same as filtering on a relational table. This is only a simple case with one triple pattern, however, as we will see in Section 5.7 it works analogously for multiple triple patterns.

5.3. *Making an RDF Dataset Versioning-Ready*

Usually, a new RDF dataset would be versioned ab-initio, i.e. each triple would receive the timestamp according to its insertion into the RDF store. In cases where an unversioned dataset should be converted to a versioned one, we can assign a common initial timestamp to all triples of that dataset. Like shown in the previous section, we want to apply the same transformation, that has been applied to `tr1` of our running example, to all triples of the given RDF dataset. One simple way how to construct a versioned dataset is to use SPARQL's update language to assign the *query execution timestamp* and the *artificial expiration timestamp* using the properties `vers:valid_from` and `vers:valid_until`, respectively. As for the *query execution timestamp*, we use SPARQL's NOW() function to retrieve it and bind it as `?tExec`. Even though W3C[5] does not specify how RDF stores should implement this function it should not matter for the identification of specific datasets as long as the function is consistent within a given RDF store. The conversion to RDF-star 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 quoted triples* together with their *query execution timestamps* via the INSERT block. The update statement in Listing 5 achieves what we just described. We assume that this update was fired at timestamp `t1`, i.e. that the NOW() function returns `t1`.

Listing 5: Construct a timestamped RDF-star dataset via SPARQL-star following the NQT-SP pattern at `t1`

```
DELETE
{ ?s ?p ?o . }
INSERT
{ <<<<?s ?p ?o>> vers:valid_from ?tExec>> vers:valid_until tE .}
WHERE
{
   ?s ?p ?o .
   BIND(xsd:dateTime(NOW()) AS ?tExec).
}
```

5.4. *Insert*

Once the RDF-star dataset has been constructed and every triple has thereby been annotated with temporal information, we can perform timestamped update operations such as inserts to add *nested quoted triples* with more recent timestamps. In Listing 6 we demonstrate a timestamped insert with `tr3` and `tr4` at timestamp `t2` from our running example. In the INSERT block we use the exact same pattern as for the initialization of the dataset in the INSERT block of Listing 5. In the WHERE-clause we employ the VALUES expression which allows us to add a set of triples. We use the variables `?s ?p ?o` to bind them to all respective RDF resources within the VALUES block. To insert all triples with the same *creation timestamp*, we bind a variable, namely, `?tExec`, to the *query execution timestamp* returned by the function NOW(). We assume that latter function returns the timestamp literal `t2`.

Practically, there are, however, limitations as of how many triples can be added. These very much depend on the RDF store vendors and their capabilities within the SPARQL engines. We have conducted preliminary experiments, however, the evaluation of SPARQL update statements are not within the scope of this paper.

Listing 6: Inserting `tr3` and `tr4` from the running example via SPARQL-star following the NQT-SP pattern at timestamp `t2`.

```
INSERT
{ << <<?s ?p ?o>> vers:valid_from ?tExec >> vers:valid_until tE . }
WHERE {
    VALUES (?s ?p ?o) {
        (tr2)
        # Add further triples
    }
    BIND(xsd:dateTime(NOW()) AS ?tExec)
}
```

5.5. Update

Updating a set of triples with a new set of triples of the same dimensions can be achieved with the SPARQL-star update statement in Listing 7. The graph update operations comprise one NQT-SP pattern in the DELETE block and two such patterns in the INSERT block. In the former block we physically delete the *currently valid triples* provided in the VALUES block. In the INSERT block we use the same pattern as in the DELETE block, just with the *query execution timestamp* instead of the *artificial expiration timestamp*. This way we achieve what we call an *outdate* (see next Section). The last pattern is the same pattern as in the INSERT block of the insert example from the previous section.

Let us turn our attention to the update logic in the WHERE clause and first explain it conceptually by formulating the triple sets as three $n \times 3$-matrices.

`mNewPre` ... A set of triples, where for each resource, that should not be updated, a placeholder UNDEF is used.
`mOld` ... A set of *currently valid triples* prior to the update.
`mNew` ... A set of *currently valid triples* after the update.

While `mNewPre` and `mOld` are input matrices, `mNew` is derived within the WHERE clause. All UNDEF values in the preliminary matrix `mNewPRE` should be replaced by entry-wise resources in `mOld` to form the final set of new triples `mNew`. The UNDEF placeholder in `mNewPre` allows us to exclude those resources that we do not want to update and thereby avoid copy-pasting resources. In SPARQL, we achieve this with a combination of an IF statement and the BIND & BOUND expressions. A binding variable is interpreted as unbound if the binding value is UNDEF. The values of the binding variables `?newSPre`, `?newPPre` and `?newOPre` correspond to the entries in `mNewPre`. If one of these variables is bound, meaning that an RDF resources was provided, we bind one of the respective new variables, namely `?newS`, `?newP` and `?newO`, to the provided resource. Otherwise, we bind latter variables to the old resource. Thereby we construct a triple as a combination of existing and new resources. The patterns in the DELETE and INSERT blocks refer to the variables that correspond to `mNew`. Let us also note that triples provided in `mOld` where no *currently valid triple* can be matched in the underlying RDF-star set will simply be ignored. One additional filter in our statement makes sure that triples, that are identical in terms of their string interpretations in both matrices should not be updated. The idea here simply is to avoid unnecessary write operations as for each triple we conceptually perform three write operations. In Listing 7 we update the triple `tr1` with triple `tr2` at timestamp `t3`, whereas latter timestamp is returned by the NOW() function.

Listing 7: Updating `tr1` with `tr2` from the running example via SPARQL-star following the NQT-SP pattern at timestamp `t3`.

```
DELETE {
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until tE .
}
INSERT {
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until ?tExec .
    << <<?newS ?newP ?newO >> vers:valid_from ?tExec >> vers:valid_until tE .
}
WHERE {
    VALUES (?s ?p ?o ?newSPre ?newPPre ?newOPre) {
      (<http://purl.uniprot.org/diseases/5622>
      skos:prefLabel
      "Intellectual developmental disorder 59"
      UNDEF
      UNDEF
      "Intellectual developmental disorder, autosomal dominant 59")
      # Add further 6-tuples
    }
    # versioning
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until tE .
    BIND(xsd:dateTime(NOW()) AS ?tExec).

    # For UNDEF values in the new triple use the same value in the old triple
    BIND(IF(BOUND(?newSPre), ?newSPre, ?s) AS ?newS)
    BIND(IF(BOUND(?newPPre), ?newPPre, ?p) AS ?newP)
    BIND(IF(BOUND(?newOPre), ?newOPre, ?o) AS ?newO)

    # nothing should be changed if old and new value are the same
    FILTER(CONCAT(STR(?s), STR(?p), STR(?o))
            != CONCAT(STR(?newS), STR(?newP), STR(?newO)))
}
```

5.6. *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 expiration timestamp* must exist on that triple. Outdating a timestamped triple simply means replacing its *artificial expiration timestamp* with an actual one, which in our solution is always the *query execution timestamp*. In terms of SPARQL-star, we achieve this the same way as we did in the update template in Listing 7 to outdate `tr1`. The only difference between the delete and update operation in the graph update operation blocks is that for the delete operation we do not insert a new, updated, triple. In the WHERE-clause we first need to match the triples that we want to outdate including their *artificial expiration timestamp*. In case, there is no *artificial expiration timestamp* the UPDATE statement will simply not update any triples. Like for the insert operation, we use a VALUES block and an additional NQT-SP pattern to match the provided data triples from the VALUES block against the underlying RDF dataset. In Listing 8 we show how we apply this logic to outdate `tr2` from our running example at timestamp `t4`.

Listing 8: Outdating `tr2` from the running example via SPARQL-star following the NQT-SP pattern at timestamp `t4`.

```
DELETE {
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until tE .
}
INSERT {
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until ?tExec.
}
WHERE {
    VALUES (?s ?p ?o) {
      (tr2)
      # Add further triples which should be deleted
    }
    # versioning
    << <<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until tE .
    BIND(xsd:dateTime(NOW()) AS ?tExec).
}
```

5.7. Version Materialisation

In Listing 4 we already presented a very simple SPARQL-star materialisation query. In here, we extend to more complex SPARQL-star queries that include more than one triple pattern, multiple BGPs which are either joined or combined by means of UNION, and subqueries. We also discuss property paths and show the limitations of our approach for property paths of arbitrary length. Finally, we assemble a set of rules that can be used to transform SPARQL queries into timestamped SPARQL-star queries and demonstrate their application on an example query. We then use the resulting SPARQL-star query to show how to retrieve the datasets as they were at timestamps `t1` (initialize RDF-star dataset), `t2` (insert), `t3` (update), and `t4` (outdate), by simply changing the timestamp literal in the query. Hereinafter, we refer to the `?ts` variable used in Listing 4 simply as *timestamp variable*. This variable can bind to specific timestamps, like `t1`, but also to the NOW() function, thus, to the *query execution timestamp*.

5.7.1. Multiple triple patterns

Consider the JOIN pattern in Listing 9 consisting of two *data triple patterns*:

Listing 9: Multiple triple patterns in SPARQL

```
?s ?p ?o .
?o ?p2 ?o2 .
```

Similarly to the query in Listing 4 we nest the *data triple patterns* into NQT-SP patterns with the addition that each *data triple pattern* gets a unique pair of *temporal information variables*. We achieve the uniqueness by adding a numerical suffix and count it up for every variable pair, as shown in Listing 10. We can apply this logic to any number of triple patterns inside the query. The reason for distinct *temporal information variables* is that each triple pattern has a different set of solution mappings which can potentially differ in their temporal information. Having only one pair of *temporal information variables* for all triple patterns would form a JOIN pattern with the *temporal information variables* as the join conditions, which is undesired for the mentioned reason. To narrow down the solution mappings to only yield triples that are valid as of a specific timestamp we add a filter condition for each NQT-SP pattern. This filter condition contains a timestamp variable which must be the same in every such filter condition. As this variable is our input parameter for a timestamp literal we add a BIND expression so that whoever is providing the input needs to do so only at that place in this query. After these extensions our final SPARQL-star pattern looks like the following:

Listing 10: Starvers approach for pattern in Listing 9

```
<< <<?s ?p ?o>> vers:valid_from ?valid_from_1 >> vers:valid_until ?valid_until_1 .
FILTER(?valid_from_1 <= ?ts  && ?ts < ?valid_until_1)
<< <<?o ?p2 ?o2>> vers:valid_from ?valid_from_2 >> vers:valid_until ?valid_until_2 .
FILTER(?valid_from_2 <= ?ts && ?ts < ?valid_until_2)
BIND(t1 as ?ts)
```

Note that for this simple query body it is straight forward to place a BIND expression so that the timestamp variable is in-scope for all filters. However, matters change if have multiple BGPs and subqueries, as we will see in the next section.

