Semantic Web 0 (0) 1
IOS Press

Expressive Querying and Scalable Management of Large RDF Archives

Olivier Pelgrin [a,*], Ruben Taelman [b], Luis Galárraga [c] and Katja Hose [d,a]
[a] *Aalborg University, Denmark*
E-mail: olivier@cs.aau.dk
[b] *Ghent University, Belgium*
E-mail: ruben.taelman@ugent.be
[c] *INRIA, France*
E-mail: luis.galarraga@inria.fr
[d] *TU Wien, Austria*
E-mail: katja.hose@tuwien.ac.at

Abstract. The proliferation of large and ever-growing RDF datasets has sparked a need for robust and performant RDF archiving systems. In order to tackle this challenge, several solutions have been proposed throughout the years, including archiving systems based on independent copies, time-based indexes, and change-based approaches. In recent years, modern solutions combine several of the above mentioned paradigms. In particular, aggregated changesets of time-annotated triples have showcased a noteworthy ability to handle and query relatively large RDF archives. However, such approaches still suffer from scalability issues, notably at ingestion time. This makes the use of these solutions prohibitive for large revision histories. Furthermore, applications for such systems remain often constrained by their limited querying abilities, where SPARQL is often left out in favor of single triple-pattern queries. In this paper, we propose a hybrid storage approach based on aggregated changesets, snapshots, and multiple delta chains that additionally provides full querying SPARQL on RDF archives. This is done by interfacing our system with a modified SPARQL query engine. We evaluate our system with different snapshot creation strategies on the BEAR benchmark for RDF archives and showcase improvements of up to one order of magnitude in ingestion speed compared to state-of-the-art approaches, while keeping competitive querying performance. Furthermore, we demonstrate our SPARQL query processing capabilities on the BEAR-C variant of BEAR. This is, to the best of our knowledge, the first openly-available endeavor that provides full SPARQL querying on RDF archives.

Keywords: RDF, Archiving, SPARQL

1. Introduction

The exponential growth of RDF data and the emergence of large collaborative knowledge graphs have driven research in the field of efficient RDF archiving [1, 2], the task of managing the change history of RDF graphs. This offers invaluable benefits to both data maintainers and consumers. For data maintainers, RDF archives serve as the foundation for version control [3]. This not only enables data mining tasks, such as identifying temporal and correction patterns [4], but in general opens the door to advanced data analytics of evolving graphs [5–9]. For data consumers, RDF archives provide a valuable means to access historical data and delve into the evolution of specific

[*]Corresponding author. E-mail: olivier@cs.aau.dk.

knowledge domains [10–12]. In essence, these archives offer a way to query past versions of RDF data, allowing for a deeper understanding of how knowledge has developed over time.

However, building and maintaining RDF archives presents substantial technical challenges, primarily due to the large size of contemporary knowledge graphs. For instance, DBpedia, as of April 2022, comprises 220 million entities and 1.45 billion triples[1]. The number of changes between consecutive releases can reach millions [2]. Yet, dealing with large changesets is not the sole obstacle faced by state-of-the-art RDF archive systems. Efficient querying also remains an open challenge, since support for full SPARQL is rare among existing systems [1, 2].

To address these challenges, we propose an approach for ingesting, storing, and querying long revision histories on large RDF archives. Our approach, which combines multiple snapshots and delta chains, has been previously detailed in our prior work [13] and outperforms existing state-of-the-art systems in terms of ingestion time and query runtime for archive queries on single triple patterns. This paper builds on top of this prior work and extends it with the following contributions:

- The design and implementation of a full SPARQL querying middle-ware on top of our multi-snapshot RDF archiving engine.
- A novel representation for the versioning metadata stored in our indexes. This representation is designed to improve ingestion time and disk usage without increasing query runtime.
- An evaluation of the two aforementioned contributions in addition to an extended evaluation of our prior work with additional baselines.

In general, we evaluate the effectiveness of our enhanced approach using the BEAR benchmark [1], our results demonstrate remarkable improvements, namely, up to several order of magnitude faster ingestion times, reduced disk usage, and overall improved querying speed compared to existing baselines. Additionally, we showcase our new SPARQL querying capabilities on the BEAR-C variant of the BEAR benchmark. This is the first time, to the best of our knowledge, that a system complete this benchmark suite and publish its results.

The remainder of this paper is organized as follows: Section 2 explains the background concepts used throughout the paper. In Section 3 we discuss the state of the art in RDF archiving, in particular existing approaches to store and query RDF archives. In Section 4, we detail our storage architecture that builds upon multiple delta chains, and proposes several strategies to handle the materialization of new delta chains. Section 5 details the algorithms employed for processing single triple patterns over our multiple-delta-chain- architecture, while Section 6 describes our new versioning metadata serialization method, and showcases how it improves ingestion times and disk usage. In Section 7, we explain our solution to support full SPARQL 1.1 archives queries on top of our storage architecture. Section 8 describes our extensive experimental evaluation. The paper concludes with Section 9, which summarizes our contributions and discusses future research directions.

2. Preliminaries

An *RDF graph* G (also called a *knowledge graph*) consists of a set of triples $\langle\, s, p, o \,\rangle$ with subject $s \in \mathcal{I} \cup \mathcal{B}$, predicate $p \in \mathcal{I}$, and object $o \in \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$, where $\mathcal{I}$ is a set of IRIs, $\mathcal{L}$ is a set of literals, and $\mathcal{B}$ is a set of blank nodes [14]. RDF graphs are queried using SPARQL [15], whose building blocks are *triple patterns*, i.e., triples that allow variables (prefixed with a '?') in any position, e.g., $\langle\, ?x, \mathit{cityIn}, \mathit{USA} \,\rangle$ matches all American cities in G.

An *RDF archive* $\mathcal{A}$ is a temporally ordered collection of RDF graphs that represents all the states of the graph throughout its update history. This can be formalized as $\mathcal{A} = \{G_0, \ldots, G_k\}$, with G_i being the graph at version (or revision) $i \in \mathbb{Z}_{\geqslant 0}$. The transition from G_{i-1} to version G_i is implemented through an update operation $G_i = (G_{i-1} \setminus u_i^-) \cup u_i^+$, where u_i^+ and u_i^- are disjoint sets of added and deleted triples. We call the pair $u_i = \langle u_i^+, u_i^- \rangle$ a *changeset* or *delta*. We can generalize changesets to any pair of versions, i.e., $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$ defines the changes between versions i and j. When a triple $\langle\, s, p, o \,\rangle$ is present in a version i of the archive, we write it as a quad $\langle\, s, p, o, i \,\rangle$. We summarize the notations used throughout the paper in Table 1.

[1]https://www.dbpedia.org/resources/knowledge-graphs/

$\langle\ s, p, o\ \rangle$	an RDF triple
$\langle\ s, p, o, i\ \rangle$	a versioned triple, i.e., an RDF quad
G	an RDF graph
G_i	the i-th version or revision of graph G
$\mathcal{A}$	an RDF graph archive
$u_i = \langle u_i^+, u_i^- \rangle$	a changeset with sets of added and deleted triples for version i
$u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$	the changeset between graph versions i and j ($j > i$)

Table 1

Notations summary.

3. Related Work

In this section, we discuss the current state of RDF archiving in the literature. We will first present how RDF archives are usually queried. We then discuss the existing storage paradigms for RDF archives and how they perform on the different query types. We conclude this section by detailing the functioning of OSTRICH, a prominent solution for managing RDF archives, which we use as baseline for our proposed design.

3.1. Querying RDF Archives

In contrast to conventional RDF, the presence of multiple versions within an RDF archive requires the definition of novel query categories. Some categorizations for versioning queries over RDF Archives have been proposed in the literature [1, 8, 16]. In this work, we build upon the proposal of Fernández et al. [1] due to its greater adoption by the community. They identify five query types, which we explain in the following through a hypothetical RDF archive that stores information about countries and their diplomatic relationships:

– **Version Materialization (VM).** These are standard SPARQL queries run against a single version i, e.g., $\langle\ ?s, \mathit{type}, \mathit{Country}, 5\ \rangle$ returns the countries present in version $i = 5$.
– **Delta Materialization (DM).** These are queries defined on changesets $u_{i,j} = \langle u_{i,j}^+, u_{i,j}^- \rangle$, e.g., the query asking for the countries added between versions $i = 3$ and $j = 5$, which implies to run $\langle\ ?s, \mathit{type}, \mathit{Country}\ \rangle$ on $u_{3,5}^+$.
– **Version Query (V).** These are standard SPARQL queries that provide results annotated with the versions where those results hold. An example is $\langle\ ?s, \mathit{type}, \mathit{Country}, ?v\ \rangle$, which returns pairs $\langle\ \mathit{country}, \mathit{version}\ \rangle$.
– **Cross-version (CV).** CV queries combine results from multiple versions, for example: *which of the countries in the latest version have diplomatic relationships with the countries in revision 0?*
– **Cross-delta (CD).** CD queries combine results from multiple changesets, for example: *in which versions were the most countries added?*

Both CV and CD queries build upon the other types of queries, i.e., V and DM queries. Therefore, full support for VM, DM, and V queries is the minimum requirement for applications relying on RDF archives. Papakonstantinou et al. [16], on the other hand, propose a categorisation into two main categories, *version* and *delta* queries, which can be of any of three types: *Materialization*, *Single version*, or *Cross Version*. As such, *materialization* queries request the full set of triples present in a given version, while *single version* queries are answered by applying restrictions or filters on the triples of that version. *Cross version* queries instead need access to multiple versions of the data. In practice, the categorizations of [16] and [1] are equally expressive. Polleres et al. [8] propose two categories of versioned queries: *Version Materialization* and *Delta Materialization*. These are identical to the categories used by Fernández et al. [1] described above. Queries applied to multiple versions are categorized as *Cross Version*, which includes the *Version Queries (V)* from Fernández et al. [1]'s classification.

SPARQL is the recommended W3C standard to query RDF data, however adapting versioned query types to standard SPARQL remains one of the main challenges in RDF archiving. Indeed, current RDF archiving systems are often limited to queries on single triple patterns [2, 17]. This puts the burden of combining the results of single triple pattern queries onto the user, further raising the barrier for the adoption of RDF archiving systems. While support for standard SPARQL on RDF archives is nonexistent, multiple endeavors have proposed either novel query languages or temporal extensions for SPARQL. Fernández et al. [1] for example, formulates their categories of versioning queries using the AnQL [18] query language. AnQL is a SPARQL extension operating on quad patterns instead of triples pattern. The additional component can be mapped to any term $u \in \mathcal{I} \cup \mathcal{L}$, and is used to represent time objects such as timestamps or version identifiers. Other works have focused on expressing SPARQL queries with temporal constraints [19–21]. T-SPARQL for example, takes inspiration from the TSQL2 language and can match triples annotated with validity timestamps. SPARQL-LTL [22] on the other hand, supports triples annotated with version numbers, which are implemented as named graphs.

All in all, there is currently no widely accepted standard for the representation of versioned queries over RDF archives within the community. Instead, current proposals are often tailored to specific use cases and applications, and no standardization effort has been proposed.

In this work, we formulate complex queries on RDF archives as standard SPARQL queries, but we assume that revisions in the archive are modeled logically as RDF graphs named according to a particular convention (explained in Section 7). This design decision makes our solution suitable for any standard RDF/SPARQL engine with support for named graphs.

3.2. Main Storage Paradigms for Storing RDF Archives

Several solutions have been proposed for storing the history of RDF graphs efficiently. We review the most prominent approaches in this section and refer the reader to [2] for a detailed survey. We distinguish three main design paradigms: independent copies (IC), change-based solutions (CB), and timestamp-based systems (TB).

Independent copies (IC) systems, such as SemVersion [23], implement full redundancy: all triples present in a version i are stored as an independent RDF graph G_i. While IC approaches excel at executing VM queries, DM and V queries suffer from the need to execute queries independently across multiple versions, requiring subsequent result set integration and filtering. Similarly, IC approaches are impractical for today's knowledge graphs because of their prohibitive storage footprint. This fact has shifted the research trend toward CB and TB systems.

Change-based (CB) solutions store their initial version as a full snapshot and subsequent versions G_j as *changesets* $u_{i,j}$ (also called deltas) where $j > i$. We call a sequence of changesets – representing an arbitrary sequence of versions – and its corresponding reference revision i, a *delta chain*. CB approaches usually require less disk space than IC architectures and are optimal for DM queries – at the expense of efficiency for VM queries. R43ples [24] is a prominent example of a system employing a CB storage paradigm.

Timestamp-based (TB) systems store triples annotated with versioning metadata such as temporal validity intervals, addition/deletion timestamps, and list of valid versions, among others. This makes TB solutions usually well suited to efficiently answer V queries, while VM and DM queries still necessitate further processing. The storage efficiency of TB solutions depends on the representation chosen to serialize the versioning metadata. TB systems notably include x-RDF-3X [25], Dydra [26], and v-RDFCSA/v-HDT [27]. The latter has been shown to provide excellent storage efficiency and query performance. However, this is achieved at the cost of flexibility, by limiting itself to storing and indexing existing full archives, and leaving out the possibility of subsequent updates. Moreover, no implementation of their method has been made publicly available at the time of writing. Dydra is only available through their cloud service and is otherwise closed source, which makes a fair comparative evaluation in an independent and controlled setup impossible.

Finally, recent approaches borrow inspiration from more than one paradigm. QuitStore [3], for instance, stores the data in fragments, for which it implements a selective IC approach. This means that only modified fragments generate new copies, whereas the latest version is always materialized in main memory. OSTRICH [17] proposes a customized CB based approach based on aggregated changesets with version-annotated triples. This approach has shown great potential both in terms of scalability and query performance [2, 17]. For this reason, our solutions use this system as underlying architecture. We describe OSTRICH in detail in the following section.

3.3. OSTRICH's Architecture and Storage Paradigm

Change-based (CB) approaches work by storing the first revision of an archive as a fully materialized snapshot, and subsequent versions as deltas $u_{i,j}$ with $i = j - 1$ (see Figure 2a for an illustration). This approach provides better storage efficiency than approaches based on independent copies (IC) as long as the deltas between versions are not larger than the materialized revisions. Also, CB approaches provide good query performance for delta-materialization (DM) queries. However, some queries can become increasingly expensive as the delta chain grows because each delta is relative to the previous one. For instance, version-materialization (VM) queries require the iteration of the full delta chain, up to the target version, in order to reconstruct the needed data on which the query will be executed. Similarly, version queries (V) need to iterate over the entire delta chain to provide a complete list of the valid version numbers for each solution of a query. On long delta chains, this process can become prohibitive.

As such, OSTRICH [17] proposes instead the use of *aggregated delta chains*, as illustrated in Figure 2b. An aggregated delta chain works by storing an initial reference snapshot, as conventional delta chains do, and then storing subsequent versions as deltas $u_{0,j}$ with 0 being the reference version. Such an approach allows for a more efficient version materialization process compared to conventional delta chains, since only one delta needs to be considered to reconstruct any version.

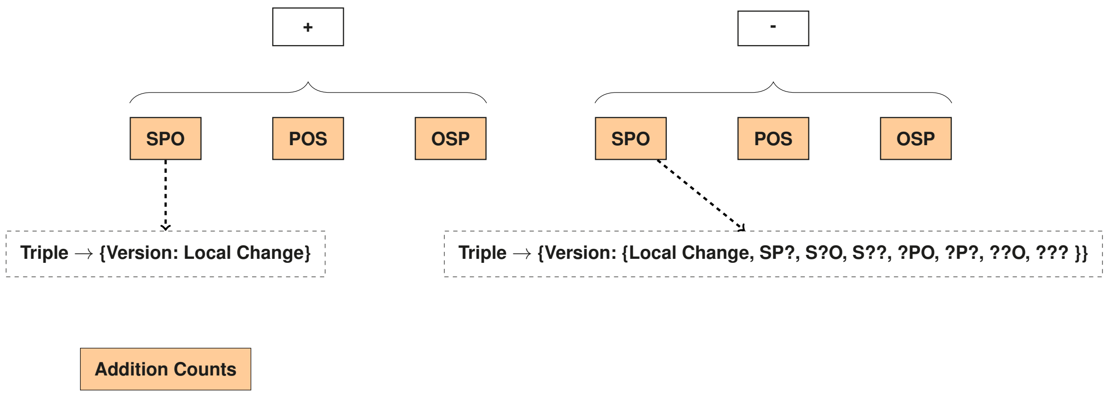

+
-
SPO
POS
OSP
SPO
POS
OSP
Triple → {Version: Local Change}
Triple → {Version: {Local Change, SP?, S?O, S??, ?PO, ?P?, ??O, ??? }}
Addition Counts

Fig. 1. OSTRICH Delta Chain Storage Overview

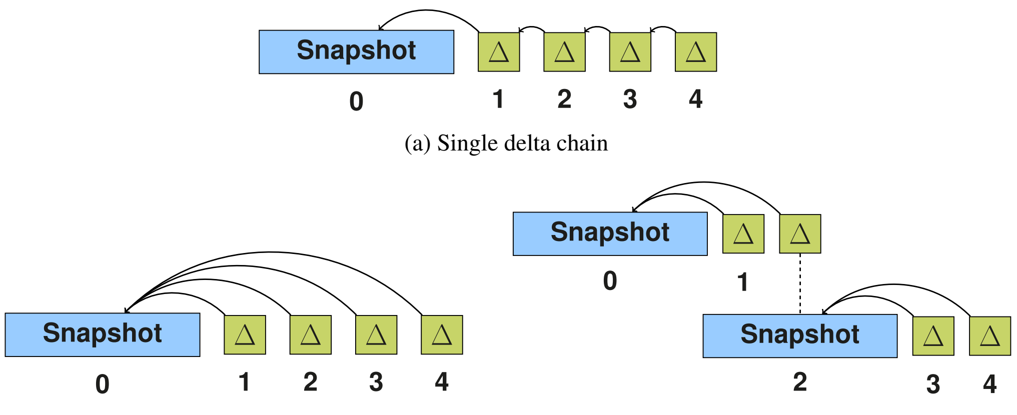
OSTRICH stores the initial snapshot in a HDT file [28] , which provides a dictionary and a compressed, yet easy to read, serialization for the triples. As standard for RDF engines, the dictionary maps RDF terms to integer identifiers, which are used in all data structures for efficiency.

In contrast to the initial snapshot, the delta chain is stored in two separate triple stores, one for additions and one for deletions. Each triple store consists of three differently ordered indexes (SPO, POS, and OSP), as illustrated in Figure 1. Each triple is annotated with versioning metadata, which is used for several purposes: First, this reduces data redundancy in the delta chain, allowing each triple to be stored only once. Secondly, the metadata can be used to accelerate query processing. As seen in Figure 1, this metadata differs between additions and deletions triples. All triples feature a collection of mappings from version to a local change flag, which indicates whether the triple reverts a previous change in the delta chain. For example, consider the quad $\langle\, s, p, o, 0 \,\rangle$ and a changeset in revision 2 that removes the triple $\langle\, s, p, o \,\rangle$. If a subsequent change adds this triple again, say in revision 4, then the entry for triple $\langle\, s, p, o \,\rangle$ will contain the entries $\{4 : \mathit{true}\}$ and $\{2 : \mathit{false}\}$ for the addition and deletion indexes, respectively. This flag can be used to filter triples early during querying. Since deltas in OSTRICH are aggregated, entries in the versioning metadata are copied for each version where a change is relevant. From the previous example, the entry $\{2 : \mathit{false}\}$ in the deletion index will also exist for revision 3 as $\{3 : \mathit{false}\}$, since the triple is deleted in both $u_{0,2}$ and $u_{0,3}$. This can create redundancies, especially in long delta chains.

We notice that deleted triples are associated to an additional vector that stores the triple's relative position in the delta for every possible triple pattern order. This allows OSTRICH to optimize for offset queries, and enables fast computation of deletion counts for any triple pattern and version. Since HDT files cannot be edited, the delta chain also has its own writable dictionary for the RDF terms that were added after the first snapshot. More details about OSTRICH's storage system can be found in the original paper [17].

OSTRICH supports VM, DM, and V queries on single triple patterns natively. Aggregated changesets have been shown to speed up VM and DM queries significantly w.r.t. a standard CB approach. As shown in [2, 17], OSTRICH is the only available solution that can handle histories for large RDF graphs, such as DBpedia. That said, scalability still remains a challenge for OSTRICH because aggregated changesets grow monotonically. This leads to prohibitive ingestion times for large histories [2, 29] – even when the original changesets are small. In this paper, we build upon OSTRICH and propose a solution to this problem.

4. Storing Archives with Multiple Delta Chains



(a) Single delta chain

(b) Single aggregated delta chain

(c) Multiple aggregated delta chains

Fig. 2. Delta chain architectures

4.1. *Multiple Delta Chains*

As discussed in Section 3.3, ingesting new revisions as aggregate changesets can quickly become prohibitive for long revision histories when the RDF archive is stored in a single delta chain (see Figure 2b). In such cases, we propose the creation of a fresh snapshot that becomes the new reference for subsequent deltas. Those new deltas will be smaller and thus easier to build and maintain. They will also constitute a new delta chain as depicted in Figure 2c.

While creating a fresh snapshot with a new delta chain should presumably reduce ingestion time for subsequent revisions, its impact on query efficiency remains unclear. For instance, V queries will have to be evaluated on multiple delta chains, becoming more challenging to answer. In contrast, VM queries defined on revisions already materialized as snapshots should be executed much faster. Storage size and DM response time may be highly dependent on the actual evolution of the data. If a new version includes many deletions, fresh snapshots may be smaller than aggregated deltas. We highlight that in our proposed architecture, revisions stored as snapshots also exist as aggregated deltas w.r.t. the previous snapshot – as shown for revision 2 in Figure 2c. Such a design decision allows us to speed up DM queries as explained later.

It follows from the previous discussion that introducing multiple snapshots and delta chains raises a natural question: *When is the right moment to create a snapshot?* We elaborate on this question from the perspective of storage, ingestion time, and query efficiency next. We then explain how to query archives in a multi-snapshot setting in Section 5.

4.2. Strategies for Snapshot Creation

A key aspect of our proposed design is to determine the right moment to place a snapshot, as this decision is subject to a trade-off among ingestion speed, storage size, and query performance. We formalize this decision via an abstract *snapshot oracle* $f : \mathbb{A} \times \mathbb{U} \rightarrow \{0, 1\}$ that, given an archive $\mathcal{A} \in \mathbb{A}$ with k revisions and a changeset $u_{k-1,k} \in \mathbb{U}$, decides whether revision k should (1) or should not (0) be materialized as a snapshot – otherwise the revision is stored as an aggregated delta. The oracle can rely on the properties of the archive and the input changeset to make a decision. In the following, we describe some natural alternatives for our snapshot oracle f and illustrate them with a running example (Table 2). All strategies start with a snapshot at revision 0. Note that we do not provide an exhaustive list of all possible strategies that one could implement.

Baseline. The baseline oracle never creates snapshots, except for the very first revision, i.e. $f(\mathcal{A}, u) \equiv (\mathcal{A} = \emptyset)$. This is akin to OSTRICH's snapshot policy [17].

Periodic. A new snapshot is created when a fixed number d of versions has been ingested as aggregated deltas, that is, $f(\mathcal{A}, u) \equiv (|\mathcal{A}| \; mod \; (d + 1) = 0)$. We call d the period.

Change-ratio. Long delta chains not only incur longer ingestion times, but also higher disk consumption due to redundancy in the aggregated changesets. When low disk usage is desired, the snapshot strategy may take into account the editing dynamics of the RDF graph. This notion has been quantified in the literature via the change ratio score [1]:

$$\delta_{i,j}(\mathcal{A}) = \frac{|u^{+}_{i,j}| + |u^{-}_{i,j}|}{|G_i \cup G_j|} \tag{1}$$

Given two revisions i and j, the change ratio normalizes the number of changes (additions and deletions) between the revisions by the joint size of the revisions. If we aggregate the change ratios of all the revisions coming after a snapshot revision s, we can estimate the amount of data not materialized in the snapshot for the current delta chain. A reasonable snapshot strategy would therefore bound the aggregated change ratios $\sum_{i=s+1}^{k} \delta_{s,i}$, put differently: $f(\mathcal{A}, u) \equiv (\sum_{i=s+1}^{k} \delta_{s,i}) \geqslant \gamma$ for some user-defined budget threshold $\gamma \in \mathbb{R}_{>0}$.

Time. If we denote by t_k the time required to ingest revision k as an aggregated changeset in an archive $\mathcal{A}$, this oracle is implemented as $f(\mathcal{A}, u) \equiv (\frac{t_k}{t_{s+1}} > \theta)$, where $s + 1$ is the first revision stored as an aggregated changeset in the current delta chain. This strategy therefore creates a new snapshot as soon as ingestion time exceeds θ times the ingestion time of version $s + 1$.

Version (k)	0	1	2	3	4	5
$\lvert u^{+}_{i,j}\rvert$	100	20	20	20	20	20
$\lvert u^{-}_{i,j}\rvert$	0	10	10	10	10	10
$\sum_{i=s+1}^{k} \delta_{s,i}$	-	0.25	0.68	1.24	0.20	0.55
t_k	-	1.00	1.50	2.25	3.38	1.00
Baseline	**S**	Δ	Δ	Δ	Δ	Δ
Periodic ($d = 2$)	**S**	Δ	**S**	Δ	**S**	Δ
Change ratio ($\gamma = 1.0$)	**S**	Δ	Δ	**S**	Δ	Δ
Time ($\theta = 3.0$)	**S**	Δ	Δ	Δ	**S**	Δ

Table 2

Creation of snapshots according to the different strategies on a toy RDF graph comprised of 100 triples and 5 revisions defined by changesets. An **S** denotes a snapshot whereas a Δ denotes an aggregated changeset.

4.3. Implementation

We implemented the proposed snapshot creation strategies and query algorithms for RDF archives on top of OSTRICH [17]. We briefly explain the most important aspects of our implementation.