5.7.2. Scoping of the timestamp variable

If we have more complex patterns like queries with multiple BGPs, UNIONs or sub queries we also want to consider the scope of the *timestamp variables* and the *temporal information variables* in which they are visible. W3C gives a good overview of how the scope of variables differs between patterns [14]. For example, if we have the timestamped SPARQL-star representation of two BGPs that are joined, like in Listing 11, the timestamp variable `?ts` will be visible, no matter whether we place it in the inner or outer BGP.

Listing 11: Starvers approach for a JOIN of two BGPs

```
<< <<?s ?p ?o>> vers:valid_from ?valid_from_1 >> vers:valid_until ?valid_until_1.
FILTER(?valid_from_1 <= ?ts && ?ts < ?valid_until_1)
{
  << <<?o ?p2 ?o2>> vers:valid_from ?valid_from_2 >> vers:valid_until ?valid_until_2.
  FILTER(?valid_from_2 <= ?ts && ?ts < ?valid_until_2)
  BIND(t1 as ?ts)
}
# Also possible to place it here
# BIND(t1 as ?ts)
```

However, if we consider a UNION of two BGPs like in Listing 12 the binding needs to be placed either outside of the combined BGPs or within each of them as combined BGPs via UNION do not share their variables with each other. This also means that the *temporal information variables* could be reused and do not have to be unique across the whole query in this case.

Listing 12: Starvers approach for a UNION of two BGPs

```
SELECT ?s ?o {
  {
    << <<?s skos:prefLabel ?o>> vers:valid_from ?valid_from >>
      vers:valid_until ?valid_until .
    FILTER(?valid_from <= ?ts && ?ts < ?valid_until)
  }
  UNION
  {
    << <<?s skos:altLabel ?o>> vers:valid_from ?valid_from >>
      vers:valid_until ?valid_until .
    filter(?valid_from <= ?ts && ?ts < ?valid_until)
  }
  BIND(t1 as ?ts)
}
```

Similarly, a subquery only propagates variables up the hierarchy that it projects, i.e. that are used in its SELECT clause. Also, a subquery does not have access to variables from outside its scope. This means that we need to add the timestamp binding at two places, once within the inner and once within outer query, as demonstrated in Listing 13. Alternatively, we could only place the timestamp binding within the subquery and project it to the outer query by adding it to the SELECT clause of the subquery. Like in the UNION example, we can re-use the *temporal information variables* inside the subquery.

[14] https://www.w3.org/TR/sparql11-query/#variableScope

Listing 13: Starvers approach for a subquery

```
SELECT * {
    <<<<?s ?p ?o>> vers:valid_from ?valid_from >> vers:valid_until ?valid_until .
    FILTER(?valid_from <= ?ts && ?ts < ?valid_until)
    BIND(t1 as ?ts) # not visible in sub query
    {
        SELECT ?o ?p2 ?o2 {
          <<<<?o ?p2 ?o2>> vers:valid_from ?valid_from >> vers:valid_until ?valid_until .
          FILTER(?valid_from <= ?ts && ?ts < ?valid_until)
          BIND(t1 as ?ts) # not visible outside of query
        }
    }

}
```

5.7.3. *Property paths*

As we explained previously, we need to assign a pair of *temporal information variables* and apply a filter to every NQT-SP triple statement. A property path with a length greater than 1, however, "hides in-between triple statements" so that we have no way to do this and thus need to resolve the path first. Luckily, there are equivalent patterns for alternative paths, inverted paths and sequence paths which only rely on simple triple patterns and UNION [15]. Take a look at following example, which includes a sequence and an alternative path:

Listing 14: A sequence path combined with a alternative path in SPARQL

```
 ?term (skos:broader/(skos:prefLabel | skos:altLabel)) ?label .
```

To resolve the sequence we can use an auxiliary variable `aux_1` and based on how long the path is we have to use an according number of auxiliary variables which again have to be unique in the scope they are visible in. Here we can again just add a sequence number as a suffix to each such variable. Furthermore, we can resolve the alternative path via UNION like we show in Listing 15. The binding can again be placed in the outer BGP so that the timestamp variable is visible in both inner graph patterns.

Listing 15: Starvers approach for pattern in Listing 14 based on path resolution

```
?term skos:broader ?aux_1 .
{ << <<?aux_1 skos:prefLabel ?label >> :valid_from ?valid_from >>
                                                       :valid_until ?valid_until .
  FILTER(?valid_from <= ?ts && ?ts < ?valid_until)
}
UNION
{ << <<?aux_1 skos:altLabel ?label >> :valid_from ?valid_from >>
                                                       :valid_until ?valid_until .
  FILTER(?valid_from <= ?ts && ?ts < ?valid_until)
}
BIND(t1 as ?ts)
```

[15]https://www.w3.org/TR/sparql11-query/#propertypath-syntaxforms

Our approach, however, reaches its limit when it comes to property paths of arbitrary length, such as `?x foaf:knows+/foaf:name ?name`. Here, we cannot resolve the path prior to query execution as we do not know how often the path will expand, i.e. we do not know how many triple statements the query actually contains. We could potentially solve this with subqueries where we first retrieve all triples for a specific timestamp, i.e. a snapshot, and then use plain SPARQL on this subgraph. Here we would also need to align the variables from the subquery with the variables from the original query. Another approach would be to first materialize the snapshot as a named graph and then query from that named graph. To experiment with paths of arbitrary length is, however, not within the scope of this paper.

5.7.4. *Protocol to transform SPARQL queries into timestamped SPARQL-star queries*

To transform any SPARQL materialisation query (except the ones with paths of arbitrary length) into a timestamped SPARQL-star materialisation query we present five rules that opt for simplicity rather than optimization in terms of variable re-use and variable scoping:

`rule1`: Property paths of fixed length must be resolved to an alternative form.
`rule2`: Each *data triple pattern* must be annotated with a unique pair of *temporal information variables* within the scope of a SPARQL SELECT-query.
`rule3`: In every BGP a `?ts` variable must be in-scope. If there are more `?ts` variables in the query, they all must bind to the same timestamp literal.
`rule4`: Each NQT-SP pattern must have a unique FILTER condition which reduces the solution mappings of that pattern to only those triples where `?ts` is in between its *temporal information variables*.
`rule5`: The timestamp variable `ts` and *temporal information variables* must only be purposed for what is described in `rule2` and `rule3`, respectively, and should otherwise not be used.

While `rule1` and `rule2` are a direct consequence from the discussions in the previous sections, `rule3` opts for a simple solution of the variable scoping problem. We have seen that for different query patterns there are at least two possibilities on where to place the timestamp variable binding. To avoid case distinctions, e.g. between queries with subqueries and those without, we simply place a timestamp variable binding in every BGP and this way ensure that it is accessible for every temporal filter condition. Users of our approach should also be aware that the additional variables we use in timestamped SPARQL-star queries are reserved and must not be used in the original SPARQL queries (see `rule5`) for obvious reasons.

Let us now use this protocol to construct a demonstrative SPARQL-star query to materialize a subset at timestamps `t1`-`t4`. In Listing 16 we show a query with two NQT-SP triple patterns that are joined by their *data triple*. Each triple pattern has a unique pair of *temporal information variables* (`rule2`). The `?ts` is bound to a timestamp `tX` (`rule3`), which is a placeholder for the aforementioned timestamps. Each NQT-SP pattern is also accompanied by a unique filter condition (`rule4`). There were no paths in the original query (`rule1`) and no conflicting variable names (`rule5`). In Table 8 we see the materialisations for each query version reflecting the dataset at each of the four timestamps. In practice, we do not want to manually build such SPARQL-star queries but rather automatically construct them by transforming a SPARQL materialisation query and a given timestamp into a timestamp-based SPARQL-star query. Our protocol answers the question *what* must be transformed. In the next section, we will also explain *how* we do the transformation.

Listing 16: SPARQL-star query that retrieves diseases and their labels at a given timestamp {vtX}

```
Select ?disease ?label ?alt_label {
  bind({tX} as ?ts)
  << <<?disease skos:prefLabel ?label>> vers:valid_from ?valid_from_1>>
    vers:valid_until ?valid_until_1 .
  filter(?valid_from_1 <= ?ts && ?ts < ?valid_until_1)
  OPTIONAL {
    << <<?disease skos:altLabel ?alt_label>> vers:valid_from ?valid_from_2>>
      vers:valid_until ?valid_until_2 .
    filter(?valid_from_2 <= ?ts && ?ts < ?valid_until_2)
    }
}
```

Table 8
Result sets from Query 16 based on different timestamps from our running example

disease	label	alt_label	Query
http://purl.uniprot.org/diseases/5622	"Intellectual developmental disorder 59"		Query where tX = `t1`
http://purl.uniprot.org/diseases/5622	"Intellectual developmental disorder 59"		Query where tX = `t2`
http://purl.uniprot.org/diseases/6011	"Albinism, oculocutaneous, 8"	"Oculocutaneous albinism, type VIII"	
http://purl.uniprot.org/diseases/5622	"Intellectual developmental disorder, autosomal dominant 59"		Query where tX = `t3`
http://purl.uniprot.org/diseases/6011	"Albinism, oculocutaneous, 8"	"Oculocutaneous albinism, type VIII"	
http://purl.uniprot.org/diseases/6011	"Albinism, oculocutaneous, 8"	"Oculocutaneous albinism, type VIII"	Query where tX = `t4` or tX = any timestamp more recent than `t4`

5.8. *Automatic translations from SPARQL to SPARQL-star*

In the previous sections we laid out the RDF-star representation for timestamped-based versioning accompanied by SPARQL-star query and update templates for our target systems, namely, RDF-star stores. The source systems and target users of our solution, however, should not be concerned with versioning and should still operate with "plain" RDF and SPARQL in order to query live data, extract snapshots and modify RDF datasets. To this end, we designed and implemented a translator with python's rdflib in a pipeline-like fashion that takes a SPARQL query and a timestamp as an input to generate a timestamped SPARQL-star query. We furthermore implemented the functions *version all triples*, *insert* and *update*, *outdate* – one for each of the respective operations in Sections 5.3 - 5.6.