Storage. In OSTRICH, an RDF archive consists of a snapshot for revision 0 and a single delta chain of aggregated changesets for the upcoming revisions (Fig. 2b). The snapshot is stored as an HDT [28] file, whereas the delta chain is materialized in two stores: one for additions and one for deletions. Each store consists of 3 indexes in different triple component orders, namely SPO, OSP, and POS, implemented as B+trees. Keys in those indexes are individual triples linked to version metadata, i.e., the revisions where the triple is present and absent. Besides the change stores, there is an index with addition and deletion counts for all possible triple patterns, e.g., $\langle$ *?s*, *?p*, *?o* $\rangle$ or $\langle$ *?s*, *cityIn*, *?o* $\rangle$, which can be used to efficiently compute cardinality estimations – particularly useful for SPARQL engines.
Dictionary. As common in RDF stores [25, 30], RDF terms are mapped to an integer space to achieve efficient storage and retrieval. Two disjoint dictionaries are used in each delta chain: the snapshot dictionary (using HDT) and the delta chain dictionary. Hence, our multi-snapshot approach uses $D \times 2$ (potentially non-disjoint) dictionaries, where D is the number of delta chains in the archive.
Ingestion. The ingestion routine depends on whether a revision will be stored as an aggregated delta or as a snapshot. For revision 0, our ingestion routine takes as input a full RDF graph to build the initial snapshot. For subsequent revisions, we take as input a standard changeset $u_{k-1,k}$ ($|\mathcal{A}| = k$), and use OSTRICH to construct an aggregated changeset of the form $u_{s,k}$, where revision $s = snapshot(k)$ is the latest snapshot in the history. When the snapshot policy decides to materialize a revision s' as a snapshot, we use the aggregated changeset $u_{s,s'}$ to compute the snapshot efficiently as $G_{s'} = (G_s \setminus u^-_{s,s'}) \cup u^+_{s,s'}$.
Change-ratio estimations. The change-ratio snapshot strategy computes the cumulative change ratio of the current delta chain w.r.t. a reference snapshot s to decide whether to create a new snapshot or not. We therefore store the approximated change ratios $\delta_{s,k}$ of each revision in a key-value store. To approximate each $\delta_{s,k}$ according to Equation 1, we rely on OSTRICH's count indexes. The terms $|u^+_{s,k}|$ and $|u^-_{s,k}|$ can be obtained from the count indexes of the fully unbounded triple pattern $\langle$ *?s*, *?p*, *?o* $\rangle$ in $O(1)$ time. We estimate $|G_s \cup G_j|$ as $|G_s| + |u^+_{s,j}|$, where $|G_s|$ is efficiently provided by HDT.

The source code of our implementation as well as the experimental scripts to reproduce the results of this paper are available in a Zenodo archive[2].

5. Single Queries on Archives with Multiple Delta Chains

In the following, we detail our algorithms to compute version materialization (VM), delta materialization (DM), and V (version) queries on RDF archives with multiple delta chains. Our algorithms focus on answering single triple patterns queries, since they constitute the building blocks for answering arbitrary SPARQL queries – which we address in Section 7. All the routines described next are defined w.r.t. to an implicit RDF archive $\mathcal{A}$.

5.1. VM Queries

In a single delta chain with aggregated deltas and reference snapshot s, executing a VM query with triple pattern p on a revision i requires us to materialize the target revision as $G_i = (G_s \cup u^+_{s,i}) \setminus u^-_{s,i}$ and then execute p on G_i. In our baseline OSTRICH, $s = 0$. In the presence of multiple delta chains, we define $s = snapshot(i)$ as the revision number of i's reference snapshot in the archive's history.

Algorithm 1 provides a high level description of the query algorithm used for version materialization queries (VM). Our baseline, OSTRICH, uses a similar algorithm where $sid_i = 0$. The algorithm starts by getting the corresponding snapshot of the target version (line 2), and retrieving the matches of the query triple pattern (line 3) on the snapshot – as a stream of results. If the target version correspond to the snapshot, the query stops there and we return the results stream (line 5). Otherwise, the algorithm retrieves, from the delta chain indexes, those added and deleted triples of the target version that match the given query pattern (line 7 and 8). The deleted triples are then filtered out from the snapshot results (line 9), which are extended with the added triples (line 10). It is important to note that this process is implemented in a streaming way, and is therefore computed lazily, as needed by the query consumer.

[2]https://doi.org/10.5281/zenodo.7256988

Algorithm 1 VM query algorithm

```
1:  function QUERYVM($i, p$)                        ▷ version $i$, triple pattern $p$
2:      $sid_i \leftarrow snapshot(i)$
3:      $q_i \leftarrow query(sid_i, p)$             ▷ we query the triple pattern on the snapshot
4:      if $sid_i = i$ then                          ▷ the target version correspond to a snapshot
5:          return $q_i$
6:      end if
7:      $u^+ \leftarrow getAdditions(i, p)$
8:      $u^- \leftarrow getDeletions(i, p)$
9:      $vm_i \leftarrow q_i \setminus u^-$          ▷ filter out the deleted triples
10:     $vm_i \leftarrow vm_i \cup u^+$              ▷ add the added triples
11:     return $vm_i$
12: end function
```

5.2. DM Queries

Algorithm 2 DM query algorithm on a single delta chain

```
1:  function SINGLEDCQUERYDM($i, j, p$)             ▷ versions $i, j$, triple pattern $p$
2:      $sid_i \leftarrow snapshot(i)$
3:      if $sid_i = i$ then                          ▷ the start version correspond to the snapshot
4:          $u_j^+ \leftarrow getAdditions(j, p)$
5:          $u_j^- \leftarrow getDeletions(j, p)$
6:          return $u_j^+, u_j^-$
7:      else
8:          $u_i^+ \leftarrow getAdditions(i, p)$; $u_j^+ \leftarrow getAdditions(j, p)$
9:          $u_i^- \leftarrow getDeletions(i, p)$; $u_j^+ \leftarrow getDeletions(j, p)$
10:         $u_{i,j}^+ \leftarrow u_j^+ \setminus u_i^+$
11:         $u_{i,j}^- \leftarrow u_j^- \setminus u_j^-$
12:         return $u_{i,j}^+, u_{i,j}^-$
13:     end if
14: end function
```

Algorithm 2 describes the procedure *singleDCQueryDM* that answers DM queries with start and end versions i, j on a single delta chain for triple pattern p. This procedure is crucial for handling DM query algorithms on multiple delta chains. This algorithm consists of two cases: The first case, described on lines 3–6, is met when the start version i corresponds to the snapshot of the delta chain. When this is the case, the execution of the query is trivial as triple pattern p can be directly evaluated on the corresponding aggregated delta $u_{i,j}$. The second case, starting from line 7, deals with a start version stored as a delta. We get the changes (additions and deletions) for both the start and end versions (lines 8–9), and filter the additions and deletions so that the ones from the end version j prevail (lines 10–11). The results consist of the combination of the newly computed addition and deletion sets. In practice, this can be efficiently implemented as a sort-merge join operation where triples are emitted only when present for version j, or when their addition flag for i and j is different (in which case the flag for version j is kept).

We now turn our attention to archives with multiple delta chains. The procedure *queryDM* in Algorithm 3 describes how to answer a DM query on two revisions i and j ($i < j$) with triple pattern p on an RDF archive with multiple delta chains. The algorithm relies on two important sub-routines, which we now explain. The first one, *singleDCQueryDM*, was already described in Algorithm 2 and executes standard DM queries on single triple patterns over a single delta chain. The second routine, called *snapshotDiff*, computes the difference between the results

Algorithm 3 DM query algorithm on multiple delta chains

```
 1: function SNAPSHOTDIFF($S_i, S_j, p$)                      ▷ snapshots $S_i, S_j$, triple pattern $p$
 2:     $d \leftarrow j - i$
 3:     if $d > 1$ then
 4:         $q_i \leftarrow query(S_i, p)$; $q_j \leftarrow query(S_j, p)$
 5:         $delta \leftarrow (q_j \setminus q_i) \cup (q_i \setminus q_j)$
 6:     else
 7:         $delta = singleDCQueryDM(i, j, p)$
 8:     end if
 9:     return $delta$
10: end function
11:
12: function QUERYDM($i, j, p$)                                ▷ versions $i$, $j$, triple pattern $p$
13:     $sid_i \leftarrow snapshot(i)$; $sid_j \leftarrow snapshot(j)$
14:     if $sid_i = sid_j$ then                                ▷ $i$ and $j$ are in the same delta-chain
15:         $delta \leftarrow singleDCQueryDM(i, j, p)$
16:     else                                                   ▷ $i$ and $j$ are not in the same delta-chain
17:         $u_{si,sj} \leftarrow snapshotDiff(sid_i, sid_j, p)$
18:         $u_{si,i}, u_{sj,j} \leftarrow \emptyset$
19:         if $i \neq sid_i$ then                             ▷ test if version $i$ is a delta
20:             $u_{si,i} \leftarrow singleDCQueryDM(sid_i, i, p)$
21:         end if
22:         if $j \neq sid_j$ then                             ▷ test if version $j$ is a delta
23:             $u_{sj,j} \leftarrow singleDCQueryDM(sid_j, j, p)$
24:         end if
25:         $u_{i,sj} \leftarrow mergeBackwards(u_{si,i}, u_{si,sj})$
26:         $(u_i, u_j) \leftarrow mergeForward(u_{i,sj}, u_{sj,j})$
27:     end if
28:     return $u_i, u_j$
29: end function
```

of p on two reference snapshots S_i and S_j. It works by first testing if the delta chains of S_i and S_j are not consecutive (line 2 in Algorithm 3). If they are not, *snapshotDiff* implements a set-difference between p's results on S_i and S_j (lines 4–5). In case the snapshots define consecutive delta chains, we leverage the fact that S_j also exists as an aggregated delta w.r.t. S_i (see Section 4.1). We can therefore treat this case efficiently as a standard DM query via *singleDCQueryDM* (line 7).

We now have the elements to explain the main DM query procedure (*queryDM*). First, the procedure checks whether both revisions are in the same delta chain, i.e., if they have the same reference snapshot (line 14). If so, the problem boils down to a single delta chain DM query that can be answered with *singleDCQueryDM* (line 15). Otherwise, we invoke the routine *snapshotDiff* on the reference snapshots (line 17) to compute the results' difference between the delta chains. This is denoted by $u_{si,sj}$.

If revisions i and j are not snapshots themselves, lines 20 and 23 compute the changes between the target versions and their corresponding reference snapshots – denoted by $u_{si,i}$ and $u_{sj,j}$. The last steps, i.e., lines 25 and 26, merge the intermediate results to produce the final output. First, the routine *mergeBackwards* merges $u_{si,sj}$, i.e., the changes between the two delta chains, with $u_{si,i}$, i.e., the changes within the first delta chain. This routine is designed as a regular sorted merge because triples are already sorted in the OSTRICH indexes. Unlike a classical merge routine, *mergeBackwards* inverts the flags of the changes present in $u_{si,i}$ but not in $u_{si,sj}$. Indeed, if a change in $u_{si,i}$ did not survive to the next delta chain, it means it was later reverted in revision sid_j. The result of this operation are therefore the changes between revisions i and sid_j, which we denote by $u_{i,sj}$. The final merge step, *mergeForward*, combines $u_{i,sj}$ with the changes in the second delta chain, i.e., $u_{sj,j}$. The routine *mergeForward* runs also a sorted merge, but now triples with opposite change flag present in both changesets are filtered from the final output as they indicate reversion operations.

5.3. V Queries

Algorithm 4 V query algorithm for single delta chains

```
1:  function SINGLEDCQUERYV($c, p$)                 ▷ $c$ is a delta chain, $p$ is a triple pattern
2:      $v \leftarrow \emptyset$
3:      $s \leftarrow getSnapshot(c)$                ▷ we get the snapshot of the delta chain
4:      $q_s \leftarrow query(s, p)$                 ▷ query the snapshot with $p$
5:      $q_a \leftarrow queryAdditions(c, p)$        ▷ query the delta chain additions with $p$
6:      for $t \in q_s \cup q_a$ do                  ▷ we iterate the triples from the queries
7:          $t_{del} \leftarrow getDeletionTriple(t, c)$   ▷ get the triple from the deletion set of the delta chain
8:          if $t_{del} \neq \emptyset$ then
9:              $t.versions \leftarrow t.versions \setminus t_{del}.versions$   ▷ filter the deleted versions from the triple valid versions
10:         end if
11:         $v.add(t)$
12:     end for
13:     return $v$
14: end function
```

Algorithm 4 describe how V queries are executed in a single delta chain setup. This is akin to how our baseline, OSTRICH, processes queries, and is used by our multiple snapshot query algorithm. We assume that each triple is annotated with its version validity, i.e. a vector of versions in which the triple exists. In OSTRICH, this is stored directly in the delta chain as versioning metadata, and therefore does not need additional computation. For the deletion triples, this metadata contains the list of versions where the triple is absent instead. The core of the algorithm iterates over the triples (line 6) that match triple pattern p in the snapshot. Each triple is queried for its existence in the deletion delta chain (line 7). If the triple exists there, then it has been deleted in a subsequent revision. We remove the versions where the triple is absent from the version validity set of the triple (line 9). Finally, we add the triple to the result set in line 11. Like the other algorithms, this routine can also be implemented in a streaming way, where each loop iteration is triggered on demand.

Algorithm 5 V query algorithm for multiple delta chains

```
1: function QUERYV($p$)                         ▷ $p$ a triple pattern
2:     $r \leftarrow \emptyset$
3:     for $c \in C$ do                         ▷ $C$ the list of delta chains
4:         $v \leftarrow singleDCQueryV(c, p)$
5:         $r \leftarrow merge(r, v)$           ▷ merge intermediate results
6:     end for
7:     return $r$
8: end function
```

Algorithm 5 describes the process of executing a V query p over multiple delta chains. This relies on the capability to execute V queries on individual delta chains via the function *singleQueryV* described above. The routine iterates over the list of delta chains (line 3), and runs *singleQueryV* on each delta chain (line 4). This gives us triples annotated with lists of versions within the range of the delta chain. At each iteration we carry out a merge step (line 5) that consists of a set union of the triples from the current delta chain and the results seen so far. When a triple is present in both sets, we merge their lists of versions.

6. Optimization of Versioning Metadata Serialization

The versioning metadata stored in the delta chain indexes is paramount to functioning of our solution and influences the multiple aspects of the system's performance. One of the current limitations of our architecture based on aggregated deltas is that it does not scale well in terms of ingestion speed and disk usage, when the number of versions grows. As we show in this section, this happens because the delta chain indexes still contain a lot of redundancy that could be removed with a proper compression scheme. In this section, we therefore discuss the limitations of the current serialization of the versioning metadata, and propose an alternative serialization scheme that brings significant improvements in terms of ingestion speed and disk usage.

6.1. Versioning Metadata Encoding

As discussed in Section 3.3, OSTRICH indexes additions and deletions in separate triple stores for each delta chain. Because aggregated deltas introduce redundancy, OSTRICH annotates triples with additional version metadata that prevents the system from storing the same triple multiple times. This versioning metadata is, in turn, leveraged during querying to filter triples based on their version validity.

Version	2	3	4	6
LC	T	T	T	T

(a) Original addition metadata in OSTRICH

Version	[2,4)	-	-	[5,∞)
LC	[2,∞)	-	-	-

(b) Compressed addition metadata

Version	2	3	4	6
LC	F	F	F	T
SP?	0	0	0	0
S?O	0	0	0	0
S??	4	6	6	0
?PO	0	0	0	1
?P?	6	8	8	0
??O	0	0	0	0
???	8	8	8	0

(c) Original deletion metadata in OSTRICH

Version	[2,5)	-	-	[6,∞)
LC	-	-	-	[6,∞)
SP?	0	-	-	-
S?O	0	-	-	-
S??	4	+2	-	-6
?PO	0	-	-	+1
?P?	6	+2	-	-8
??O	0	-	-	-
???	8	-	-	-8

(d) Compressed deletion metadata

Table 3

Representation of the versioning metadata in the indexes for arbitrary example triples in OSTRICH and compressed in our new implementation. *LC* denotes the local change flag.

In Table 3, we show side by side the versioning metadata of an arbitrary triple as stored by OSTRICH (see Section 3.3), and as stored using our proposed representation. Although triples are stored only once, we observe that the index entries still store a lot of repeated information. Consider the example in Table 3a where we can see that the local change flag is always set to true for each version. Our proposed representation compresses this information by storing intervals of versions where the value does not change. In our example, in Table 3b, the local change flag is stored as an interval [2,∞), meaning that the flag is true starting from revision 2. We highlight that the version numbers are also stored as intervals. Similarly to the uncompressed metadata, the logical model is one of a mapping from version to a local change flag. In practice, this means that if a version is not present in any of the intervals, then there is no corresponding valid local change flag, regardless of the content of the local change intervals.

Deletion indexes contain more metadata than the addition indexes. Indeed, they also store the relative position of the triple within its respective delta for all triple pattern combinations, as illustrated in Table 3c. This data can be large, especially for long delta chains, and can be both costly to create during ingestion and to deserialize during querying. OSTRICH alleviates these issues by restricting this metadata to the SPO index. We propose to replace this representation with a delta-compressed vector list, as illustrated in Table 3d. This first position vector in the list is stored plainly, as before, but we only store deltas for subsequent changes. In case where no changes occur in a given revision, like in version 4 of our example, the vector is empty. In the next section we elaborate on the implementation details of this serialization scheme.

6.2. Implementation Considerations

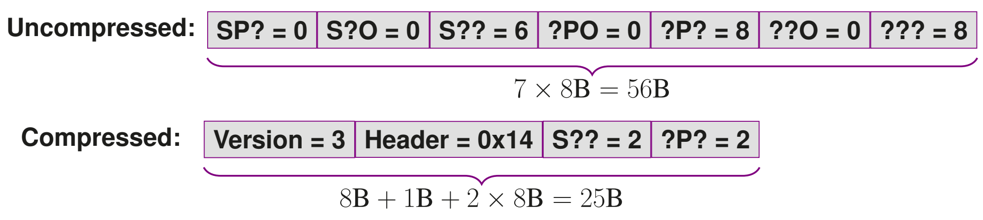


Fig. 3. Representation of the positions vectors for version 3 of our example, without and with compression.

As depicted in Figure 3d, some of the entries in the position vectors of the deleted triples can be empty. We handle those empty entries by means of a 8-bit header mask that precedes the position vector. Consider the second column vector (version 3) in our example Table 3d. This vector contains two values: +2 for the triple pattern S?? in the third position and +2 for the triple pattern ?P? in the fifth position. As such, the 8-bit header would be the string "0010100" (or 14 in hexadecimal), with 1s indicating the positions where valid values exists. Notice that this header mask is preceded by the version identifier, since this number cannot be easily inferred from the intervals. Figure 3 offers a visual representation of the serialization of the position vectors for our example 3. This compressed representation uses only 25 bytes, versus the 56 bytes required by the original serialization.

Furthermore, we highlight a key advantage of delta encoding: since most vector entries consist of small values, our representation can benefit from further compression via variable size integer encoding. The compression ultimately depends on how often the value of the positions changes between versions. Our experiments Section 8.4 demonstrate the efficacy of our approach in practice.

7. SPARQL 1.1 support for RDF Archives

Section 5 described the algorithms to answer versioned queries on single triple patterns on top of our multi-snapshot storage engine. In this section, we describe our solution to support SPARQL queries over RDF Archives. We will first discuss how to formulate versioned queries using SPARQL. We then provide details of the proposed architecture and query engine.

7.1. SPARQL Versioned Queries

As discussed in Section 2, there have been a few efforts to write versioned queries comprising multiple triple patterns. All those endeavors rely on ad-hoc extensions to the SPARQL language. For this reason, none of these extensions has reached broad community acceptance. As consequence, we have opted for a query middleware based on native SPARQL that models revisions as named graphs [24]. Versioning requirements are therefore expressed using the SPARQL *GRAPH* keyword on named graphs with URIs of the form <*version:i*>, e.g., <*version:0*> denotes the initial revision. Our SPARQL engine interprets the provided graph URI and translates it into a proper retrieval operation within the physical data model described earlier in this paper. Our approach supports the base versioned query types, namely Version Materialization (VM), Delta Materialization (DM), and Version (V) queries.

We illustrate the different queries with an example RDF Archive $\mathcal{A}$ describing information about countries. For the sake of simplicity, we assume that each version of the graph G_i represents a specific year, e.g., G_{2003} contains information about countries in the year 2003. Table 4 illustrates different versioned queries asking for country membership in the European Union (EU). First, VM queries are similar to standard SPARQL queries where the *GRAPH* clause is used to limit query evaluation to the target version. DM queries require the use of *FILTER* sub-queries to select the changes between versions, as exemplified in Table 4. The example DM query retrieves the countries that joined in revision 2004, i.e., EU members in $u^{+}_{2003,2004}$. In our design, a query on $u^{-}_{2003,2004}$ (deletions) has the same form, but the version numbers are swapped. Finally, V queries can be expressed with the *GRAPH* keyword followed by a variable.

Version Materialization (VM) Which countries were part of the EU in 2003?	**Delta Materialization (DM)** Which countries joined the EU in 2004?	**Version Query (V)** Which countries were part of the EU in each year?

SELECT * WHERE {
GRAPH <version:2003> {
?country rdf:type ex:country .
?country ex:member ex:EU .
}
}
	SELECT * WHERE {
	GRAPH <version:2004> {
	?country rdf:type ex:country .
	?country ex:member ex:EU .
	} FILTER (NOT EXISTS {
	GRAPH <version:2003> {
	?country rdf:type ex:country .
	?country ex:member ex:EU .
	}
	})
	}
		SELECT * WHERE {
		GRAPH ?version {
		?country rdf:type ex:country .
		?country ex:member ex:EU .
		}
		}

<ex:Austria> <ex:Belgium> <ex:Denmark> <ex:Finland> . . .	<ex:Cyprus> <ex:Czech Republic> <ex:Estonia> <ex:Hungary> . . .	<ex:Austria> <version:1995> <ex:Austria> <version:1996> . . . <ex:Belgium> <version:1958> . . .

Table 4
Example of SPARQL representation and results for VM, DM, and V queries using the *GRAPH* keyword.

7.2. Architecture and Implementation

In order to support full SPARQL query processing over RDF archives, we make use of the *Comunica* [31] query engine deployed on top of our multi-snapshot storage layer and our algorithms for processing single triple patterns (described in Section 5). Comunica is a scalable and extensible query engine written in TypeScript with full support for the SPARQL 1.1 language. Due to its modularity, it is a natural choice for extending our system, and initial work was already done by Taelman et al. [32] to support archives queries (although without V queries support). We have adapted this implementation to our multi-snapshot storage architecture and extended it to support V queries. We have also implemented several optimizations, notably in regard to the communication between the query engine and the storage layer. Previously, results from a triple pattern query would be buffered in OSTRICH until all results have been gathered. Because Comunica is designed to work with streams of triples, this create locking in the query processing while waiting for the availability of the triples. Instead, we now buffer triples into smaller buffers of configurable size. When a buffer is filled, the triples it contains can be sent to Comunica without waiting for the remaining triples. This allows for shorter locking time in Comunica and allows us to take better advantage of its asynchronous query processing capabilities.

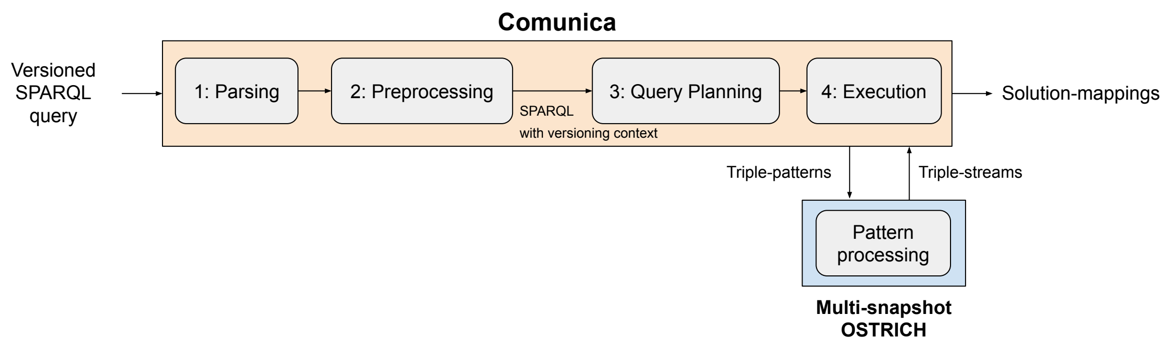


Fig. 4. SPARQL query processing pipeline

Figure 4 illustrates the query processing pipeline of our solution for SPARQL queries on RDF archives. There are two main software components interacting with each other: the first is the storage layer consisting of our multi-snapshot version of OSTRICH (see Sections 4 and 5), and the second is the Comunica [31] query engine, which includes several modules.

A versioned SPARQL query like the ones in Table 4 (Section 7.1), is first transformed back to a graph- and filter-free SPARQL query, i.e. without the special GRAPH URIs and/or FILTER clauses, and annotated with a versioning context. This versioning context depends on the query type (e.g., revisions for VM queries, changesets for DM queries) and the target version(s) when relevant, and is used to select the type of triple pattern queries to send for execution by OSTRICH. The communication between Comunica and OSTRICH is done through a NodeJS native addon. Our implementation is open source[3][4] and a demonstration system is available at [33].

8. Experiments

To determine the effectiveness of our multi-snapshot approach for RDF archiving, we evaluate the four proposed snapshot creation strategies described in Section 4 along three dimensions: ingestion time (Section 8.2.1), disk usage (Section 8.2.2), and query runtime for VM, DM, and V queries (Section 8.3). Thereafter, in Section 8.4, we delve into the effectiveness of our versioning metadata representation described in Section 6. This is done by comparing its performance against the original representation – across the three aforementioned evaluation dimensions. Section 8.5 concludes our experiments with an evaluation of our full SPARQL query capabilities.