Our pipeline for translating SPARQL queries, as depicted in Figure 1, is divided into five high-level steps. In the first step we transform the SPARQL query into a SPARQL algebra tree (see Section 3.3). Our rationale for this is that it is easier to operate on well-defined and normalized expressions. rdflib's algebra implementation is based on W3C's SPARQL grammar[16]. In the second step we traverse this tree to seek out path expressions, such as `SequencePath`, and replace them with equivalent patterns. In the third step, we traverse the tree another time and track down every BGP in the query as this is where we would find SPARQL triple patterns. We replace all *data triple patterns* with placeholders for the timestamping extensions. This is because rdflib does not support SPARQL-star in its query algebra implementation as of version 6.2. So we need to inject our SPARQL-star extensions later in the query text. In the fourth step we build the preliminary query text from the query tree. For this we implemented an "algebra-to-query" translator, namely `translateAlgebra` in rdflib[17], which rdflib's main developers reviewed and merged into the main branch. This translator covers all SPARQL 1.1. expressions, however, it is still under development as some bugs have been reported e.g. for the SERVICE keyword. Our broad range of test cases shows that query trees can be properly translated into the query text. In the last step we replace every BGP with a new graph pattern including the NQT-SP expressions with the embedded *data triple patterns* & attached unique *temporal*

[16]https://www.w3.org/TR/sparql11-query/#sparqlGrammar
[17]https://rdflib.readthedocs.io/en/stable/_modules/rdflib/plugins/sparql/algebra.html

information variables, the corresponding filters and a binding for the *timestamp variables*. The *timestamp variables* are enumerated up to the number of BGPs in the query, hence, there are no two BGPs binding the same *timestamp variable*. Our algorithm is implemented in the `timestamp_query` function of our python API and available on our Github page[18].

Our implementation of SPARQL update statements employs the templates we showed in Listings 6 - 8. All three SPARQL update functions take a set of triples as first parameter whereas the `update` function also takes a second set that represents that new triples that will replace the old ones after the execution. As in practice there is a limited number of triples that can be updated during one execution we added a `chunk_size` parameter to set the number of triples that the Starvers write functions (insert, update, outdate) should send via HTTP POST at maximum and thereby determine the number of iterations needed to update the whole batch. For example, if we want to insert 15.000 triples the operation might fail at two points. Either, the request is too big for HTTP POST or the targeted triple store cannot handle the load. We can, however, use the aforementioned parameter and set the chunk size to 5000, which will lead to three iterations to insert all provided triples. Some of the provided triples might contain blank nodes. As blank nodes can be seen as variables, with labels that triple store engines will not preserve, it is somewhat counter-intuitive to add temporal metadata to them. Besides, RDF triple stores like GraphDB do not allow to have blank nodes in a DELETE statement. However, we can use skolemization[19] to turn blank nodes into actual resources prior to their insertion. As of now, our SPARQL update functions do not treat blank nodes in any special way but technically it is possible to at least insert them.

6. Evaluation

6.1. Evaluation setup

Naming conventions To maintain a clear and consistent naming convention for our policies and datasets, we follow a specific format. The first two letters of a policy denote the versioning approach used, while the following two letters are always "sr" within the scope of this experiment to indicate a single repository. Another possibility would be "mr" which stands for multiple repositories. The final two letters indicate the data representation type, with "ng" standing for named graphs and "rs" representing RDF-star. From here on, we use the term *dataset variant* to refer to a dataset that serves the evaluation of a specific policy and that is serialized using a certain representation. If we refer to a specific variant we write *<BEAR<X>> <policy>* dataset, e.g. the BEARC `ic_sr_ng` dataset is a *dataset variant* that serves the evaluation of the `ic_sr_ng`-policy and is part of BEARC. We also sometimes use synonyms, such as BEARC's IC-based dataset variant.

Datasets We conducted our experiment with three out of four BEAR datasets[20], namely two DBPedia live datasets (BEARB day and hour) and one from the European Open Data Portal (BEARC). They all vary in the number of snapshots (update frequency), number of triples, change ratios and other parameters, which we display in Table 9. If we take BEARB_day as a basis, then we can notice the comparably high change ratio & high number of

[18]https://github.com/GreenfishK/starvers/blob/main/src/starvers/starvers.py
[19]https://www.w3.org/TR/rdf11-concepts/#section-skolemization
[20]https://aic.ai.wu.ac.at/qadlod/bear.html

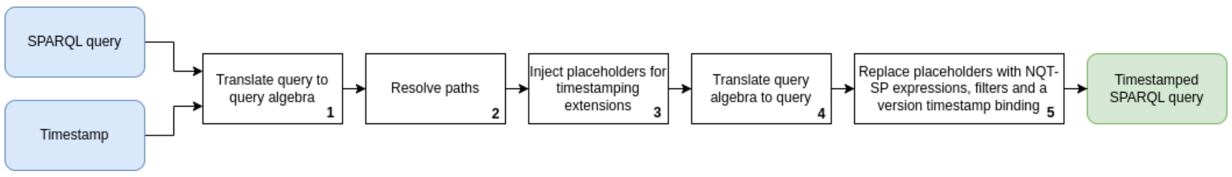

SPARQL query
Timestamp
Translate query to query algebra 1
Resolve paths 2
Inject placeholders for timestamping extensions 3
Translate query algebra to query 4
Replace placeholders with NQT-SP expressions, filters and a version timestamp binding 5
Timestamped SPARQL query

Fig. 1. SPARQL to SPARQL-star translation process

version-oblivious triples in BEARC and the high number of versions in BEARB_hour. *Version-oblivious* triples are all distinct *data triples* across all versions of a dataset. Each of these datasets has different representations, files and serializations to cater to the common versioning approaches IC, CB and TB. E.g. data belonging to the IC approach are a bundle of individual RDF files, each representing one snapshot and serialized as n-triples, while the TB dataset is one n-quads file. We initially also experimented with BEARA, the fourth and largest dataset, however, due to data quality issues in this dataset we exclude it from our evaluation. In particular, the independent copies contain 6.62% of invalid triples on average, invalidated by Jena's and GraphDB's RDF parsers. In the BEARA `tb_sr_ng` dataset 35,5% of all triples are invalid. Furthermore, the blank nodes in most ICs are skolomized, i.e. turned into valid IRIs, while they are not in the "named graphs"-based *dataset variant*, making these two sets misaligned.

Query Sets Each of the BEAR datasets also comes with a **query set** where for the BEARB dataset these are subdivided into 62 simple lookup and 20 JOIN queries. BEARC comprises 10 complex queries with more than two triple patterns and the following expressions: OPTIONAL, FILTER, UNION, ORDER BY, LIMIT and OFFSET.

Policies To store different snapshots (ICs) the authors use independent repositories/instances of the evaluated storage technologies. Whether it is called a repository or instance depends on the RDF store vendor. We use these terms interchangeably hereinafter. For the CB approach they use the same approach to store the individual change sets. Additionally they experiment with hybrid policies where they use named graphs to index the change sets and load them into a single repository. BEAR's TB *dataset variants* employ a versioning approach that is substantially different from most timestamped-based approaches that we have mentioned in Section 4.1 which is why we explain it in the following: First, we want to mention that even though the authors call it timestamp-based approach, there are no timestamps in these *dataset variants*. Each triple is versioned like the following:

> # *version_string* ... string of contiguous version numbers in which this triple holds
> # *version_number* ... specific version number
> @prefix ex: <http://bear_example.org/>
> @prefix : <http://example.org/>
>
> `ex:s ex:p ex:o :v_`*{version_string}*
> {`:v_`*{version_string}* `owl:versionInfo "`*{version_number}*`" :versions .` | for each *{version_number}* in *{version_string}* }

For example, a triple `ex:s1 ex:p1 ex:o1` inserted in version 21, deleted in version 24, re-inserted in version 25 and deleted again in version 26 would be annotated like this:

```
ex:s1 ex:p1 ex:o1 :v_21_22_23_25 .
:v_21_22_23_25 owl:versionInfo "21" :versions .
:v_21_22_23_25 owl:versionInfo "22" :versions .
:v_21_22_23_25 owl:versionInfo "23" :versions .
:v_21_22_23_25 owl:versionInfo "25" :versions .
```

Baselines Like in BEAR [66] we evaluate our approach with different policies as baselines. Unlike BEAR we establishe our baseline policies by fixing the storage mechanism to only one repository per policy, which keeps all data necessary to evaluate that policy. Moreover, we use named graphs in all baseline policies to index different snapshots (IC), change sets (CB) and annotate triples (TB). This way we eliminate the variability in the number of index structures across polices and make our approach better comparable to named graphs as a frequently used metadata annotation approach. As there is no publicly available CB-based RDF archiving system that fulfils our requirements in our evaluation process we implement a standard CB approach by mixing SPARQL and procedural paradigms. The discussed RDF Archiving Systems in Section 4 are either not designed for RDF stores [37], do not conform with the standard SPARQL(-star) language [9], [12] or cannot be connected to an arbitrary RDF store as they implement custom storages [28], [67]. We also cannot re-use BEAR's CB-based algorithm for this approach as it can only handle simple lookup materialisation queries, which is the only type of materialisation queries that the BEAR authors evaluate. In contrast, our baseline implementation also works with JOIN and complex SPARQL queries. Given a query, it first constructs the specific dataset version that is encoded in the query and then executes the

Table 9
Dataset configurations [66]

Dataset label	versions	Number of triples in first snapshot	Number of triples in last snapshot	Change ratio (Arithmetic mean)	Static core	Version-oblivious triples	Query Sets
BEARB_hour	1.299	33.502	43.907	0.304%	32.303	178.618	62 lookup queries and 20 join queries
BEARB_day	89	33.502	43.907	1.778%	32.448	83.134	62 lookup queries and 20 join queries
BEARC	33	485.179	563.738	67.61%	178.484	9.403.540	10 complex queries

Table 10
RDF archiving/versioning policies that we evaluated. For each policy we constructed one file that holds all data for that policy and loaded it into one repository.

Policy	Description	Serialization format	Mapping to BEAR policy
ic_sr_ng	One named graph for each IC.	TriG	no such policy
cb_sr_ng	Two named graphs for each change set (add and delete).	TriG	TB/CB
tb_sr_ng	Each triple is assigned to a named graph. The named graph IRI encodes all versions, in which this triple is valid.	NQuads	TB
tb_sr_rs	Each *StarVers data triple* is nested into the RDF-star NQT-SP pattern holding a creation and expiration timestamp	N-Triples-star	no such policy

query against that dataset. The specific dataset version is always constructed by starting from the initial dataset and consecutively applying all changesets/patches up to that version. We are aware of other variations/improvements, e.g. to add an "in-between" snapshot every k change sets in order to not always start from the initial dataset version [66]. However, as this is just a tweak to trade-off storage consumption for query performance at every k-th snapshot we do not think that including these hybrid CB/IC policies would change the conclusion that we draw from our performance evaluation. We show our policies, their features and a mapping to the BEAR policies in Table 10.

RDF stores As storage technology we used two RDF stores, namely Jena TDB2 store and GraphDB 10.1.2, which are both capable of storing multi-level nested triples. Jena's TDB2 store is part of the stain/jena-fuseki 4.0.0 Docker image.

6.2. Evaluation process

Our automated and reproducible evaluation process is inspired by OSTRICH [67] and their containerized solution [21]. We divide it in 7 steps, as depicted in Figure 2, and execute each step using a separate docker-compose service (blue boxes).

In the **first step**, we download the IC and TB files of all BEAR datasets from the BEAR homepage[20] and extract them. **Next**, we clean the datasets by removing triples that are invalid according to the rdf4j and Jena's parser and skolemizing blank nodes. This step was initially included due to bad data quality in the BEARA dataset, which we eventually could not process due to its size and therefore excluded it. As for the other datasets, we found only one invalid line per snapshot in BEARC. Also, these remaining three datasets do not contain any blank nodes. In the **third step**, we construct one *dataset variant* for each of the `ic_sr_ng`, `cb_sr_ng` and `tb_sr_rs` policies. We first compute changesets from the ICs and then use the ICs and/or CBs as inputs to construct the *dataset variants* for the respective policies. Latter datasets match the descriptions and serialization formats in Table 10. To construct

[21]https://github.com/rdfostrich/BEAR

the `tb_sr_rs` dataset variants we use our StarVers python API to first insert all triples from the initial snapshot and then subsequently insert and outdate triples from the positive and negative change sets, respectively. We thereby employ GraphDB as storage engine. We use a timestamp increment of 1 second for each subset constituting one snapshot, with the only purpose to make them distinguishable and queryable by their timestamps. We evaluate the performance of each update operation for a range of chunk sizes. We perform this specific evaluation only with the BEARC dataset as it is the only one that has large enough change sets which practically cannot be updated with a single SPARQL update statement but needs to be split into multiple sets of statements (chunks). In the **fourth step** we ingest the constructed *dataset variants* and the original original BEAR `tb_sr_ng` datasets into either RDF store and measure the ingestion time, raw file size and database file size. This produces 24 indexed RDF(-star) datasets, i.e. 12 repositories for GraphDB and 12 instances for Jena. **Next**, we construct the final materialisation queries from the raw queries for each *dataset variant* and dataset version. E.g. BEARB_hour has 1299 versions, 62 lookup and 20 JOIN queries. Thus, after this step we have 1299 * (62 + 20) * 4 = 426.072 BEARB_hour queries. In total, we generate 12 query sets. In the **sixth step** we execute queries from each query set against their two matching repositories from step 4. At the beginning of each iteration we re-start the database server and re-configure the location to the repository so that only one repository is active at a time. During this step we measure the time for every single query execution. In the **last step**, we aggregate the measurements and generate plots for the data ingestion, storage consumption and query performance, which we show in the next section.

We run our evaluation on a virtual machine which uses four cores of an AMD EPYC 7542 processor. The cores have a 2.9GHz clock frequency. We assigned 110GiB of RAM to the virtual machine. Our evaluation process is available on Github[22].

6.3. Results

We document the query performance, data ingestion time and storage consumption in Figures 3 - 5 and the update performance in Figure 6. To calculate the mean ingestion time we take the arithmetic mean of 10 ingestion runs per RDF store and *dataset variant*. We measure the storage consumption of the raw (unindexed) dataset files and RDF store repositories after we load the files. To measure the execution time of a single query, regardless of the RDF store, we capture the timestamp before we sent the query to the SPARQL endpoint via rdflib's SPARQLWrapper and the timestamp after it returns the result set. We aggregate the query runtimes on query set level by computing an arithmetic mean. We capture the time for the performance evaluation of the SPARQL-star update statements in the same manner as we did for the queries. While we are able to successfully evaluate the update statements using GraphDB, we faile to do so with Jena TDB2. In the next three Sub-sections we elaborate on the results and some problems we encounter with Jena TDB2.

6.3.1. Data ingestion and storage consumption

In Figure 3 we see the results for the mean ingestion time and storage consumption for **BEARB_day** in the lower plot. While the results for the change-based and both timestamp-based policies are comparable, the IC-based policy has a much higher storage need and therefore also a higher ingestion time. This is because of the large *static core* and comparably small set of *version-oblivious* triples in BEARB_day. The BEARB_day `ic_sr_ng` dataset carries the *static core* in each of the 89 independent copies while the other *dataset variants* do not have a redundant *static core*. Interestingly, GraphDB stores the BEARB_day `ic_sr_ng` dataset more efficiently than Jena TDB2 as the storage need for the indexed data is almost half of what the raw file needs for former while it is twice as much for latter. In terms of storage consumption we cannot identify one clearly preferably policy for the BEARB_day dataset.

If we look at **BEARB_hour** we see how the `tb_sr_ng` policy joins the previous bad performer with its huge raw file size and thereby influenced DB file size and ingestion time. If we recall the approach in BEAR's `tb_sr_ng` datasets to version one triple (see Section 6.1), we can easily see how the large number of versions blows up the number of metadata quads and also enlarges the *version strings* that encode all versions in which a triple was valid. For this dataset the `cb_sr_ng` and `tb_sr_rs` policy are on par in terms of storage efficiency.

[22]https://github.com/GreenfishK/starvers_eval

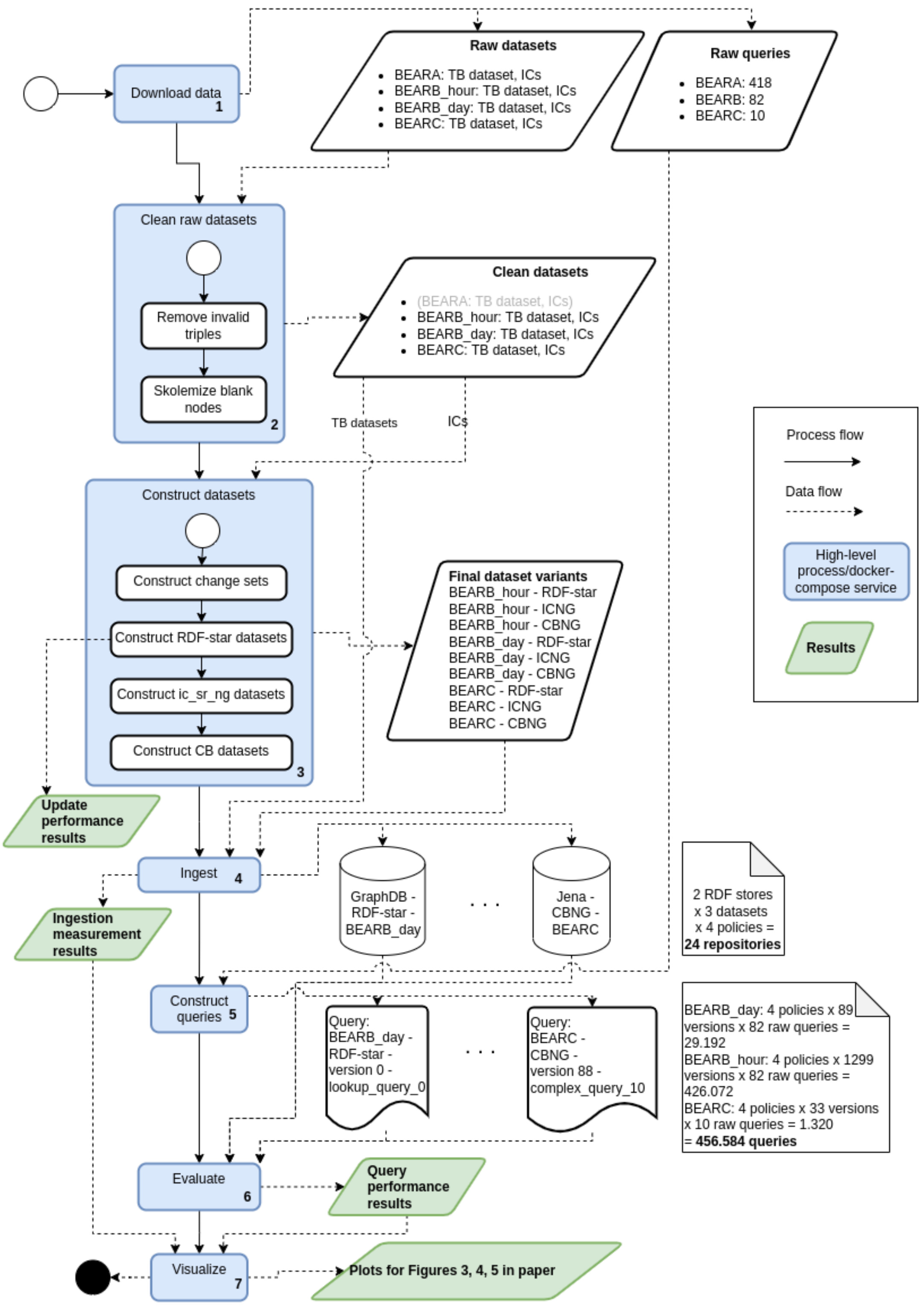
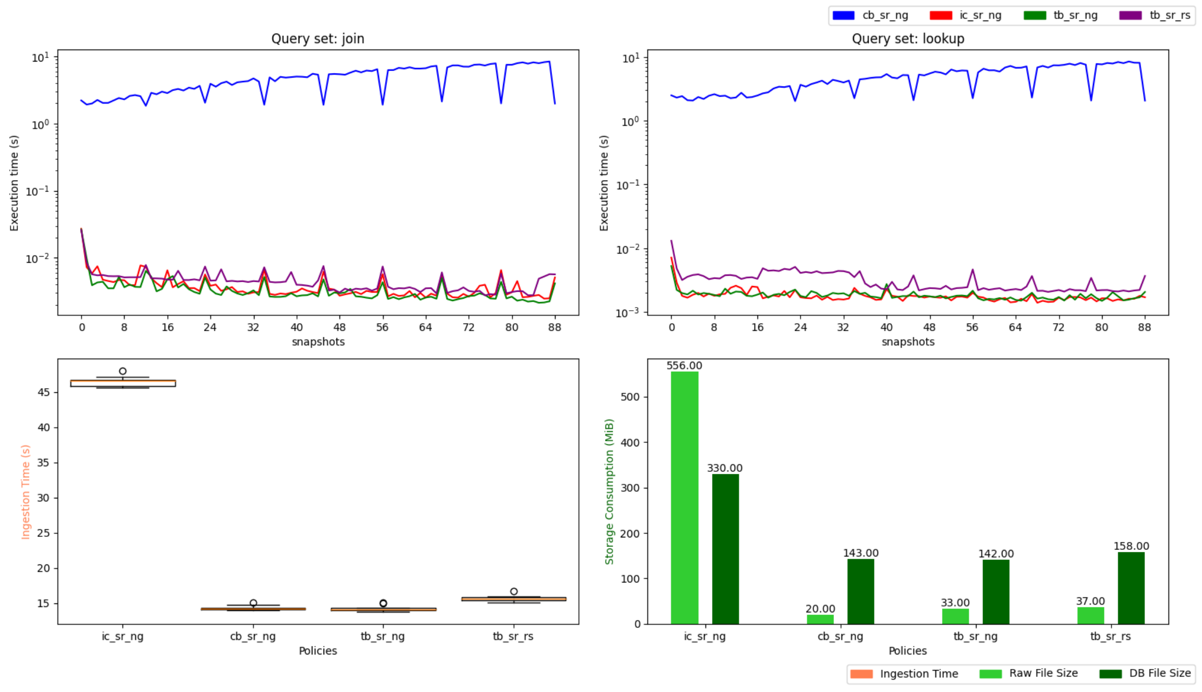
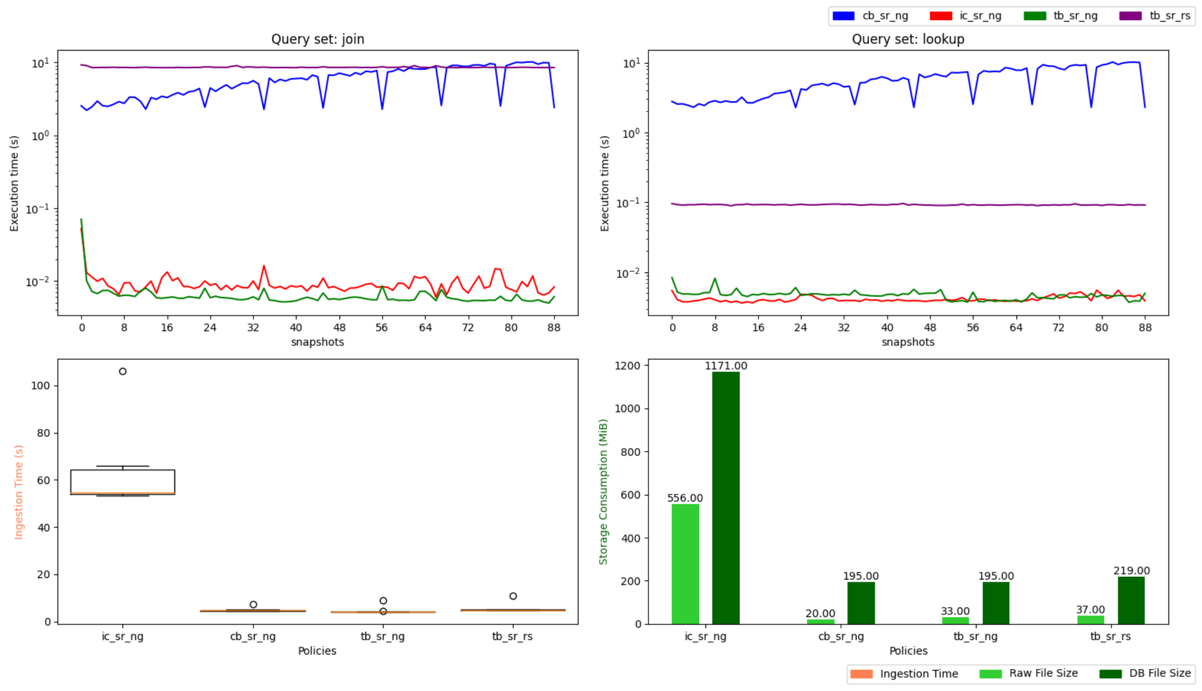