8.1. Experimental Setup

We resort to the BEAR benchmark for RDF archives [1] for our evaluation. BEAR comes in three flavors: BEAR-A, BEAR-B, and BEAR-C, which comprise a representative selection of different RDF graphs and query loads. Table 5 summarizes the characteristics of the experimental datasets and query loads. Due to the very long history of BEAR-B instant, OSTRICH could only ingest one third of the archive's history (7063 out of 21046 revisions) after one month of execution – before crashing. In a similar vibe, OSTRICH took one month to ingest the first 18 revisions (out of 58) of BEAR-A. Despite the dataset's short history, changesets in BEAR-A are in the order of millions of changes, which also makes ingestion intractable in practice. On these grounds, the original OSTRICH paper [17] excluded BEAR-B instant from its evaluation, and considered only the first 10 versions of BEAR-A. Multi-snapshot solutions, on the other hand, allow us to manage these datasets. We provide, nevertheless, the results of the baseline strategy (OSTRICH) for entire history of BEAR-A. We emphasize, however, that ingesting this archive was beyond what would be considered reasonable for any use case: it took more than five months of execution. We provide those results as a baseline (although an easy one) to highlight the challenge of scaling to large datasets. All our experiments were run on a Linux server with a 16-core CPU (AMD EPYC 7281), 256 GB of RAM, and 8TB hard disk drive.

		BEAR-B			
	BEAR-A	Daily	Hourly	Instant	BEAR-C
# versions	58	89	1299	21046	32
$\|G_i\|$'s range	30M - 66M	33K - 44K	33K - 44K	33K - 44K	485K - 563K
$\overline{\|\Delta\|}$	22M	942	198	23	568K
# queries	368	62 (49 ?P? and 13 ?PO)			11 (SPARQL)

Table 5

Dataset characteristics. $|G_i|$ is the size of the individual revisions, $\overline{|\Delta|}$ denotes the average size of the individual changesets $u_{k-1,k}$.

[3]https://github.com/dkw-aau/ostrich-node

[4]https://github.com/dkw-aau/comunica-feature-versioning

We evaluate the different strategies for snapshot creation detailed in Section 4.2 along ingestion speed, storage size, and query runtime. Except for our baseline (OSTRICH), all our strategies are defined by parameters that we adjust according to the dataset:
Periodic. This strategy is defined by the period d. We set $d \in \{2, 5\}$ for BEAR-A and BEAR-C, $d \in \{5, 10\}$ for BEAR-B daily, $d \in \{50, 100\}$ for BEAR-B hourly, and $d \in \{100, 500\}$ for BEAR-B instant. Values of d were adjusted per dataset experimentally w.r.t. the length of the revision history and the baseline ingestion time. High periodicity, i.e., smaller values for d, lead to more and shorter delta chains.
Change-ratio (CR). This strategy depends on a cumulative change-ratio budget threshold γ. We set $\gamma \in \{2.0, 4.0\}$ for all the tested datasets. $\gamma = 2.0$ yields 10 delta chains for BEAR-A, 9 for BEAR-C, as well as 5, 23, and 151 delta chains for BEAR-B daily, hourly, and instant, respectively. For $\gamma = 4.0$, we obtain instead 6 delta chains for BEAR-A, 6 for BEAR-C, and 3, 16, and 98 for the BEAR-B datasets.
Time. This strategy depends on the ratio θ between the ingestion time of the new revision and the ingestion time of the first delta in the current delta chain. We set $\theta = 20$ for all datasets. This produces 3, 26, and 293 delta chains for the daily, hourly, and instant variants of BEAR-B respectively, and 2 delta chains for BEAR-A. As for BEAR-C, no new delta chains are created with $\theta = 20$, and so it is equivalent to the baseline.
We omit the reference systems included with the BEAR benchmark since they are outperformed by OSTRICH [17].

8.2. Results on Resource Consumption

		BEAR-B			
	BEAR-A	Daily	Hourly	Instant	BEAR-C
High Periodicity	**13472.16**	**0.67**	**12.95**	57.89	**43.97**
Low Periodicity	14499.45	0.98	23.05	298.36	96.38
CR $\gamma = 2.0$	20505.93	1.88	13.79	77.01	75.61
CR $\gamma = 4.0$	21588.25	2.34	19.47	114.83	111.78
Time $\theta = 20$	49506.15	2.64	15.83	**43.53**	543.82
Baseline	253676.98	6.89	1514.85	-	550.90

(a) Ingestion times in minutes

		BEAR-B			
	BEAR-A	Daily	Hourly	Instant	BEAR-C
High Periodicity	72417.47	199.17	322.34	2283.43	1149.63
Low Periodicity	49995.00	102.96	**185.33**	**787.75**	**890.44**
CR $\gamma = 2.0$	47335.74	51.49	284.47	1690.38	920.46
CR $\gamma = 4.0$	**42203.04**	37.91	211.71	1175.15	939.30
Time $\theta = 20$	46614.98	38.33	325.13	3972.32	1365.10
Baseline	45965.40	**19.82**	644.50	-	1365.10

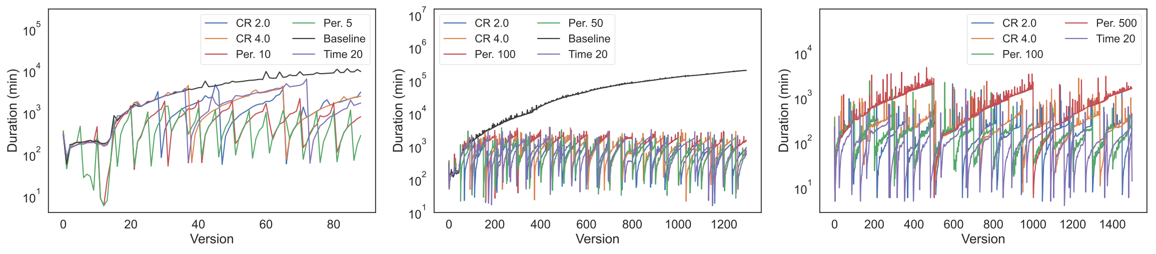
(b) Disk usage in MB

Table 6

Time and disk usage used by our different strategies to ingest the data of the BEAR benchmark datasets.

8.2.1. Ingestion Time

Table 6a depicts the total time to ingest the experimental datasets. Since we always test two different values of d for the periodic strategy on each dataset, in both Table 6a and 6b, we refer to them as "high" and "low" periodicity. This is meant to abstract away the exact parameters which vary for each dataset, so that we can focus instead on the effects of higher/lower periodicity. We remind the reader that the baseline (OSTRICH) cannot ingest BEAR-B instant, which explains its absence in Table 6a. But even when OSTRICH can ingest the entire history (in around 26 hours), a multi-snapshot strategy still incurs a significant speed-up. This becomes more significant for long histories



(a) BEAR-B Daily (b) BEAR-B Hourly (c) BEAR-B Instant

Fig. 5. Detailed ingestion times (log scale) per revision. We include the first 1500 revisions for BEAR-B instant since the runtime pattern is recurrent along the entire history.

as observed for BEAR-B hourly, where the speed-up can reach two orders of magnitude. The good performance of the high periodicity strategy and change-ratio with the smaller budget threshold $\gamma = 2.0$ suggests that shorter delta chains are beneficial for ingestion time. This is confirmed by Fig. 5, where we also notice that ingestion time reaches a minimum for the revisions following a snapshot.

8.2.2. *Disk Usage*

Unlike ingestion time, where shorter delta changes are clearly beneficial, the gains in terms of disk usage depend on the dataset as shown in Table 6b. Overall, more delta chains tend to increase disk usage. For BEAR-B daily, frequent snapshots (high periodicity $d = 5$) incur a large overhead w.r.t. the baseline because the changesets are small and the revision history is short. Similar results are observed for BEAR-A and BEAR-B instant, even though we still need multiple snapshots to be able to ingest the data. BEAR-B hourly is interesting because it shows that for long histories, a single delta chain can be inefficient in terms of disk usage. Interestingly, for BEAR-A, the change-ratio $\gamma = 4.0$ uses less storage than both the time strategy with $\theta = 20$ and the baseline, despite using more delta chains. This hints that very large aggregated deltas can be less efficient than multiple delta chains with smaller aggregated deltas. For BEAR-B instant, the good performance of the change-ratio strategies and the low periodicity strategy ($d = 500$) suggests that a few delta chains can provide significant space savings. On the other hand, the time strategy with $\theta = 20$ performs slightly worse because it creates too many delta chains. The bottom line is that redundancies in the delta chains explain the storage overhead in archives, can be caused either by very long delta chains (BEAR-B Hourly and Instant), or by large delta chains (BEAR-A), i.e., delta chains with voluminous changesets. Multiple snapshots tackle the redundancy of long delta chains naturally, but can also be beneficial for bulky delta chains, as demonstrated by the BEAR-A results with change-ratio $\gamma = 4.0$.

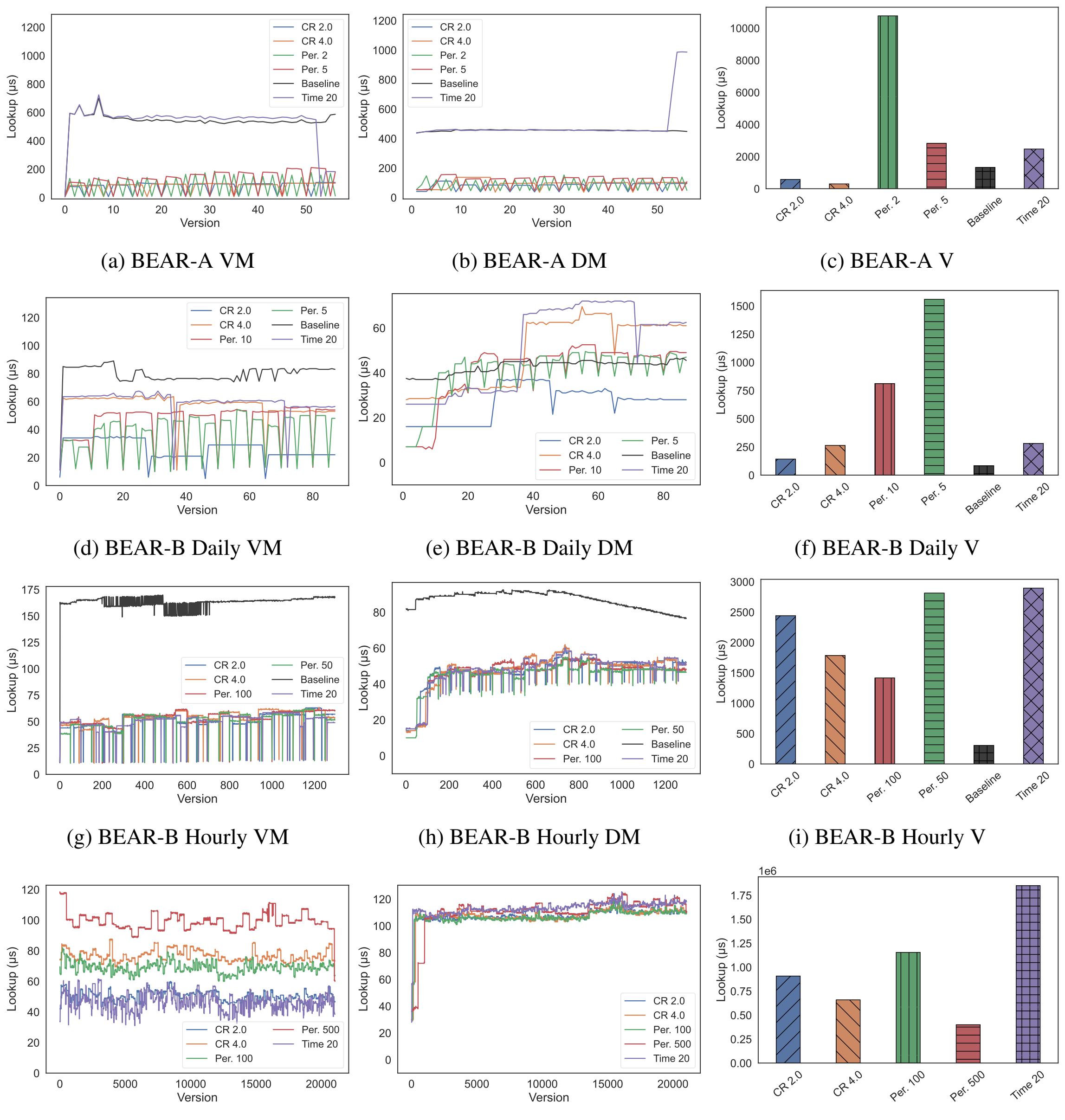
8.3. *Query Runtime Evaluation*

In this section, we evaluate the impact of our snapshot creation strategies on query runtime. We use the queries provided with the BEAR benchmark for BEAR-A and BEAR-B. These are DM, VM, and V queries on single triple patterns. Each individual query was executed 5 times and the runtimes averaged. All the query results are depicted in Figure 6.

8.3.1. *VM queries*

We report the average runtime of the benchmark VM queries for each version i in the archive. The results are depicted in Figures 6a, 6d, 6g, and 6j. We report runtimes in micro-seconds for all strategies.