Download data 1
Raw datasets
BEARA: TB dataset, ICs
BEARB_hour: TB dataset, ICs
BEARB_day: TB dataset, ICs
BEARC: TB dataset, ICs
Raw queries
BEARA: 418
BEARB: 82
BEARC: 10
Clean raw datasets
Remove invalid triples
Skolemize blank nodes
2
Clean datasets
(BEARA: TB dataset, ICs)
BEARB_hour: TB dataset, ICs
BEARB_day: TB dataset, ICs
BEARC: TB dataset, ICs
TB datasets
ICs
Construct datasets
Construct change sets
Construct RDF-star datasets
Construct ic_sr_ng datasets
Construct CB datasets
3
Final dataset variants
BEARB_hour - RDF-star
BEARB_hour - ICNG
BEARB_hour - CBNG
BEARB_day - RDF-star
BEARB_day - ICNG
BEARB_day - CBNG
BEARC - RDF-star
BEARC - ICNG
BEARC - CBNG
Process flow
Data flow
High-level process/docker-compose service
Results
Update performance results
Ingest 4
Ingestion measurement results
GraphDB - RDF-star - BEARB_day
. . .
Jena - CBNG - BEARC
2 RDF stores x 3 datasets x 4 policies = 24 repositories
Construct queries 5
Query: BEARB_day - RDF-star - version 0 - lookup_query_0
. . .
Query: BEARC - CBNG - version 88 - complex_query_10
BEARB_day: 4 policies x 89 versions x 82 raw queries = 29.192
BEARB_hour: 4 policies x 1299 versions x 82 raw queries = 426.072
BEARC: 4 policies x 33 versions x 10 raw queries = 1.320
= 456.584 queries
Evaluate 6
Query performance results
Visualize 7
Plots for Figures 3, 4, 5 in paper