Using multiple delta chains is consistently beneficial for VM query runtime, which is best when the target revision is materialized as a snapshot. When it is not the case, runtime is proportional to the *size* of the delta chain, which depends on its length and the volume of changes that must be applied to the snapshot before running the query. This is obvious for BEAR-A with the baseline strategy or with the time $\theta = 20$ strategy. The latter strategy splits the history into two imbalanced delta-chains, where one of them contains the first 53 revisions (out of 58). Both strategies are significantly outperformed by the other multi-snapshot strategies. Similar results can be observed on the BEAR-B variants, particularly for BEAR-B Hourly, where the baseline strategy is outperformed by all the strategies with more than one snapshot.



(a) BEAR-A VM (b) BEAR-A DM (c) BEAR-A V

(d) BEAR-B Daily VM (e) BEAR-B Daily DM (f) BEAR-B Daily V

(g) BEAR-B Hourly VM (h) BEAR-B Hourly DM (i) BEAR-B Hourly V

(j) BEAR-B Instant VM (k) BEAR-B Instant DM (l) BEAR-B Instant V

Fig. 6. Query results for the BEAR benchmark

8.3.2. *DM Queries*

We report for each revision i in the archive the average runtime of the benchmark DM queries on changesets $u_{0,i}$ and $u_{1,i}$. As described in Section 5.2, DM queries are executed on both the additions and deletions indexes to retrieve the full set of results for the given query pattern. Such a setup tests the query routine in all possible scenarios: between two snapshots, between a snapshot and a delta (and vice versa), and between two deltas. The results are depicted in Figures 6b, 6e, 6h, and 6k. The results show a rather mixed benefit of multiple delta chains in query runtime: highly positive for the long history of BEAR-B hourly and modest for BEAR-B daily. Overall, DM queries benefit from short delta chains as illustrated in Figure 6b and to a lesser extent by the periodic strategy with $d = 5$ in Figure 6e. All our strategies beat the baseline by a large margin on BEAR-B hourly because delta operations become very expensive as the single delta chain grows. That said, the baseline runtime tends to decrease slightly with i because the data from two distant versions tend to diverge more, which requires the engine to filter fewer results from the aggregated deltas. For BEAR-B daily, multiple delta chains may perform comparably or slightly worse – by no more than 20% – than the baseline. This happens because BEAR daily's history is short, and hence efficiently manageable with a single delta chain. In this case, the overhead of multiple snapshots and delta chains does not bring any advantage for DM queries.

8.3.3. V Queries

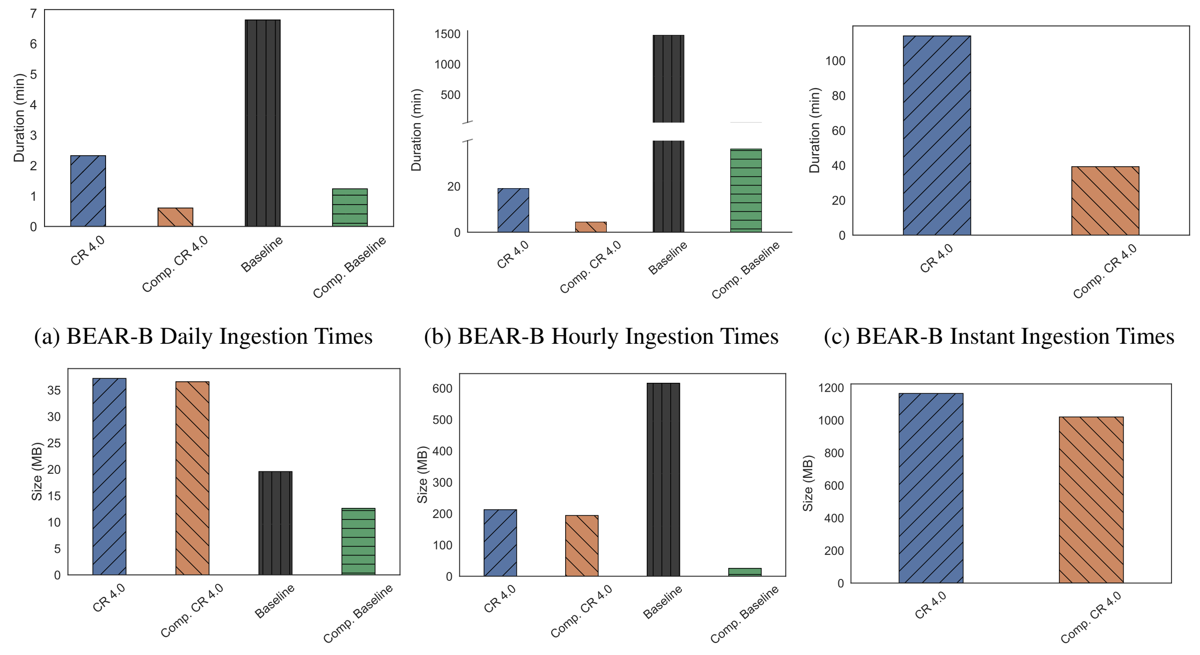
Figure 6c, 6f, 6i, and 6l show the total runtime of the benchmark V queries on the different datasets. V queries are the most challenging queries for the multi-snapshot archiving strategies as suggested by Figures 6f and 6i. As described in Algorithm 5, answering V queries requires us to query each delta chain individually, buffer the intermediate results, and then merge them. It follows that runtime scales proportionally to the number of delta chains, which means that, contrary to DM and VM queries, many short delta chains are detrimental to V query performance. The only exception is BEAR-A, where the change-ratio strategies can outperform the baseline strategy. This indicates that delta chains with very large aggregated deltas can also be detrimental to V query performance. However, BEAR-A is the only dataset showcasing such a behavior in our experiments. Nonetheless, due to their prohibitive ingestion cost, querying datasets such as BEAR-A and BEAR-B instant is only possible with a multi-snapshot solution.

8.4. Experiments on the Metadata Representation

We now evaluate our proposed encoding for versioning metadata, described in Section 6. We conduct the evaluation across the dimensions of ingestion time, disk usage, and query performance on the BEAR-B variants of the BEAR benchmark. For the sake of legibility, we apply our new encoding on archives stored using a single-snapshot strategy, i.e., the baseline OSTRICH, and one multi-snapshot strategy. We chose the change-ratio strategy with $\gamma = 4.0$ in this case since it exhibited overall good performance across the different evaluation criteria in Section 8.2. We omit the baseline strategy for BEAR-B Instant since we could not ingest it using a single snapshot.

Figure 7 shows the ingestion time and disk usage of the BEAR-B variants with the original versioning metadata encoding as used in OSTRICH and our proposed compressed encoding – denoted with the "*Comp.*" prefix in the graphs. The new encoding incurs a drastic decrease in ingestion time. This is particularly notable for the baseline strategy, where ingestion times are reduced by as much as a factor of 40 on the BEAR-B hourly dataset, as illustrated by Figure 7b. Here, the ingestion time drops from 1473 minutes to just 36 minutes. Disk usage is also notably improved with the new encoding. This is particularly obvious for the baseline strategy, because larger delta chains imply more redundancy, which in turn means more room for compression. In the most remarkable case, disk usage is reduced from 615 MB to just 25 MB for BEAR-B hourly when using the baseline strategy. The improvements in disk usage are more modest on multiple snapshot strategies, as expected from the smaller delta chains. In these cases, the snapshots account for more of the disk usage of the entire archive. For example, on BEAR-B hourly, disk usage is only reduced from 212 MB to 193 MB.

In Figure 8, we show the performance of queries with the two encodings of versioning metadata. Contrary to ingestion times and disk usage, the picture for query runtime is more nuanced. Overall, our new encoding scheme does not provide a clear advantage over the previous encoding in terms of query performance. For BEAR-B Daily, as shown in Figures 8a, 8b and 8c, the archives using the new encoding are systematically slower at resolving queries than the archives with the original encoding. As for BEAR-B Hourly, Figures 8d, 8e, and 8c, we note that queries are faster with the new encoding on the baseline strategy, but slightly slower with the change-ratio strategy. We can explain this by the large amounts of data that must be retrieved from long delta chains. In such cases, the gains obtained by reading less data – thanks to compression – outweight the costs of decompression, which translates into overall faster retrieval times. Finally, for BEAR-B Instant, the compressed representation is slower for VM queries, similar for DM queries, and slightly faster for V queries when compared to the original representation. All in all, compressing the versioned metadata reduces the redundancy in the delta chains, which goes in the same direction as using shorter delta chains. This explains why the compressed representation performs worst for querying on multiple delta chains: redundancy has already (or partially) been reduced by the use of multiple snapshots. This



(a) BEAR-B Daily Ingestion Times (b) BEAR-B Hourly Ingestion Times (c) BEAR-B Instant Ingestion Times

(d) BEAR-B Daily Disk Usage (e) BEAR-B Hourly Disk Usage (f) BEAR-B Instant Disk Usage

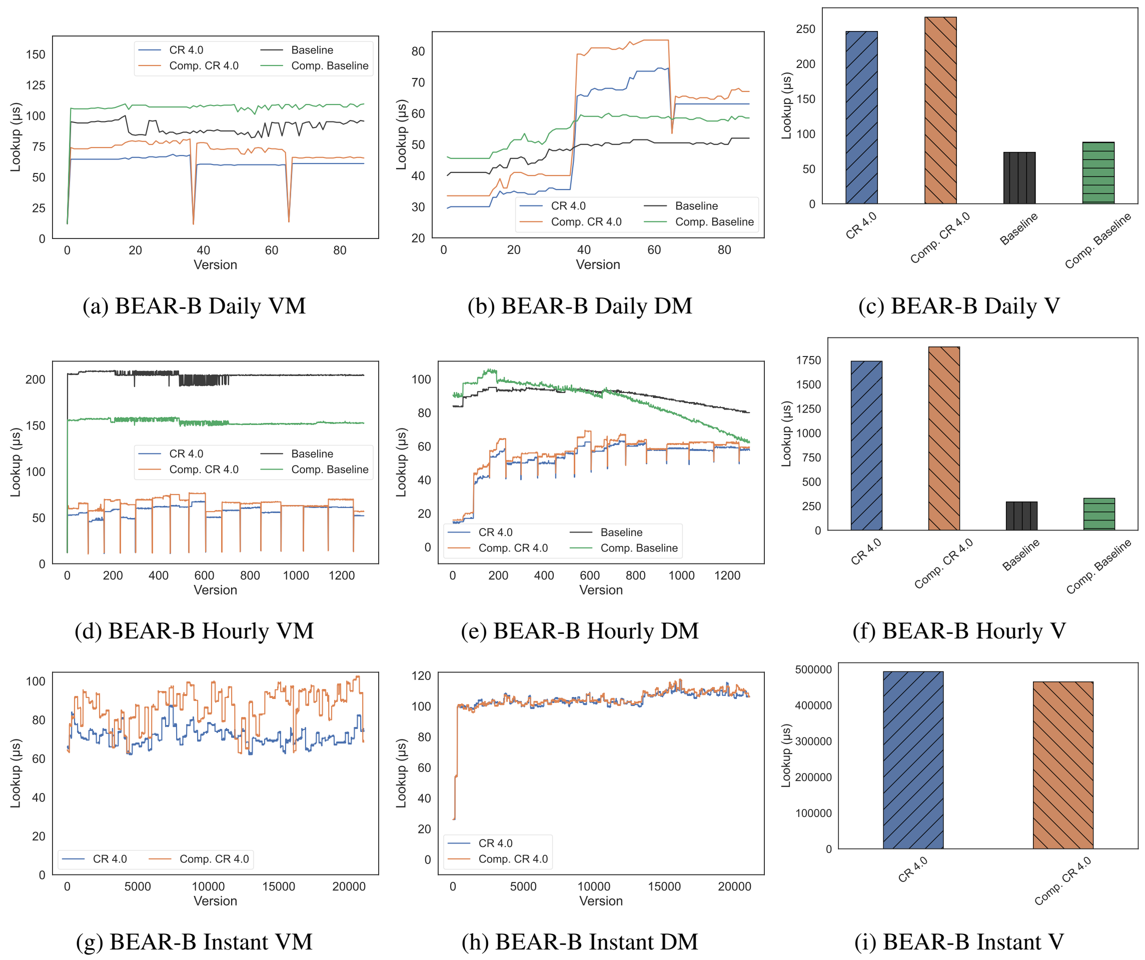
Fig. 7. Ingestion times (top row) and disk usage (bottom row) for OSTRICH and a multi-snapshot storage strategy applied on the different BEAR-B flavours with the original version metadata representation and our new compressed representation.

diminishes the gains of further compression with our approach, which comes with a performance penalty due to decompression.

8.5. SPARQL Performance Evaluation on BEAR-C

We evaluate our solution for full SPARQL support on the BEAR-C dataset. BEAR-C is based on 32 weekly snapshots of the European Open Data Portal taken from the Open Data Portal Watch project [34]. Each version contains between 485K and 563K triples, which puts BEAR-C between BEAR-B Daily and BEAR-A in terms of size. Table 5 in Section 8.1 summarizes the characteristics of the datasets. BEAR-C's query workload consists of 11 full SPARQL queries. The queries contain between 2 and 12 triple patterns and include the operators `FILTER`, `OPTIONAL` , `UNION`, `LIMIT` and `OFFSET`. In accordance with our experimental setup, we run each query 5 times and report the average runtime. Since there are no other publicly available SPARQL-compliant RDF archiving systems [2], we compare our change-ratio (CR) multi-snapshot strategy ($\gamma = 4.0$ and $\gamma = 6.0$) to the baseline. We chose the CR multi-snapshot strategy due to its good overall performance in our evaluations in Sections 8.2 and 8.3. We include the strategy CR $\gamma = 6.0$ that generates snapshots less often than CR $\gamma = 4.0$ (used in the previous experiments). This is due to the smaller size of BEAR-C when compared to more challenging datasets such as BEAR-A or BEAR-B instant. Finally, we make use of the compressed representation for the versioning metadata presented in Section 6, and evaluated earlier in this section.