Fig. 2. Evaluation steps of automated evaluation process

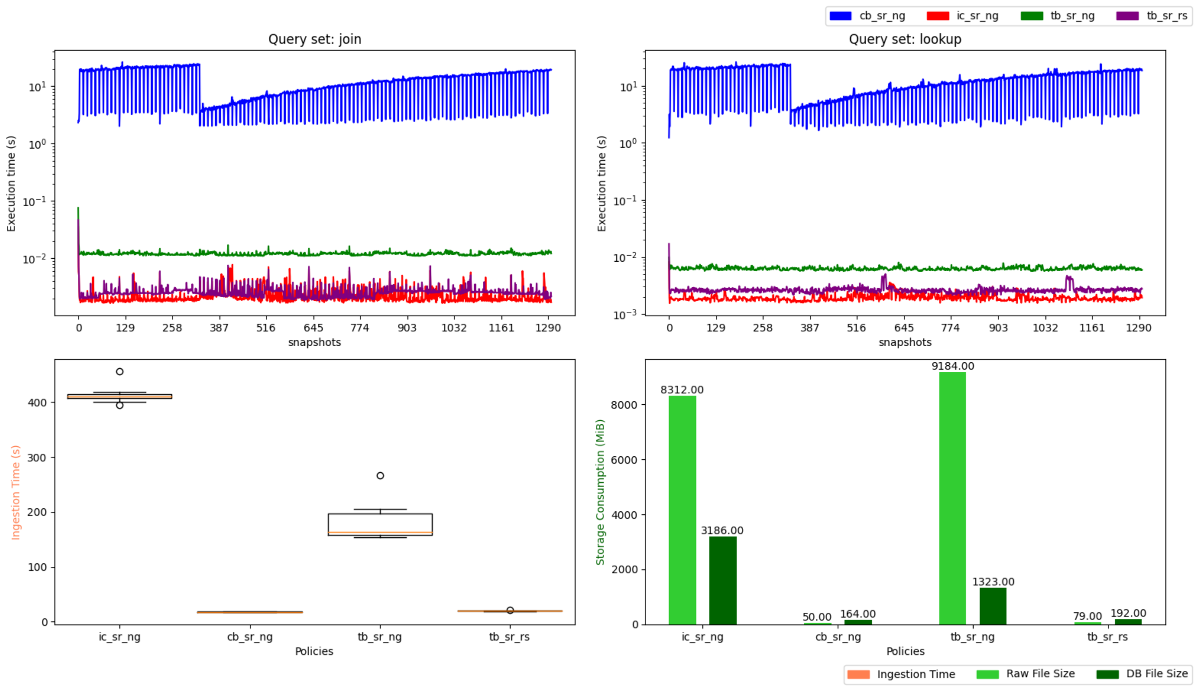


(a) Results for GraphDB

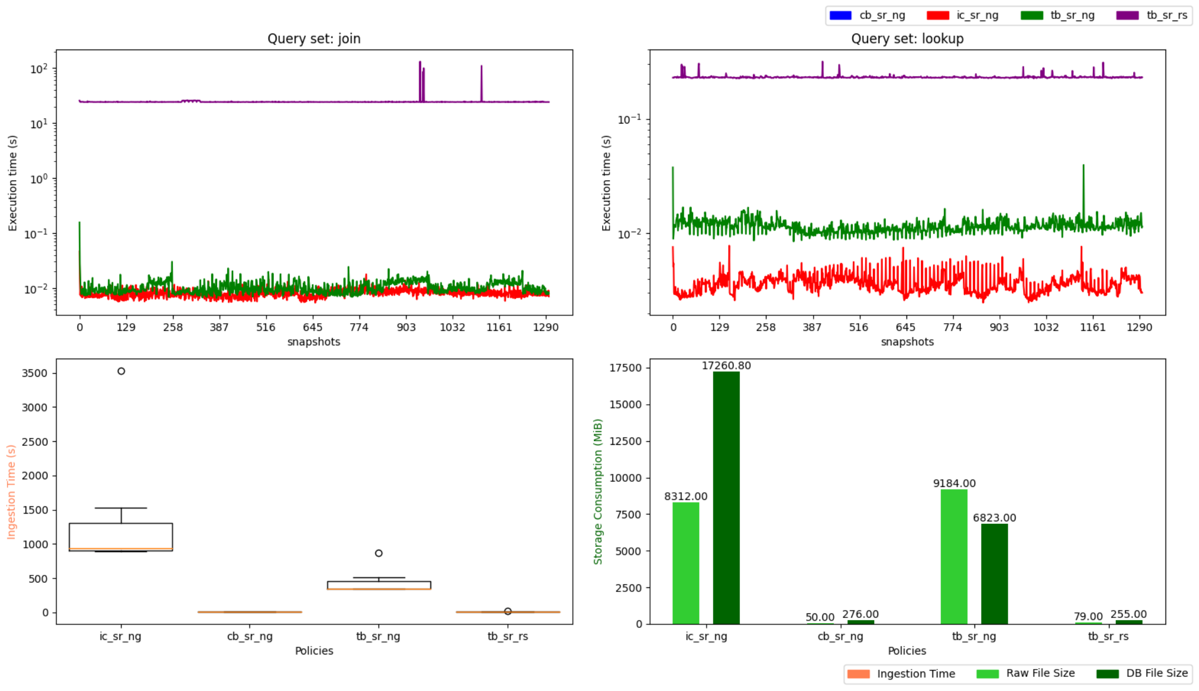


(b) Results for Jena

Fig. 3. Query performance, data ingestion and storage consumption for BEARB_day

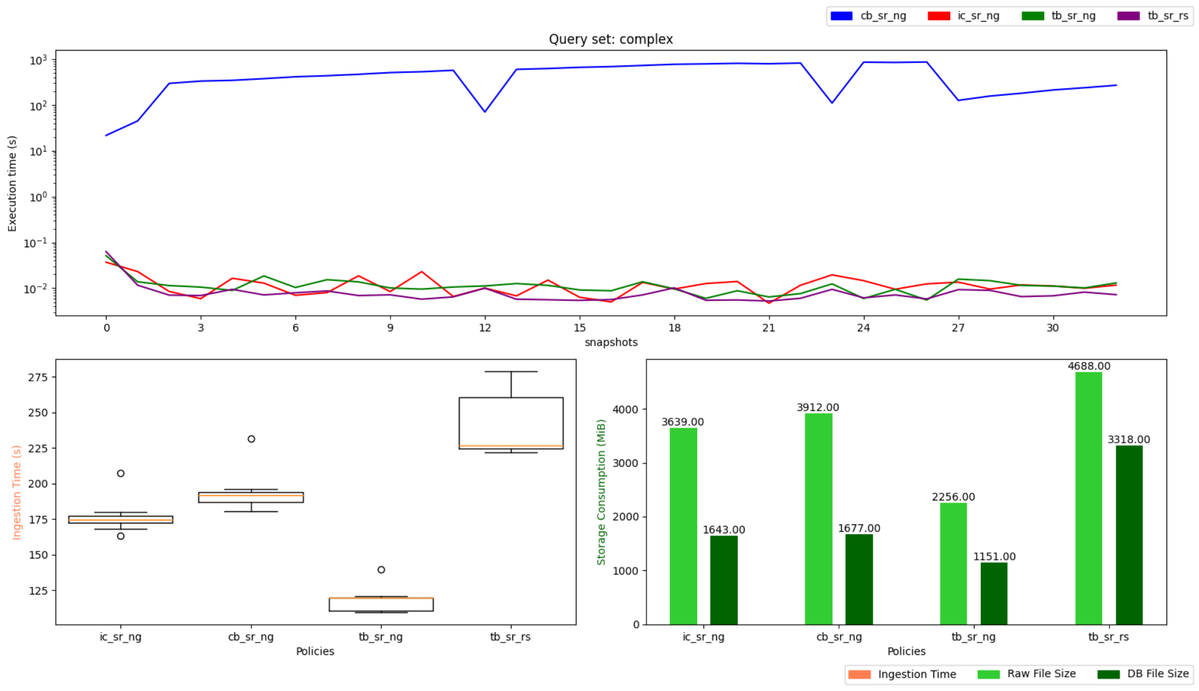


(a) Results for GraphDB

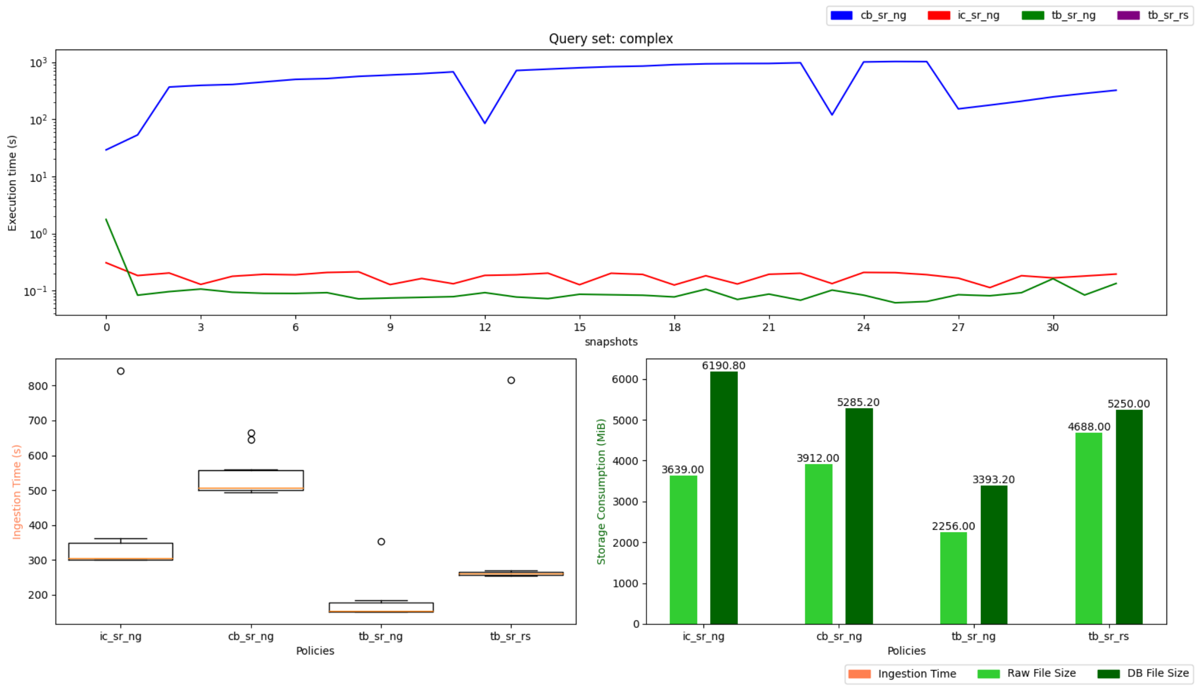


(b) Results for Jena

Fig. 4. Query performance, data ingestion and storage consumption for BEARB_hour



(a) Results for GraphDB



(b) Results for Jena

Fig. 5. Query performance, data ingestion and storage consumption for BEARC

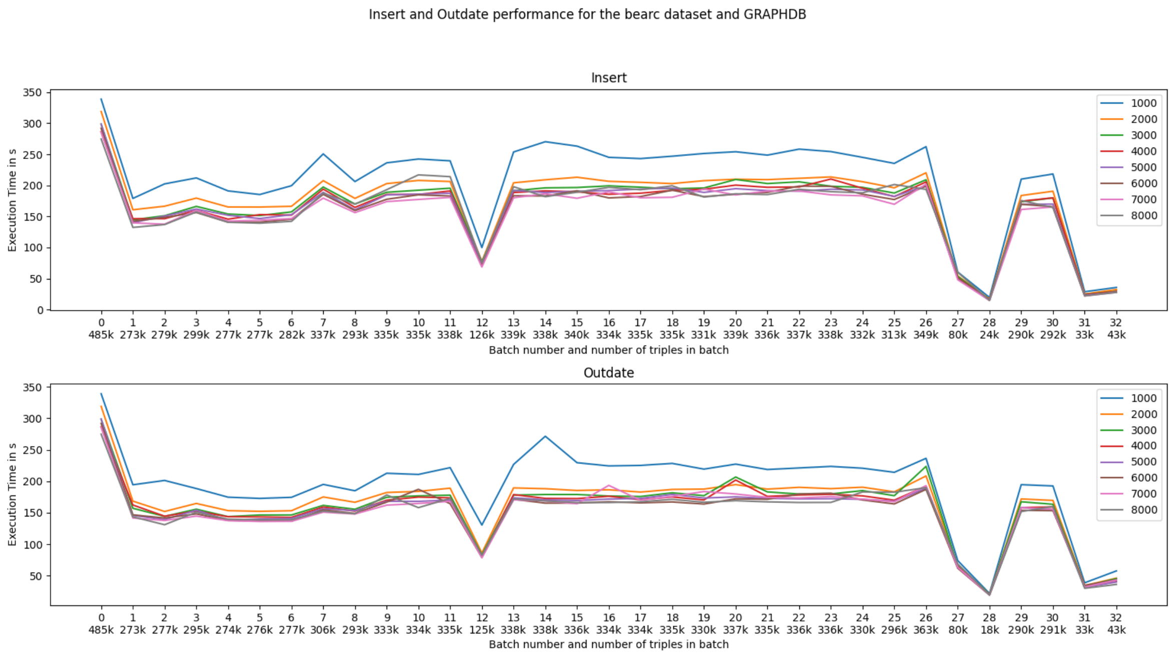


Fig. 6. Update performance for BEARC

Turning our attention to the **BEARC** dataset, we see how the `tb_sr_ng` policy outperforms all others. This is because latter policy never re-states a *data triple* but only updates the associated named graph URI by adding a version number to the version string. Every data triple is unique which is not always the case in the other policies. This method saves spaces in scenarios where triples are deleted and then re-inserted at a later point.

6.3.2. *Query Performance*

The query performance results for the smallest dataset, **BEARB_day**, and its lookup and JOIN queries reveal that the query execution runtime of our RDF-star based policy can vastly differ between RDF store vendors. With GraphDB the measurements of `tb_sr_rs`, `ic_sr_ng` and `tb_sr_ng` all lie within an order of magnitude for both query sets. However, with Jena TDB2 it performs even worse than the change-based policy when JOIN queries are issued. A Jena community member recently confirmed [23] that Jena is not optimized for nested quoted triples. The change-based policy performs worse than all other policies in general. This is due to high snapshot construction time in the first step of the algorithm. The SPARQL queries themselves retrieve the results with a speed comparable to IC-based approaches, once the snapshot has been constructed or materialized in any way.

The plots for the **BEARB_hour** dataset show a clearer picture and less intertwined curves. For GraphDB, we can see that the IC-based policy performs the best, followed by our RDF-star-based policy. While still within an order of magnitude the `tb_sr_ng` policy clearly distances itself from the aforementioned two policies toward longer running queries. For Jena, we see a similar picture as with BEARB_day. The curve for the CB-approach is not in the Jena-specific plot because the Jena engine closed the connection after 350-400 snapshot builds without enough details in the stack trace why it did so. We can only exclude a memory overflow as we monitored the memory footprints of all processes during the execution and none of them did even come close to the RAM limit.

The query performance results for **BEARC** and its complex query set show that for GraphDB both timestamp-based policies outperform the IC-based policy while the performance of latter two is indistinguishable. Again, the curve for our RDF-star-based policy is not shown on the Jena-specific plot. This because the Jena engine could

[23]https://github.com/apache/jena/issues/1744

not finish the execution of a single complex NQT-SP query within one hour of time. So we decided to exclude the evaluation of our policy for this setting.

6.3.3. Update Performance

In Figure 6 we show the runtime performance for the insert and delete operations of the starvers API that we execute during the build stage of the `tb_sr_rs` variant of the BEARC dataset. The plot exhibits curves for a range of chunk sizes for the insert (top) and outdate (bottom) SPARQL-star update operations against GraphDB. The results suggest that updates are generally faster with bigger chunk sizes, but more importantly, that there is a practical limit of how many triples can be updated with a single SPARQL-star statement. This practical limit can vary as it depends on the browser, i.e. processing unit for HTTP POST requests, but also on the RDF store and its SPARQL engine. Therefore, the reported limit of 8.000 triples per SPARQL-star update statement should not be considered as globally true. On GraphDB's website, they report a batch insert of 237M triples [24]. We therefore conclude that web-based transfer is causing this limitation rather than the RDF store engine. However, also RDF stores can have some practical limitations, e.g in the encoding of datatyped datetime literals, which leads to a stalled execution of our *outdate* operation against Jena TDB2. In particular, `xsd:dateTime`-typed datetime literals with years greater than 7999, such as `"9999-12-31T00:00:00.000+02:00"^^xsd:dateTime`, are not value-encoded but stored in their original lexical form (see Andy Seaborn's resolving post to our question [25]). This makes our *outdate* SPARQL pattern malfunction as the currently valid triples, annotated with former timestamp are not being deleted. Despite this observation we still consider the choice of the datetime literal for the *artificial expiration timestamp* as conceptually valid. Nevertheless, we will need to update it in our practical solution in order to cover a wider range of RDF stores.

7. Conclusion and Discussion

In this paper we designed an RDF-star and SPARQL-star framework for timestamp-based versioning of RDF datasets. The framework includes an RDF-star-based model to represent timestamped RDF triples, SPARQL-star templates for common update statements (insert, update, delete) in a timestamped manner, a SPARQL-star template for timestamped version materialisation queries, a ruleset and a SPARQL-algebra based approach to translate SPARQL queries to SPARQL-star queries and a simple way to transform RDF datasets into timestamped RDF-star datasets. To evaluate our solution we employed three datasets, BEARB_day, BEARB_hour and BEARC, from the BEAR framework. We thereby re-used the named-graph based variants as baselines for the timestamp-based policy. To cover the IC-based versioning approach we added a new policy to the baseline which stores each snapshot in a named sub-graph instead of individual repositories. Lastly, our CB-based policy is based on BEAR's hybrid TB/CB policy, however, we use another retrieval mechanism as opposed to BEAR to also allow for materializing data with JOIN and more complex query patterns. To cater to our RDF-star based policy we constructed information equivalent RDF-star variant for each of the three BEAR datasets.

In general, there is no approach that surpasses all others in every scenario. If we consider storage and query performance as equally important and we are free to choose the storage technology, then, for dynamic datasets like BEARB_hour our RDF-star based approach would be the preferable one. It has a slightly worse query performance than the IC-based approach, but outperforms latter approach in terms of storage consumption by far. Moreover, it exceeds BEAR's TB-based approach in both - query performance and storage consumption. Latter is over six times smaller for the indexed data. For datasets with high change-ratios like BEARC our RDF-star based approach still delivers a compatible query performance that lies within the same order of magnitude as BEAR's TB-based approach and our IC-based baseline approach. However, due to its redundant data triples, which are added whenever a delete or re-insert occurs throughout the history of updates, the raw dataset is twice as big as BEAR's storage-efficient and redundancy-free timestamp-based representation. Efficient indexing techniques, like those of GraphDB can compress the dataset and thereby reduce the size. Nevertheless, even the indexed data is up to three times greater

[24]https://graphdb.ontotext.com/documentation/10.2/benchmark.html
[25]https://github.com/apache/jena/issues/1810

in size than the baseline policies. We believe, that indexing techniques for RDF-star can still be improved as we already see a big difference between Jena TDB2 and GraphDB, and thereby achieve an even higher compression for RDF-star datasets.

Even though BEAR's timestamp-based versioning approach allows for efficient queries in the scenario of high change-ratios due to its compact representation, we argue that updates are far from being scalable. In Section 6.1 we have shown how BEAR annotates a triple with a version string at the quad position that contains all versions in which the triple resides. Updating the dataset requires a check performed on every single triple to either append a new version number in case nothing changed for that triple or leaving the triple's version string unmodified in case of a deletion. That is why we suggest to implement StarVers, which allows us to efficiently update and query dynamic RDF datasets in a timestamp-based fashion and thereby enable reproducible research, traceability and re-evaluation based on earlier states of a data set for any arbitrary subgraph identified via SPARQL-star queries at any arbitrary point in time.

With the introduction of the chunk size parameter we have shown that any practical limits for bulk updates can be easily overcome by dividing batches into chunks. We have also seen that SPARQL DELETE statements come with additional challenges. Especially, blank nodes, which are not allowed in SPARQL DELETE blocks, and different value-encodings of datetime literals between RDF stores is what requires us to think of adaptation to the specifics of the engine used.

Note, that not all of the current RDF-star systems can handle arbitrary nesting levels. Also, there is no agreed upon internal representation of multiple nesting levels between RDF store vendors [75]. Our experiments show what impact this heterogeneity can have on the query performance. That is why the choice of an RDF store should be evaluated before using it in a production system or migrating to a different one.

8. Future Work

As we expect the Linked Data community to increasingly adopt RDF-star for a wide range of applications we want to re-evaluate our work with other RDF stores which are yet to become RDF-star stores. This includes further experimentation with the BEAR framework, especially with the largest dataset BEARA, but also looking into new frameworks which encompass other datasets and query sets. Furthermore, we want to evaluate datasets and queries from real world applications, such as curated ontologies like the Uniprot Ontology from our running example. Moreover, as we are aiming toward the timestamping of highly dynamic and streaming data, we want to evaluate our SPARQL update statements with real-world RDF streams, such as DBPedia live. This way we hope to engage in 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.

One of RDF's main characteristics is that RDF stores can use rulesets to infer and materialize new triples upon ingestion or insertion. However, there seems to be a lack of support for inference with RDF-star. The RDF-star stores we evaluated do not apply inference mechanisms on nested triples. Even if they did, we would still need to timestamp these newly materialized triples, i.e. turn them into nested quoted triples. While this does not seem like an unsolvable task, it still needs more research on how to efficiently achieve this.

So far, we only focused on materialisation queries. As we have mentioned in Section 4.2 there are other types of queries that RDF archiving systems usually support. In a preliminary experiment we have already successfully shown that queries such as delta materialisation queries, version queries and cross-version join queries can be supported by our RDF-star based framework. We aim to integrate these types of queries into our framework in the near term and to conduct a scientific evaluation and report on them.

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.

[2] M.Y. Jaradeh, A. Oelen, K.E. Farfar, M. Prinz, J. D'Souza, G. Kismihók, M. Stocker and S. Auer, Open research knowledge graph: next generation infrastructure for semantic scholarly knowledge, in: *Proceedings of the 10th International Conference on Knowledge Capture*, 2019, pp. 243–246.

[3] D. Dessí, F. Osborne, D.R. Recupero, D. Buscaldi and E. Motta, SCICERO: A deep learning and NLP approach for generating scientific knowledge graphs in the computer science domain, *Knowledge-Based Systems* **258** (2022), 109945.

[4] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis and G. Roussakis, Versioning for Linked Data: Archiving Systems and Benchmarks., *Proceedings of the Workshop on Benchmarking Linked Data (BLINK'16) co-located with the 15th International Semantic Web Conference (ISWC)* **1700** (2016).

[5] 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.

[6] A. Rauber, B. Gößwein, C.M. Zwölf, C. Schubert, F. Wörister, J. Duncan, K. Flicker, K. Zettsu, K. Meixner, L.D. McIntosh, S. Pröll, T. Miksa and M.A. Parsons, Precisely and Persistently Identifying and Citing Arbitrary Subsets of Dynamic Data, *Harvard Data Science Review* **3**(4) (2021). doi:10.1162/99608f92.be565013.

[7] J.J. Carroll, C. Bizer, P. Hayes and P. Stickler, Named graphs, *Journal of Web Semantics* **3**(4) (2005), 247–267.

[8] 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.

[9] J. Anderson and A. Bendiken, Transaction-Time Queries in Dydra., in: *Joint Proceedings of the 2nd Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW'16) and the 3rd Workshop on Linked Data Quality (LDQ'16) co-located with 13th European Semantic Web Conference (ESWC'16)*, Vol. 1585, 2016, pp. 11–19.

[10] K. Bereta, P. Smeros and M. Koubarakis, Representation and querying of valid time of triples in linked geospatial data, in: *Proceedings of the 10th International Conference on The Semantic Web: Semantics and Big Data (ISWC'13)*, Springer, 2013, pp. 259–274.

[11] F. Grandi, T-SPARQL: A TSQL2-like Temporal Query Language for RDF, in: *Symposium on Advances in Databases and Information Systems (ADBIS'10)*, 2010, pp. 21–30.

[12] 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.

[13] A. Rodrıguez, R. McGrath, Y. Liu, J. Myers and I. Urbana-Champaign, Semantic management of streaming data, in: *Proceedings of the 2nd International Workshop on Semantic Sensor Networks*, Vol. 80, 2009, pp. 80–95.

[14] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle and M. Grossniklaus, C-SPARQL: SPARQL for continuous querying, in: *Proceedings of the 18th International Conference on World Wide Web*, 2009, pp. 1061–1062.

[15] R. Keskisärkkä, E. Blomqvist, L. Lind and O. Hartig, RSP-QL: Enabling Statement-Level Annotations in RDF Streams, in: *Proceedings of the 15th International Conference on Semantic Systems. The Power of AI and Knowledge Graphs (SEMANTiCS'19)*, Springer, 2019, pp. 140–155.