Figure 9 illustrates the average execution time for each category of versioned query (VM, DM, V) on BEAR-C. The results are displayed per individual query and averaged across all revisions for VM queries, and pairs of revisions $\langle\ 1, i\ \rangle$ and $\langle\ 0, i\ \rangle$ for DM queries – in concordance with our experimental protocol. We note that the results are consistent with our single-triple patterns evaluation on the other BEAR datasets. That is, BEAR-C's relatively short history (32 versions) puts our CR stategies in disadvantage w.r.t. to the baseline strategy. The query runtime of the change-ratio strategies and the baseline are almost identical for VM queries, with the change-ratio strategies having a slight advantage for some queries (notably, queries 1, 3, and 7). For DM queries, runtimes are also closely matched between the different strategies. Overall, the CR $\gamma = 6.0$ strategy performs best on average.



(a) BEAR-B Daily VM (b) BEAR-B Daily DM (c) BEAR-B Daily V

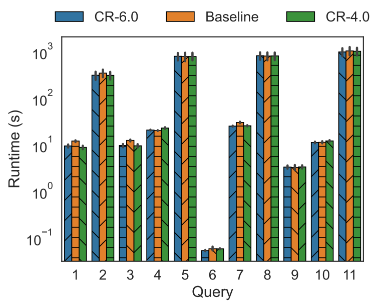
(d) BEAR-B Hourly VM (e) BEAR-B Hourly DM (f) BEAR-B Hourly V

(g) BEAR-B Instant VM (h) BEAR-B Instant DM (i) BEAR-B Instant V

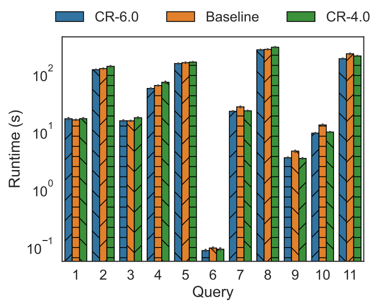
Fig. 8. Query results for the BEAR benchmark with the original version metadata encoding and the new compressed encoding.

Finally, we can observe large differences in runtimes between the different strategies on the V queries. While all strategies are closely matched overall, we can notice that the baseline strategy gets significantly outperformed on query 9 and 10, whereas it outperforms the CR strategies on query 6. The CR $\gamma = 4.0$ is notably faster than the alternatives on query 1 and 3. The overall good performance of the change-ratio strategies seems to contradict our previous findings, as V queries tend to become more expensive with more delta chains. However, we observed a similar behavior on BEAR-A in our previous experiments (Section 8.3), so to say, that the baseline strategy was outperformed by multi-snapshot strategies on V queries. This confirms the hypothesis that bulky deltas – common for BEAR-A and to a lesser extent for BEAR-C – are also detrimental to V query performance, justifying the use of multiple less voluminous delta chains.

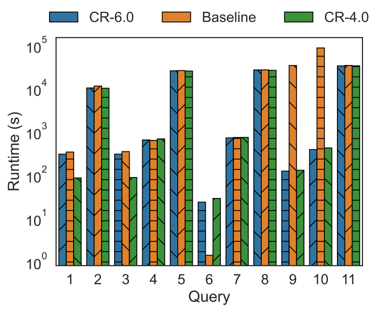
In Figure 10 we plot the runtime of VM and DM queries across revisions for queries #1 and query #2 of BEAR-C. The figures for all the other queries can be found in Appendix A. We selected those queries due to their representative runtime behavior. Query #1 has a relatively stable runtime for VM queries, with a slight increase in later revisions. Oppositely, query #2 sees a linear increase in runtime for VM queries as the target revision increases. In contrast, the runtime of DM queries is stable. We can observe that the CR strategies consistently outperform the baseline strategy on VM queries, while the baseline is faster on average for DM queries on query #1, and on par with CR $\gamma = 6.0$ for query #2. Overall, the differences between strategies are small, and vary depending on the query, as seen on Figure 9.



(a) BEAR-C VM

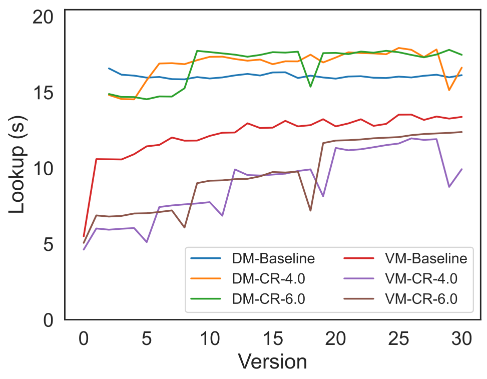


(b) BEAR-C DM

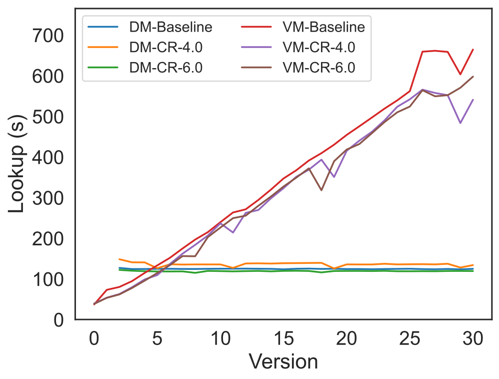


(c) BEAR-C V

Fig. 9. BEAR-C average query execution time in seconds for VM, DM, and V queries. (log scale)



(a) DM and VM runtime for BEAR-C query #1



(b) DM and VM runtime for BEAR-C query #2

Fig. 10. BEAR-C average query execution time in seconds for VM, DM, and V queries.

8.6. Discussion

We now summarize our findings in previous sections and draw a few design lessons for efficient RDF archiving.

- The disk usage and overall performance in querying of a storage approach based on aggregated delta chains depends on the amount of redundancy present in the delta chain. This redundancy can be caused by various factors, such as large changesets, long change histories, but also by the nature of the changes, e.g., changes that revert previous changes.
- It follows that for small datasets, small changesets, or relatively short histories, the overhead of multi-snapshot strategies does not pay off in terms of query runtime and disk usage. This observation is particularly striking for V queries for which runtime increases with the number of delta chains.
- Short delta chains are mostly beneficial for VM and DM queries because these query types require us to iterate over changes within two delta chains in the worst case (for DM queries). They also systematically translate into faster ingestion times. In contrast, numerous short delta chains are detrimental to storage consumption and V query performance.
- That said, when individual deltas are very bulky, as with BEAR-A and BEAR-C, multiple delta chains can be beneficial to V query performance, and can use less disk space than a single-snapshot storage strategy.
- Change-ratio strategies strike an interesting trade-off because they take into account the amount of data stored in the delta chain as criterion to create a snapshot. This ultimately has a direct positive effect on ingestion time, VM/DM querying, and storage size.
- In general, compressing the version metadata stored in the delta chain is a sensible alternative: compression increases ingestion speed and reduces disk storage. While it can increase query runtime, its impact is usually

minimal and depends on the amount of data that needs to be fetched from disk. For very large delta chains (e.g., delta chains with big deltas), compression can even be beneficial for query performance because the overhead of decompression is insignificant compared to the savings in terms of retrieved data. This observation holds promise for distributed settings.

– The performance of full SPARQL queries on RDF archives is subject to same performance trade-off as queries on single triple patterns.

The bottom line is that the snapshot creation strategy for RDF archives is subject to a trade-off among ingestion time, disk consumption, and query runtime for VM, DM, and V queries. As shown in our experimental section, there is no one-size-fits-all strategy. The suitability of a strategy depends on the application, namely the users' priorities or constraints, the characteristics of the archive (snapshot size, history length, and changeset size), and the query load. For example, implementing version control for a collaborative RDF graph will likely yield an archive like BEAR-B instant, i.e., a very long history with many small changes and VM/DM queries mostly executed on the latest revisions. Depending on the server's capabilities and the frequency of the changes, the storage strategy could therefore rely on the change ratio or the ingestion time ratio and be tuned to offer arbitrary latency guarantees for ingestion. On a different note, a user doing data analytics on the published versions of DBpedia (as done in [2]) may be confronted to a dataset like BEAR-A and therefore resort to numerous snapshots, unless their query load includes many real-time V queries.

Furthermore, we showcased our results for full SPARQL processing over RDF archives on the BEAR-C benchmark. To the best of our knowledge, this is the first approach that provides a solution for BEAR-C. Nonetheless, there are still several opportunities for future work in field of SPARQL querying over RDF archives. First, we highlight the lack of standardization for SPARQL querying on RDF archives. This has encouraged solution providers to come up with their own language extensions and ad-hoc implementations. None of them, however, has attained wide acceptance within the research and developer communities. Second, we note that the number and diversity of benchmarks for SPARQL query workloads on RDF archives is limited [35]. In BEAR, for example, only the BEAR-C dataset offers full SPARQL queries. Those 11 queries, are alas, insufficient to provide a comprehensive evaluation of the capabilities of novel systems. Alternatives, such as SPBv [16] have not seen similar adoption by the community, probably because they are not easy to deploy[5]. We expect this work to prepare the ground for the emergence of more efficient, standardized, and expressive solutions for managing RDF archives.

9. Conclusion

In this paper, we have presented a hybrid storage architecture for RDF archiving based on multiple snapshots and chains of aggregated deltas with support for full SPARQL versioned queries. We have evaluated this architecture with several snapshot creation strategies on ingestion times, disk usage, and query performance using the BEAR benchmark. The benefits of this architecture are bolstered thanks to a novel and efficient compression scheme for versioning metadata, which has yielded impressive improvements over the original serialization scheme. This has further improved the scalability of our system when handling large datasets with long version histories. All these building blocks cleared the way to introduce a new SPARQL processing system on top of our storage architecture. We are now capable of answering full SPARQL VM, DM or V queries over RDF archives.

Our evaluation shows that our architecture can handle very long version histories, at a scale not possible before with previous techniques. We used our experimental results on the different snapshot creation strategies to draw a set of design lessons that can help users choose the best storage policy based on their data and application needs. We showcased our ability to handle the BEAR-C variant of the BEAR benchmark – the first evaluation on this dataset to the best of our knowledge. This is a first step towards the support of more sophisticated applications on top of large RDF archives, and we hope it will expedite research and development in this area.

As future work, we plan to further explore different snapshot creation strategies, e.g., using machine learning, to further improve the management of complex and large RDF archives. Furthermore, we plan to investigate novel

[5]We could not use the benchmark because it relies on outdated and unsatisfiable software dependencies.

approaches in the compact representation of semantic data [36, 37], which could lead to a promising alternative to the use of B+ trees. We envision further efforts towards the practical implementation of versioning use cases of RDF, such as the implementation of version control features, like branching and tagging, into our system. Such features are paramount to real world usages of versioning software, and can benefit RDF dataset maintainers [3, 24]. With the recent popularity of RDF-star [38, 39], which can be used to capture versioning in the form of metadata, we also plan to look into recent advances in this area. Finally, the lack of an accepted standard for expressing versioning queries with SPARQL limits the wider adoption of RDF archiving systems. We aim to work towards a standardization effort, notably on a novel syntax and a formal definition of the semantics of versioned queries.

Acknowledgement

This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B, the Poul Due Jensen Foundation, and the TAILOR Network (EU Horizon 2020 research and innovation program under GA 952215). Ruben Taelman is a postdoctoral fellow of the Research Foundation – Flanders (FWO) (1202124N).

References

[1] J.D. Fernández, J. Umbrich, A. Polleres and M. Knuth, Evaluating query and storage strategies for RDF archives, *J. Web Semant.* **10**(2) (2019), 247–291. doi:10.3233/SW-180309.

[2] O. Pelgrin, L. Galárraga and K. Hose, Towards fully-fledged archiving for RDF datasets, *Semantic Web Journal* **12**(6) (2021), 903–925. doi:10.3233/SW-210434.

[3] N. Arndt, P. Naumann, N. Radtke, M. Martin and E. Marx, Decentralized Collaborative Knowledge Management Using Git, *J. Web Semant.* **54** (2019), 29–47. doi:10.1016/j.websem.2018.08.002.

[4] T. Pellissier Tanon, C. Bourgaux and F. Suchanek, Learning How to Correct a Knowledge Base from the Edit History, in: *WWW*, 2019, pp. 1465–1475. doi:10.1145/3308558.3313584.

[5] Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris and Y. Stavrakas, A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets, in: *ISWC*, Vol. 9366, 2015, pp. 495–512. doi:10.1007/978-3-319-25007-6_29.