[16] D. Alocci, J. Mariethoz, O. Horlacher, J.T. Bolleman, M.P. Campbell and F. Lisacek, Property graph vs RDF triple store: A comparison on glycan substructure search, *PloS one* **10**(12) (2015), e0144578.

[17] M.V. Sokolova, F.J. Gómez and L.N. Borisoglebskaya, Migration from an SQL to a hybrid SQL/NoSQL data model, *Journal of Management Analytics* **7**(1) (2020), 1–11. doi:10.1080/23270012.2019.1700401.

[18] G.E. Modoni, M. Sacco and W. Terkaj, A survey of RDF store solutions, in: *Proceedings of the 2014 International Conference on Engineering, Technology and Innovation (ICE)*, IEEE, 2014, pp. 1–7.

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

[20] K. Alaoui, A Categorization of RDF Triplestores, in: *Proceedings of the 4th International Conference on Smart City Applications*, SCA '19, Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362894. doi:10.1145/3368756.3369047.

[21] 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.

[22] D. Graux, F. Orlandi, T. Kaushik, D. Kavanagh, H. Jiang, B. Bredican, M. Grouse and D. Geary, Timelining Knowledge Graphs in the Browser, in: *Proceedings of the 6th International Workshop on the Visualization and Interaction for Ontologies and Linked Data (VOILA'21)*, 2021.

[23] J. Eriksson and A. Hakim, Format Conversions and Query Rewriting for RDF* and SPARQL, 2018, student thesis.

[24] 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: *Proceedings of the International Semantic Web Conference (ISWC'21)*, 2021, pp. 1–5.

[25] R. Cyganiak, A relational algebra for SPARQL, *Digital Media Systems Laboratory HP Laboratories Bristol* **35**(9) (2005).

[26] O. Pelgrin, L. Galárraga and K. Hose, Towards fully-fledged archiving for RDF datasets, *Semantic Web* (2020), 1–24.

[27] M. Frommhold, R.N. Piris, N. Arndt, S. Tramp, N. Petersen and M. Martin, Towards versioning of arbitrary RDF data, in: *Proceedings of the 12th International Conference on Semantic Systems*, 2016, pp. 33–40.

[28] A. Cerdeira-Pena, A. Farina, J.D. Fernández and M.A. Martínez-Prieto, Self-indexing RDF archives, in: *Proceedings of the Data Compression Conference (DCC)*, IEEE, 2016, pp. 526–535.

[29] 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.
[30] F. Zhang, Z. Li, D. Peng and J. Cheng, RDF for temporal data management–a survey, *Earth Science Informatics* **14**(2) (2021), 563–599.
[31] H.-T. Wang and A.U. Tansel, Temporal extensions to RDF, *Journal of Web Engineering* (2019).
[32] J. Frey, K. Müller, S. Hellmann, E. Rahm and M.-E. Vidal, Evaluation of metadata representations in RDF stores, *Semantic Web* **10**(2) (2019), 205–229.
[33] A. Analyti and I. Pachoulakis, A survey on models and query languages for temporally annotated RDF, *International Journal of Advanced Computer Science & Applications* **1**(3) (2008), 28–35.
[34] A. Pugliese, O. Udrea and V. Subrahmanian, Scaling RDF with time, in: *Proceedings of the 17th International Conference on World Wide Web*, 2008, pp. 605–614.
[35] L. Yan, P. Zhao and Z. Ma, Indexing temporal RDF graph, *Computing* **101** (2019), 1457–1488.
[36] J. Huang, W. Chen, A. Liu, W. Wang, H. Yin and L. Zhao, Cluster query: a new query pattern on temporal knowledge graph, *World Wide Web* **23** (2020), 755–779.
[37] T. Neumann and G. Weikum, x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases, in: *Proceedings of the 36th International Conference on Very Large Data Bases*, Vol. 3, VLDB Endowment, 2010, pp. 256–263.
[38] I. Cuevas and A. Hogan, Versioned Queries over RDF Archives: All You Need is SPARQL?, in: *Proceedings of the 6th Workshop on Managing the Evolution and Preservation of the Data Web (MEPDaW)*, 2020, pp. 43–52.
[39] C. Gutierrez, C. Hurtado and A. Vaisman, Temporal rdf, in: *Proceedings of the Second European Semantic Web Conference on The Semantic Web: Research and Applications (ESWC'05)*, Springer, 2005, pp. 93–107.
[40] F. Grandi, Multi-temporal RDF Ontology Versioning, in: *Proceedings of the 3rd International Workshop on Ontology Dynamics at the 8th International Semantic Web Conference*, 2009.
[41] 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.
[42] F. Zhang, K. Wang, Z. Li and J. Cheng, Temporal data representation and querying based on RDF, *Ieee access* **7** (2019), 85000–85023.
[43] E.G. Kalayci, S. Brandt, D. Calvanese, V. Ryzhikov, G. Xiao and M. Zakharyaschev, Ontology–based access to temporal data with Ontop: A framework proposal, *International Journal of Applied Mathematics and Computer Science* **29**(1) (2019), 17–30.
[44] C. Zaniolo, S. Gao, M. Atzori, M. Chen and J. Gu, User-friendly temporal queries on historical knowledge bases, *Information and Computation* **259** (2018), 444–459.
[45] D. Yang and L. Yan, Transforming XML to RDF (S) with temporal information, *Journal of computing and information technology* **26**(2) (2018), 115–129.
[46] M. Chekol, G. Pirrò, J. Schoenfisch and H. Stuckenschmidt, Marrying uncertainty and time in knowledge graphs, in: *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, Vol. 31, 2017.
[47] S. Batsakis, E.G. Petrakis, I. Tachmazidis and G. Antoniou, Temporal representation and reasoning in OWL 2, *Semantic Web* **8**(6) (2017), 981–1000.
[48] M. RobatJazi, M.Z. Reformat, W. Pedrycz and P. Musilek, Lori: Linguistically oriented RDF interface for querying fuzzy temporal data, in: *Proceedings of the 11th International Conference on Flexible Query Answering Systems*, Springer, 2015, pp. 337–352.
[49] 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*, 2014, pp. 759–770.
[50] M. Dylla, I. Miliaraki and M. Theobald, A temporal-probabilistic database model for information extraction, *Proceedings of the 39th International Conference on Very Large Data Bases* **6**(14) (2013), 1810–1821.
[51] B. Motik, Representing and querying validity time in RDF and OWL: A logic-based approach, *Journal of Web Semantics* **12** (2012), 3–21.
[52] S. Bykau, J. Mylopoulos, F. Rizzolo and Y. Velegrakis, On modeling and querying concept evolution, *Journal on Data Semantics* **1** (2012), 31–55.
[53] A. Zimmermann, N. Lopes, A. Polleres and U. Straccia, A general framework for representing, reasoning and querying with annotated semantic web data, *Journal of Web Semantics* **11** (2012), 72–95.
[54] Y. Wang, M. Zhu, L. Qu, M. Spaniol and G. Weikum, Timely yago: harvesting, querying, and visualizing temporal knowledge from wikipedia, in: *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 697–700.
[55] O. Udrea, D.R. Recupero and V. Subrahmanian, Annotated RDF, *ACM Transactions on Computational Logic (TOCL)* **11**(2) (2010), 1–41.
[56] C. Tao, W.-Q. Wei, H.R. Solbrig, G. Savova and C.G. Chute, CNTRO: a semantic web ontology for temporal relation inferencing in clinical narratives, in: *AMIA Annual Symposium Proceedings*, Vol. 2010, American Medical Informatics Association, 2010, p. 787.
[57] N. Lopes, A. Polleres, U. Straccia and A. Zimmermann, AnQL: SPARQLing up annotated RDFS, in: *Proceedings of the 9th International Semantic Web Conference*, Springer, 2010, pp. 518–533.
[58] J. Tappolet and A. Bernstein, Applied temporal RDF: Efficient temporal querying of RDF data with SPARQL, in: *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, Springer, 2009, pp. 308–322.
[59] B. McBride and M. Butler, Representing and querying historical information in RDF with application to E-discovery, *HP Laboratories Technical Report, HPL-2009-261* (2009).
[60] C. Gutierrez, C.A. Hurtado and A. Vaisman, Introducing time into RDF, *IEEE Transactions on Knowledge and Data Engineering* **19**(2) (2006), 207–218.
[61] M. Gergatsoulis and P. Lilis, Multidimensional RDF, in: *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, Springer-Verlag, 2005.
[62] J. Huber, Temporal reasoning for RDF (S): A Markov logic based approach (2014), Working Paper.

[63] B. Schueler, S. Sizov, S. Staab and D.T. Tran, Querying for meta knowledge, in: *Proceedings of the 17th International Conference on World Wide Web*, 2008, pp. 625–634.

[64] M. Koubarakis and K. Kyzirakos, Modeling and querying metadata in the semantic sensor web: The model stRDF and the query language stSPARQL, in: *Proceedings of the 7th Extended Semantic Web Conference on The Semantic Web: Research and Applications*, Springer, 2010, pp. 425–439.

[65] M.W. Chekol and H. Stuckenschmidt, Time-Aware Probabilistic Knowledge Graphs, in: *26th International Symposium on Temporal Representation and Reasoning (TIME'19)*, J. Gamper, S. Pinchinat and G. Sciavicco, eds, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 147, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 8:1–8:17. ISSN 1868-8969. ISBN 978-3-95977-127-6. doi:10.4230/LIPIcs.TIME.2019.8.

[66] 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.

[67] R. Taelman, M. Vander Sande, J. Van Herwegen, E. Mannens and R. Verborgh, Triple storage for random-access versioned querying of RDF archives, *Journal of Web Semantics* **54** (2019), 4–28. doi:10.1016/j.websem.2018.08.001.

[68] L. Bayoudhi, N. Sassi and W. Jaziri, Towards a semantic querying approach for a multi-version OWL 2 DL ontology, *International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM)* **13** (2021), 80–90.

[69] 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.

[70] 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.

[71] M. Vander Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens and R. Van de Walle, R&Wbase: git for triples, in: *Proceedingso of the 6th Workshop on Linked Data on the Web*, 2013.

[72] 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'14)*, Citeseer, 2014.

[73] 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. doi:10.1016/j.websem.2018.08.002.

[74] D. Petkovic, Temporal data in relational database systems: A comparison, in: *New Advances in Information Systems and Technologies*, Springer, 2016, pp. 13–23.

[75] F. Orlandi, D. Graux and D. O'Sullivan, How many stars do you see in this constellation?, in: *The Semantic Web: ESWC 2020 Satellite Events*, Springer, 2020, pp. 175–180.

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