[6] J. Brunsmann, Archiving Pushed Inferences from Sensor Data Streams, in: *International Workshop on Semantic Sensor Web*, 2010, pp. 38–46. doi:10.5220/0003116000380046.

[7] N. Gür, T.B. Pedersen, E. Zimányi and K. Hose, A foundation for spatial data warehouses on the Semantic Web, *Semantic Web* **9**(5) (2018), 557–587.

[8] A. Polleres, R. Pernisch, A. Bonifati, D. Dell'Aglio, D. Dobriy, S. Dumbrava, L. Etcheverry, N. Ferranti, K. Hose, E. Jiménez-Ruiz, M. Lissandrini, A. Scherp, R. Tommasini and J. Wachs, How Does Knowledge Evolve in Open Knowledge Graphs?, *TGDK* **1**(1) (2023), 11:1–11:59.

[9] K. Hose, Knowledge Graph (R)Evolution and the Web of Data, in: *MEPDaW@ISWC*, CEUR Workshop Proceedings, Vol. 3225, CEUR-WS.org, 2021, pp. 1–7.

[10] T. Huet, J. Biega and F.M. Suchanek, Mining History with Le Monde, in: *Workshop on Automated Knowledge Base Construction*, 2013, pp. 49–54. doi:10.1145/2509558.2509567.

[11] T.P. Tanon and F.M. Suchanek, Querying the Edit History of Wikidata, in: *ESWC*, Vol. 11762, 2019, pp. 161–166. doi:10.1007/978-3-030-32327-1_32.

[12] C. Aebeloe, G. Montoya and K. Hose, ColChain: Collaborative Linked Data Networks, in: *WWW*, 2021, pp. 1385–1396. doi:10.1145/3442381.3450037.

[13] O. Pelgrin, R. Taelman, L. Galárraga and K. Hose, Scaling Large RDF Archives To Very Long Histories, in: *ICSC*, 2023, pp. 41–48.

[14] Y. Raimond and G. Schreiber, RDF 1.1 Primer, W3C Recommendation, 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/.

[15] A. Seaborne and S. Harris, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013, http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[16] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis and Y. Roussakis, SPBv: Benchmarking Linked Data Archiving Systems, in: *BLINK/NLIWoD3@ISWC*, Vol. 1932, 2017.

[17] R. Taelman, M.V. Sande, J.V. Herwegen, E. Mannens and R. Verborgh, Triple Storage for Random-access Versioned Querying of RDF Archives, *J. Web Semant.* **54** (2019), 4–28. doi:10.1016/j.websem.2018.08.001.

[18] A. Zimmermann, N. Lopes, A. Polleres and U. Straccia, A general framework for representing, reasoning and querying with annotated Semantic Web data, *J. Web Semant.* **11** (2012), 72–95.

[19] F. Grandi, T-SPARQL: A TSQL2-like Temporal Query Language for RDF, in: *ADBIS (Local Proceedings)*, 2010, pp. 21–30.

[20] K. Bereta, P. Smeros and M. Koubarakis, Representation and Querying of Valid Time of Triples in Linked Geospatial Data, in: *ESWC*, Vol. 7882, 2013, pp. 259–274.

[21] M. Perry, P. Jain and A.P. Sheth, SPARQL-ST: Extending SPARQL to Support Spatiotemporal Queries, in: *Geospatial Semantics and the Semantic Web*, Vol. 12, 2011, pp. 61–86.

[22] V. Fionda, M.W. Chekol and G. Pirrò, Gize: A Time Warp in the Web of Data, in: *ISWC*, Vol. 1690, 2016.

[23] M. Volkel, W. Winkler, Y. Sure, S.R. Kruk and M. Synak, SemVersion: A Versioning System for RDF and Ontologies, *ESWC* (2005).

[24] M. Graube, S. Hensel and L. Urbas, R43ples: Revisions for Triples - An Approach for Version Control in the Semantic Web, in: *LDQ@SEMANTICS*, 2014.

[25] T. Neumann and G. Weikum, x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases, *PVLDB* **3**(1) (2010), 256–263. doi:10.14778/1920841.1920877.

[26] J. Anderson and A. Bendiken, Transaction-Time Queries in Dydra, in: *MEPDaW/LDQ@ESWC*, CEUR Workshop Proceedings, Vol. 1585, CEUR-WS.org, 2016, pp. 11–19.

[27] A. Cerdeira-Pena, G. de Bernardo, A. Fariña, J.D. Fernández and M.A. Martínez-Prieto, Compressed and queryable self-indexes for RDF archives, *Knowledge and Information Systems* (2023), 1–37.

[28] J.D. Fernández, M.A. Martínez-Prieto, C. Gutiérrez, A. Polleres and M. Arias, Binary RDF representation for publication and exchange (HDT), *J. Web Semant.* **19** (2013), 22–41. doi:10.1016/j.websem.2013.01.002.

[29] R. Taelman, T. Mahieu, M. Vanbrabant and R. Verborgh, Optimizing storage of RDF archives using bidirectional delta chains, *J. Web Semant.* **13** (2022), 705–734.

[30] C. Weiss, P. Karras and A. Bernstein, Hexastore: sextuple indexing for semantic web data management, *PVLDB* **1**(1) (2008), 1008–1019. doi:10.14778/1453856.1453965.

[31] R. Taelman, J. Van Herwegen, M. Vander Sande and R. Verborgh, Comunica: a Modular SPARQL Query Engine for the Web, in: *ISWC*, 2018.

[32] R. Taelman, M.V. Sande and R. Verborgh, Versioned Querying with OSTRICH and Comunica in MOCHA 2018, in: *SemWebEval@ESWC*, Vol. 927, 2018, pp. 17–23.

[33] O. Pelgrin, R. Taelman, L. Galárraga and K. Hose, GLENDA: Querying RDF Archives with Full SPARQL, in: *ESWC (Satellite Events)*, Lecture Notes in Computer Science, Vol. 13998, Springer, 2023, pp. 75–80.

[34] S. Neumaier, J. Umbrich and A. Polleres, Automated Quality Assessment of Metadata across Open Data Portals, *ACM J. Data Inf. Qual.* **8**(1) (2016), 2:1–2:29.

[35] O. Pelgrin, R. Taelman, L. Galárraga and K. Hose, The Need for Better RDF Archiving Benchmarks, in: *MEPDaW@ISWC*, CEUR Workshop Proceedings, CEUR-WS.org, 2023.

[36] R. Perego, G.E. Pibiri and R. Venturini, Compressed Indexes for Fast Search of Semantic Data, *IEEE Trans. Knowl. Data Eng.* **33**(9) (2021), 3187–3198.

[37] T. Sagi, M. Lissandrini, T.B. Pedersen and K. Hose, A design space for RDF data representations, *VLDB J.* **31**(2) (2022), 347–373.

[38] O. Hartig, Foundations of RDF★ and SPARQL★ (An Alternative Approach to Statement-Level Metadata in RDF), in: *AMW*, CEUR Workshop Proceedings, Vol. 1912, CEUR-WS.org, 2017.

[39] G. Abuoda, C. Aebeloe, D. Dell'Aglio, A. Keen and K. Hose, StarBench: Benchmarking RDF-star Triplestores, in: *QuWeDa/MEPDaW@ISWC*, CEUR Workshop Proceedings, Vol. 3565, CEUR-WS.org, 2023, pp. 34–49.

[40] T. Neumann and G. Weikum, RDF-3X: A RISC-Style Engine for RDF, *Proc. VLDB Endow.* **1**(1) (2008), 647–659–. doi:10.14778/1453856.1453927.

[41] Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris and Y. Stavrakas, A Flexible Framework for Understanding the Dynamics of Evolving RDF Datasets, in: *International Semantic Web Conference (ISWC)*, 2015. doi:10.1007/978-3-319-25007-6_29.

[42] I. Cuevas and A. Hogan, Versioned Queries over RDF Archives: All You Need is SPARQL?, in: *MEPDaW@ISWC*, CEUR Workshop Proceedings, Vol. 2821, CEUR-WS.org, 2020, pp. 43–52.

[43] M.V. Sande, P. Colpaert, R. Verborgh, S. Coppens, E. Mannens and R.V. de Walle, R&Wbase: git for triples, in: *LDOW*, CEUR Workshop Proceedings, Vol. 996, CEUR-WS.org, 2013.

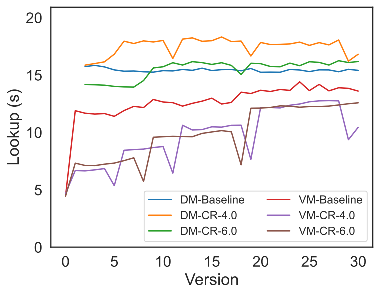
[44] D. Im, S. Lee and H. Kim, A Version Management Framework for RDF Triple Stores, *Int. J. Softw. Eng. Knowl. Eng.* **22**(1) (2012), 85–106.

[45] D. Dell'Aglio, E.D. Valle, J. Calbimonte and Ó. Corcho, RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems, *Int. J. Semantic Web Inf. Syst.* **10**(4) (2014), 17–44.

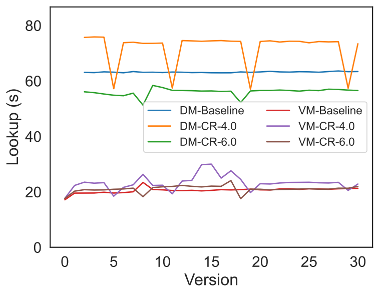
[46] R.T. Snodgrass, I. Ahn, G. Ariav, D.S. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W. Käfer, N. Kline, K.G. Kulkarni, T.Y.C. Leung, N.A. Lorentzos, J.F. Roddick, A. Segev, M.D. Soo and S.M. Sripada, TSQL2 Language Specification, *SIGMOD Record* **23**(1) (1994), 65–86. doi:10.1145/181550.181562.

[47] O. Pelgrin, L. Galárraga and K. Hose, TrieDF: Efficient In-memory Indexing for Metadata-augmented RDF, in: *MEPDaW@ISWC*, CEUR Workshop Proceedings, Vol. 3225, CEUR-WS.org, 2021, pp. 20–29.

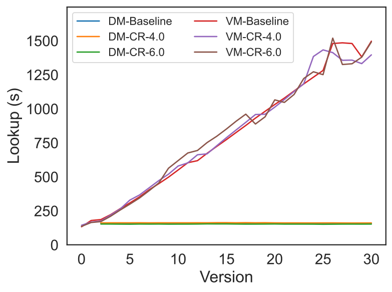
Appendix A. Additional SPARQL Results



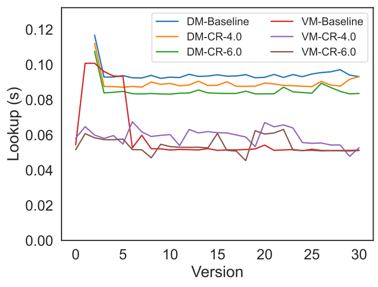
(a) BEAR-C Query #3



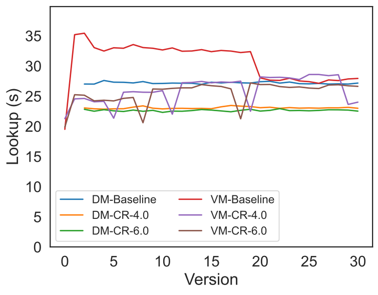
(b) BEAR-C Query #4



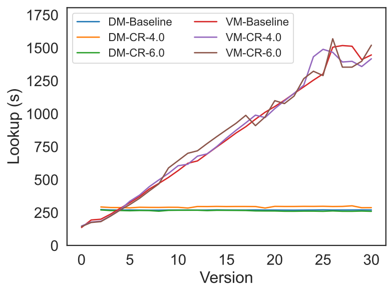
(c) BEAR-C Query #5



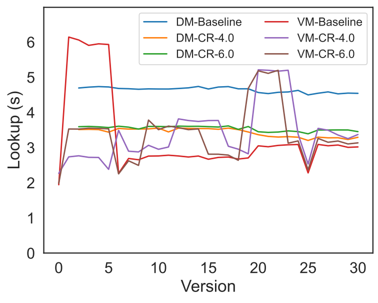
(d) BEAR-C Query #6



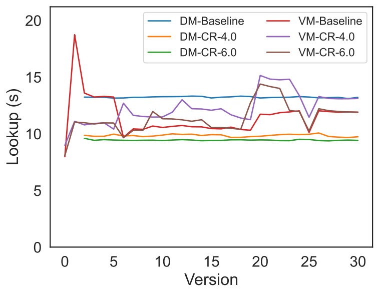
(e) BEAR-C Query #7



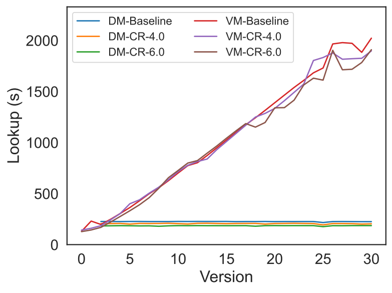
(f) BEAR-C Query #8



(g) BEAR-C Query #9



(h) BEAR-C Query #10



(i) BEAR-C Query #11

Fig. 11. Individual runtime of DM and VM SPARQL queries for BEAR-C.